MAGEPLAZA

Magento 2

Module Development

CREDITS

Magento expert: Brian Tran.

Editor: Tracy Vu.

Assistant Editor: Linda Thompson.

Design: Sam Thomas.

Special thanks to: LiLy White, Brian Tran.

Also thanks to Mageplaza team who supported a lot in the release of this special Module Development ebook for Magento 2.

TABLE OF CONTENTS

OVERVIEW

Magento 2 Module development trend is increasing rapidly while Magento releases the official version. That why we - Mageplaza - are writing about series of topics that introduces how to create a simple Hello World module in Magento 2.

TOPIC 1: MAGENTO 2 HELLO WORLD

As you know, the module is a directory that contains `blocks, controllers, models, helper`, etc - which are related to a specific business feature. In Magento 2, modules will be live in `app/code` directory of a Magento installation, with this format: `app/code/<Vendor>/<ModuleName>`.

Now we will follow this steps to create a simple module which works on Magento 2 and display `Hello World`.

To create Hello World module in Magento 2

- Step 1: Create a directory for the module like above format.
- Step 2: Declare module by using configuration file module.xml
- Step 3: Register module by registration.php
- Step 4: Enable the module
- Step 5: Create a Routers for the module.
- Step 6: Create controller and action.

Step 1. Create a directory for the module like above format

In this module, we will use `Mageplaza` for Vendor name and `HelloWorld` for ModuleName. So we need to make this folder: `app/code/Mageplaza/HelloWorld`

Step 2. Declare module by using configuration file module.xml

Magento 2 looks for configuration information for each module in that module's etc directory. We need to create folder etc and add module.xml:

```
app/code/Mageplaza/HelloWorld/etc/module.xml
```

And the content for this file:

```
<?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="Mageplaza_HelloWorld" setup_version="1.0.0" />
</config>
```

In this file, we register a module with name `Mageplaza_HelloWorld` and the version is `1.0.0`.

Step 3. Register module by registration.php

All Magento 2 module must be registered in the Magento system through the Magento Component Registrar class. This file will be placed in module root directory. In this step, we need to create this file:

```
app/code/Mageplaza/HelloWorld/registration.php
```

And it’s content for our module is:

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

Step 4. Enable the module

By finish above step, you have created an empty module. Now we will enable it in Magento environment. Before enabling the module, we must check to make sure Magento has recognized our module or not by entering the following at the command line:

```
php bin/magento module:status
```

If you follow above step, you will see this in the result:

```
List of disabled modules:
Mageplaza_HelloWorld
```

This means the module has recognized by the system but it is still disabled. Run this command to enable it:

```
php bin/magento module:enable Mageplaza_HelloWorld
```

The module has enabled successfully if you saw this result:

```
The following modules has been enabled:
- Mageplaza_HelloWorld
```

This is the first time you enable this module so Magento requires to check and upgrade module database. We need to run this comment:

```
php bin/magento setup:upgrade
```

Now you can check under `Stores -> Configuration -> Advanced -> Advanced` that the module is present.

Step 5. Create a Routers for the module.

In the Magento system, a request URL has the following format:

```
http://example.com/<router_name>/<controller_name>/<action_name>
```

The Router is used to assign a URL to a corresponding controller and action. In this module, we need to create a route for frontend area. So we need to add this file:

```
app/code/Mageplaza/HelloWorld/etc/frontend/routes.xml
```

And content for this file:

```
<?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="mageplaza" frontName="helloworld">
            <module name="Mageplaza_HelloWorld" />
        </route>
    </router>
</config>
```

After define the route, the URL path to our module will be:
`http://example.com/helloworld/*`

Step 6. Create controller and action.

In this step, we will create controller and action to display `Hello World`. Now we will choose the url for this action. Let assume that the url will be:
`http://example.com/helloworld/index/display`

So the file we need to create is:

```
app/code/Mageplaza/HelloWorld/Controller/Index/Display.php
```

And we will put this content:

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

class Display extends \Magento\Framework\App\Action\Action
```

```
{
  public function __construct(
\Magento\Framework\App\Action\Context $context)
  {
    return parent::__construct($context);
  }

  public function execute()
  {
    echo 'Hello World';
    exit;
  }
}
```

If you have followed all above steps, you will see `Hello World` when opening the url `http://example.com/helloworld/index/display`

TOPIC 2: MAGENTO 2 CREATE VIEW: BLOCK, LAYOUT, TEMPLATES

In this topic Magento 2 Create: Block, Layouts, Templates we will learn about View in Magento 2 including Block, Layouts, and Templates. In the previous topic, we discussed CRUD Models. As you know, a View will be used to output representation of the page. In Magento 2, View is built by three paths: block, layout, and template. We will find how it works by building the simple module Hello World using View path.

To create view in Magento 2

- Step 1: Call view in controller
- Step 2: Declare layout file
- Step 3: Create block
- Step 4: Create template file

Step 1: Call view in controller

In the previous Magento 2 Hello World topic, we have built a simple module and show the `Hello World` message on the screen directly by the controller. Now we will edit it to call view to render page.

```
#file: app/code/Mageplaza/HelloWorld/Controller/Index/Display.php
<?php
namespace Mageplaza\HelloWorld\Controller\Index;

class Display 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();
        }
}
```

We have to declare the PageFactory and create it in execute method to render view.

Step 2: Declare layout file

The Layout is the major path of view layer in Magento 2 module. The layout file is an XML file which will define the page structure and will be located in `module_root}/view/{area}/layout/` folder. The Area path can be *frontend* or *adminhtml* which define where the layout will be applied.

There is a special layout file name `default.xml` which will be applied to all the page in its area. Otherwise, the layout file will have name as format: `{router_id}_{controller_name}_{action_name}.xml`.

You can understand the layout in detail in this Magento topic , and the instruction of a layout structure.

When rendering page, Magento will check the layout file to find the handle for the page and then load Block and Template. We will create a layout handle file for this module:

```
#file: app/code/Mageplaza/HelloWorld/view/frontend/layout/helloworld_index_display.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_configuratio
n.xsd">
    <referenceContainer name="content">
        <block class="Mageplaza\HelloWorld\Block\Display" name="helloworld_display"
template="Mageplaza_HelloWorld::sayhello.phtml" />
    </referenceContainer>
</page>
```

In this file, we define the block and template for this page:

Block class: `Mageplaza\HelloWorld\Block\Display` Template file: `Mageplaza_HelloWorld::sayhello.phtml`

Step 3: Create block

The Block file should contain all the view logic required, it should not contain any kind of html or css. Block file are supposed to have all application view logic.

Create a file:

```
app/code/Mageplaza/HelloWorld/Block/Display.php
```

The Block for this module:

```
<?php
namespace Mageplaza\HelloWorld\Block;
class Display extends \Magento\Framework\View\Element\Template
{
	public function __construct(\Magento\Framework\View\Element\Template\Context
$context)
	{
		parent::__construct($context);
	}

	public function sayHello()
	{
		return __('Hello World');
	}
}
```

Every block in Magento 2 must extend from `Magento\Framework\View\Element\Template`. In this block, we will define a method sayHello() to show the word “Hello World”. We will use it in the template file.

Step 4. Create template file

Create a template file call `sayhello.phtml`

```
app/code/Mageplaza/HelloWorld/view/frontend/templates/sayhello.phtml
```

Insert the following code:

```
<?php
echo $block->sayHello();
```

In the layout file, we define the template by `Mageplaza_HelloWorld::sayhello.phtml`. It mean that Magento will find the file name sayhello.phtml in templates folder of module Mageplaza_HelloWorld. The template folder of the module is `app/code/{vendor_name}/{module_name}/view/frontend/templates/`.

In the template file, we can use the variable $block for the block object. As you see, we call the method `sayHello()` in Block. It’s done, please access to this page again (http://example.com/helloworld/index/display) and see the result.

TOPIC 3: CRUD MODELS IN MAGENTO 2

CRUD Models in Magento 2 can manage data in the database easily, you do not need to write many lines of code to create a CRUD. CRUD to stand for Create, Read, Update and Delete. We will learn about some main contents: How to setup Database, Model, Resource Model and Resource Magento 2 Get Collection and do database related operations.

Before learning this topic, let's decide how the table which we work with will look. I will create a table `mageplaza_topic` and take the following columns:

- `topic_id` - the topic unique identifier
- `title` - the title of the topic
- `content` - the content of the topic
- `creation_time` - the date created

To create Model in Magento 2

- Step 1: Setup Script
- Step 2: Model
- Step 3: Resource Model
- Step 4: Resource Model Collection
- Step 5: Factory Object

Step 1: Setup Script

Firstly, we will create the database table for our CRUD models. To do this we need to insert the setup file:

`app/code/Mageplaza/HelloWorld/Setup/InstallSchema.php`

This file will execute only one time when installing the module. Let put this content for this file to create above table:

```
namespace Mageplaza\HelloWorld\Setup;
class InstallSchema implements \Magento\Framework\Setup\InstallSchemaInterface
{
    public function install(
        \Magento\Framework\Setup\SchemaSetupInterface $setup,
        \Magento\Framework\Setup\ModuleContextInterface $context)
    {
        $installer = $setup;
        $installer->startSetup();
        $table = $installer->getConnection()->newTable(
            $installer->getTable('mageplaza_topic')
        )->addColumn(
            'topic_id',
            \Magento\Framework\DB\Ddl\Table::TYPE_SMALLINT,
            null,
            ['identity' => true, 'nullable' => false, 'primary' => true, 'unsigned' =>
true],
            'Topic ID'
        )->addColumn(
            'title',
            \Magento\Framework\DB\Ddl\Table::TYPE_TEXT,
            255,
            ['nullable' => false],
            'Topic Title'
        )->addColumn(
            'content',
            \Magento\Framework\DB\Ddl\Table::TYPE_TEXT,
            '2M',
            [],
            'Topic Content'
        )->addColumn(
            'creation_time',
            \Magento\Framework\DB\Ddl\Table::TYPE_TIMESTAMP,
            null,
            ['nullable' => false, 'default' =>
\Magento\Framework\DB\Ddl\Table::TIMESTAMP_INIT],
            'Topic Creation Time'
        )->setComment(
            'Mageplaza Topic Table'
        );
        $installer->getConnection()->createTable($table);
        $installer->endSetup();
    }
}
```

This content is showing how the table created, you can edit it to make your own table. Please note that Magento will automatically run this file for the first time when installing the module. If you installed the module before, you will need to upgrade module and write the table create code to the UpgradeSchema.php in that folder.

After this please run this command line:

```
php bin/magento setup:upgrade
```

Now checking your database, you will see a table with name 'mageplaza_topic' and above columns. If this table is not created, it may be because you ran the above command line before you add content to InstallSchema.php. To fix this, you need to remove the information that let Magento know your module has installed in the system. Please open the table 'setup_module', find and remove a row has module equals to 'mageplaza_topic'. After this, run the command again to install the table.

This `InstallSchema.php` is used to create database structure. If you want to install the data to the table which you was created, you need to use `InstallData.php` file:

```
app/code/Mageplaza/HelloWorld/Setup/InstallData.php
```

Please take a look in some InstallData file in Magento to know how to use it. This's some file you can see:

```
- vendor/magento/module-tax/Setup/InstallData.php
- vendor/magento/module-customer/Setup/InstallData.php
- vendor/magento/module-catalog/Setup/InstallData.php
```

As I said above, those install file will be used for the first time install the module. If you want to change the database when upgrading module, please try to use `UpgradeSchema.php` and `UpgradeData.php`.

Step 2: Model

Model is a huge path of MVC architecture. In Magento 2 CRUD, models have many different functions such as manage data, install or upgrade module. In this tutorial, I only talk about data management CRUD. We have to create Model, Resource Model, Resource Model Collection to manage data in the table: `mageplaza_topic` as I mentioned above.

Before create model, we need to create the interface for it. Let create the `TopicInterface`:

```
app/code/Mageplaza/HelloWorld/Model/Api/Data/TopicInterface.php
```

And put this content:

```
<?php
namespace Mageplaza\HelloWorld\Model\Api\Data;
interface TopicInterface
{
       public function getId();
       public function setId();
```

```
        public function getTitle();
        public function setTitle();

        public function getContent();
        public function setContent();

        public function getCreationTime();
        public function setCreationTime();
}
```

This interface has defined the set and get method to table data which we would use when interacting with the model. This interface plays an important role when it comes time to exporting CRUD models to Magento service contracts based API.

Now we will create the model file:

```
app/code/Mageplaza/HelloWorld/Model/Topic.php
```

And this is the content of that file:

```
<?php
namespace Mageplaza\HelloWorld\Model;
class Topic extends \Magento\Framework\Model\AbstractModel implements
\Magento\Framework\DataObject\IdentityInterface,
\Mageplaza\HelloWorld\Model\Api\Data\TopicInterface
{
        const CACHE_TAG = 'mageplaza_topic';

        protected function _construct()
        {
                $this->_init('Mageplaza\HelloWorld\Model\ResourceModel\Topic');
        }

        public function getIdentities()
        {
                return [self::CACHE_TAG . '_' . $this->getId()];
        }
}
```

This model class will extends AbstractModel class `Magento\Framework\Model\AbstractModel` and implements `TopicInterface` and `IdentityInterface\Magento\Framework\DataObject\IdentityInterface`.

The IdentityInterface will force Model class to define the `getIdentities()` method which will return a unique id for the model. You must only use this interface if your model required cache refresh after database operation and render information to the frontend page.

The `_construct()` method will be called whenever a model is instantiated. Every CRUD model has to use the _construct() method to call _init() method. This _init() method will define the resource model which will actually fetch the information from the database. As above, we define the resource model Mageplaza\Topic\Model\ResourceModel\Topic The last thing about model is some variable which you should you in your model:

- `$_eventPrefix` - a prefix for events to be triggered
- `$_eventObject` - an object name when access in event
- `$_cacheTag` - a unique identifier for use within caching

Step 3: Resource Model

As you know, the model file contains overall database logic, it does not execute SQL queries. The resource model will do that. Now we will create the Resource Model for this table: `Mageplaza\HelloWorld\Model\ResourceModel\Topic`

Content for this file:

```
<?php
namespace Mageplaza\HelloWorld\Model\ResourceModel;
class Topic extends \Magento\Framework\Model\ResourceModel\Db\AbstractDb
{
	protected function _construct()
	{
		$this->_init('mageplaza_topic', 'topic_id');
	}
}
```

Every CRUD resource model in Magento must extend abstract class `\Magento\Framework\Model\ResourceModel\Db\AbstractDb` which contains the functions for fetching information from the database.

Like model class, this resource model class will have required method `_construct()`. This method will call `_init()` function to define the table name and primary key for that table. In this example, we have table 'mageplaza_topic' and the primary key 'topic_id'.

Step 4: Resource Model Collection - Get Model Collection

The collection model is considered a resource model which allow us to filter and fetch a collection table data. The collection model will be placed in:

```
Mageplaza\HelloWorld\Model\ResourceModel\Topic\Collection.php
```

The content for this file:

```
<?php
namespace Mageplaza\HelloWorld\Model\ResourceModel\Topic;
class Collection extends
\Magento\Framework\Model\ResourceModel\Db\Collection\AbstractCollection
{
	protected function _construct()
	{
		$this->_init('Mageplaza\HelloWorld\Model\Topic',
'Mageplaza\HelloWorld\Model\ResourceModel\Topic');
	}
}
```

The CRUD collection class must extends from `\Magento\Framework\Model\ResourceModel\Db\Collection\AbstractCollection` and call the `_init()` method to init the model, resource model in `_construct()` function.

Step 5: Factory Object

We are done with creating the database table, CRUD model, resource model and collection. So how to use them?

In this part, we will talk about Factory Object for the model. As you know in OOP, a factory method will be used to instantiate an object. In Magento, the Factory Object does the same thing.

The Factory class name is the name of Model class and appends with the 'Factory' word. So for our example, we will have TopicFactory class. You must not create this class. Magento will create it for you. Whenever Magento's object manager encounters a class name that ends in the word 'Factory', it will automatically generate the Factory class in the var/generation folder if the class does not already exist. You will see the factory class in

```
var/generation/<vendor_name>/<module_name>/Model/ClassFactory.php
```

To instantiate a model object we will use automatic constructor dependency injection to inject a factory object, then use a factory object to instantiate the model object.

For example, we will call the model to get data in Block. We will create a Topic block:

`Mageplaza\HelloWorld\Block\Topic.php`

Content for this file:

```
<?php
namespace Mageplaza\HelloWorld\Block;
class Topic extends \Magento\Framework\View\Element\Template
{
	protected $_topicFactory;
	public function _construct(
		\Magento\Framework\View\Element\Template\Context $context,
		\Mageplaza\HelloWorld\Model\TopicFactory $topicFactory
	){
		$this->_topicFactory = $topicFactory;
		parent::_construct($context);
	}

	public function _prepareLayout()
	{
		$topic = $this->_topicFactory->create();
		$collection = $topic->getCollection();
		foreach($collection as $item){
			var_dump($item->getData());
		}
		exit;
	}
}
```

As you see in this block, the TopicFactory object will be created in the _construct() function. In the _prepareLayout() function, we use `$topic = $this->_topicFactory->create();` to create the model object.

TOPIC 4: HOW TO CREATE CONTROLLERS IN MAGENTO 2

Controller especially is one of the important things in Magento 2 module development and PHP MVC Framework in general. Its functionality 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 to workflow, but admin controller is a little different. 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.

How controller work?

It receives a request from end-user (browser or command line), for example:

```
http://example.com/route_name/controller/action
```

- `route_name` is a unique name which is set in routes.xml.
- `controller` is the folder inside Controller folder.
- `action` is a class with execute method to process request.

One of the important in Magento system is frontController (`Magento\Framework\App\FrontController`), it alway receive request then route controller, action by`route_name` Let take an example of routing a 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?

To create a controller, we need to create a folder inside `Controller` folder of the module and declare an action class inside it. For example, we create a `Test controller` and a `Hello` action for module `Mageplaza_HelloWorld`:

```
app/code/Mageplaza/HelloWorld/Controller/Test/SayHello.php
```

And content of this file should be:

```
namespace Mageplaza\HelloWorld\Controller\Test;
class SayHello extends \Magento\Framework\App\Action\Action
{
        public function execute()
        {
                echo 'Hello World! Welcome to Mageplaza.com';
                exit;
        }
}
```

As you see, all controller must extend from `\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 the response for the request.

Forward and redirect in action

`\Magento\Framework\App\Action\Action` class provides 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 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 is the result will be displayed 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 changed to `http://example.com/route_name/test/hello` and show the message `Hello World! Welcome to Mageplaza.com` on the screen.

How to rewrite controller in Magento 2

To rewrite controller, you can do it by using preference. It means that you need to put a rule in your router config using before attribute.

Open `Mageplaza/HelloWorld/etc/di.xml` insert the following block of code inside `<config>` tag rewrite controller in Magento 2

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

<router id="standard">
    <route id="mageplaza" frontName="hello">
       <module name="Mageplaza_HelloWorld" before="Magento_Customer" />
    </route>
</router>

</config>
```

This will completely change `controller/action` of module `Magento_Customer` with your controller code, so you should extend rewrite controller and make a change on the function which you want. Also, the controller and action in your module must have the same name with rewrite `controller/action`. For example, if you want to rewrite controller: `Magento\Customer\Controller\Account\Create.php`

You have to register a router like above and create a controller:

```
NameSpace\ModuleName\Controller\Account\Create.php
```

Content of Create.php file:

```
namespace Mageplaza\HelloWorld\Controller\Account;

use Magento\Customer\Model\Registration;
use Magento\Customer\Model\Session;
use Magento\Framework\View\Result\PageFactory;
use Magento\Framework\App\Action\Context;

class Create extends \Magento\Customer\Controller\AbstractAccount
```

```
{
    /** @var Registration */
    protected $registration;

    /**
     * @var Session
     */
    protected $session;

    /**
     * @var PageFactory
     */
    protected $resultPageFactory;

    /**
     * @param Context $context
     * @param Session $customerSession
     * @param PageFactory $resultPageFactory
     * @param Registration $registration
     */
    public function __construct(
        Context $context,
        Session $customerSession,
        PageFactory $resultPageFactory,
        Registration $registration
    ) {
        $this->session = $customerSession;
        $this->resultPageFactory = $resultPageFactory;
        $this->registration = $registration;
        parent::__construct($context);
    }

    /**
     * Customer register form page
     *
     * @return
\Magento\Framework\Controller\Result\Redirect|\Magento\Framework\View\Result\Page
     */
    public function execute()
    {
        if ($this->session->isLoggedIn() || !$this->registration->isAllowed()) {
            /** @var \Magento\Framework\Controller\Result\Redirect $resultRedirect */
            $resultRedirect = $this->resultRedirectFactory->create();
            $resultRedirect->setPath('*/*');
            return $resultRedirect;
        }

        /** @var \Magento\Framework\View\Result\Page $resultPage */
        $resultPage = $this->resultPageFactory->create();
        return $resultPage;
    }
}
```

TOPIC 5: MAGENTO 2 HOW TO CREATE System.xml CONFIGURATION

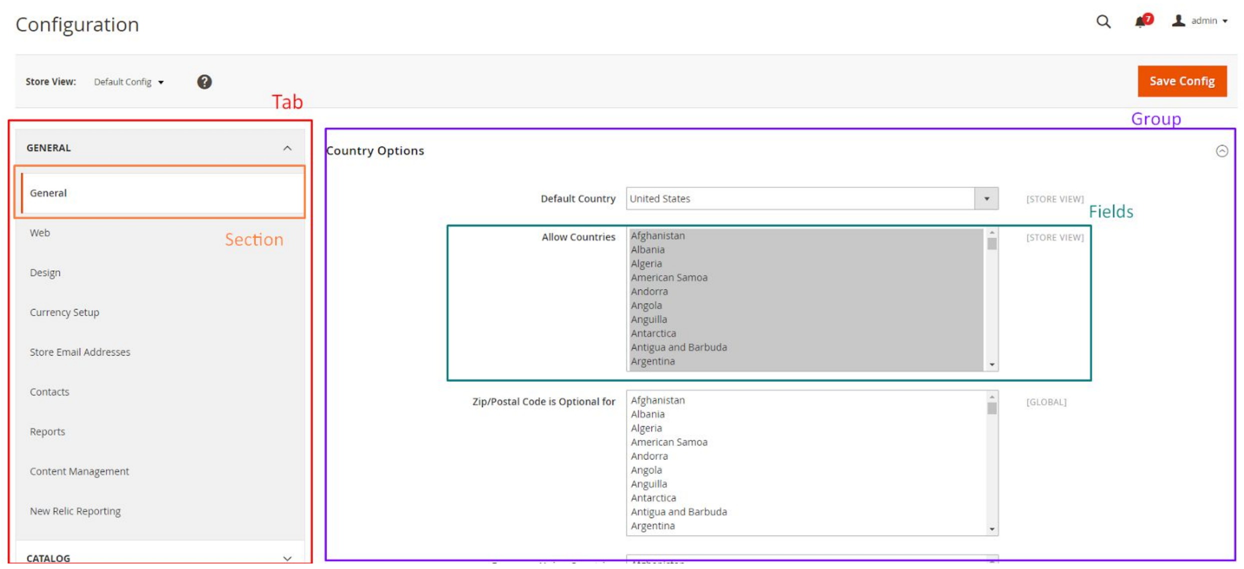
The system.xml is a configuration file which is used to create configuration fields in Magento 2 System Configuration. You will need this if your module has some settings which the admin needs to set. You can go to `Store -> Setting -> Configuration` to check how it looks like.

To Create system.xml

- Step 1: Create System.xml
- Step 2: Set default value
- Step 3: Flush Magento cache

Step 1: Create System.xml

The magento 2 system configuration page is divided logically into few parts: Tabs, Sections, Groups, Fields. Please check this images to understand about this:



Store View: Default Config

Default Country: United States [STORE VIEW]

Allow Countries: Afghanistan, Albania, Algeria, American Samoa, Andorra, Angola, Anguilla, Antarctica, Antigua and Barbuda, Argentina [STORE VIEW]

Zip/Postal Code is Optional for: Afghanistan, Albania, Algeria, American Samoa, Andorra, Angola, Anguilla, Antarctica, Antigua and Barbuda, Argentina [GLOBAL]

So let's start to create a simple configuration for the simple Module Hello World. The system.xml is located in `etc/adminhtml` folder of the module, we will create it a new Tab for our vendor "Mageplaza", a new Section for our module Hello World, a Group to contain some simple fields: enable module and text.

File: `app/code/Mageplaza/HelloWorld/etc/adminhtml/system.xml`

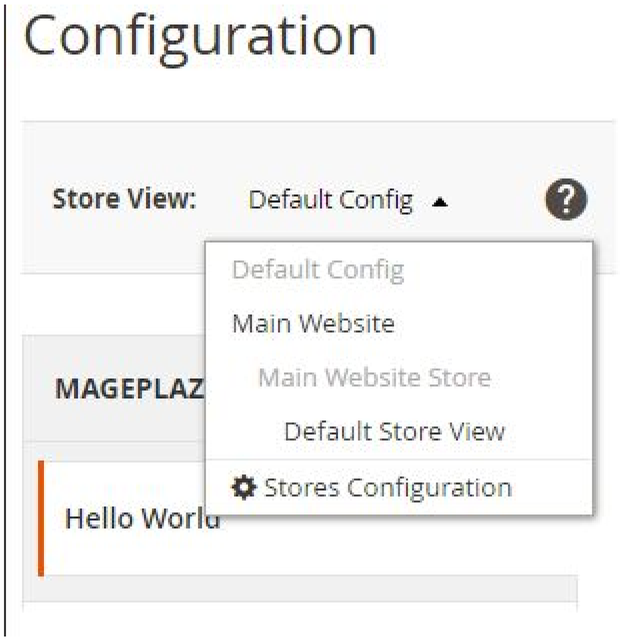
```
<?xml version="1.0"?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:magento:module:Magento_Config:etc/system_file.xsd">
    <system>
        <tab id="mageplaza" translate="label" sortOrder="10">
            <label>Mageplaza</label>
        </tab>
        <section id="hello" translate="label" sortOrder="130" showInDefault="1"
showInWebsite="1" showInStore="1">
            <class>separator-top</class>
            <label>Hello World</label>
            <tab>mageplaza</tab>
            <resource>Mageplaza_HelloWorld::hello_configuration</resource>
            <group id="general" translate="label" type="text" sortOrder="10"
showInDefault="1" showInWebsite="0" showInStore="0">
                <label>General Configuration</label>
                <field id="enable" translate="label" type="select" sortOrder="1"
showInDefault="1" showInWebsite="0" showInStore="0">
                    <label>Module Enable</label>

<source_model>Magento\Config\Model\Config\Source\Yesno</source_model>
                </field>
                <field id="display_text" translate="label" type="text" sortOrder="1"
showInDefault="1" showInWebsite="0" showInStore="0">
                    <label>Display Text</label>
                    <comment>This text will display on the frontend.</comment>
                </field>
            </group>
        </section>
    </system>
</config>
```

Checking this code, you will see how to create a Tab, Section, Group, and Field. We will find more detail about each element:

- The Tab element may have many sections and some main attributes and child:
    - Id attribute is the identify for this tab
    - sortOrder attribute will define the position of this tab.

        - Translate attribute lets Magento know which title need to translate
        - Label element child is the text which will show as tab title.
    - The Section element will have an id, sortOrder, translate attributes like the Tab element. Some other attributes (showInDefault, showInWebsite, showInStore) will decide this element will be show on each scope or not. You can change the scope here



The section may have many groups and some other child elements:

- `Class`: this value will be added as a class for this element. You should use it if you want to make-up this element
- `Label`: the text title of this element
- `Tab`: this's a tab id. This tab element will let Magento know the tab which this section belongs to. This section will be placed under that tab
- `Resource`: defined the ACL rule which the admin user must have in order to access this configuration
- `Group`: This element may have many fields and some attributes which are as same as Sections
- `Fields`: is the main path of this page. It will save the data which we want to set. In this element, we focus on the type attribute. It will define how the element is when display. It can be: text, select, file... In this example, we

create 2 fields with type select and text. With each type, we will define the child element for the field to make it work as we want.

For example, with the type `select/multiselect` you must define the child element `resource_model`.

Step 2: Set default value

Each field in system.xml after create will not have any value. When you call them, you will receive 'null' result. So for the module, we will need to set the default value for the field and you will call the value without go to config, set value and save it. This default value will be saved in config.xml which is located in etc folder. Let's create it for this simple configuration:

File: `app/code/Mageplaza/HelloWorld/etc/config.xml`

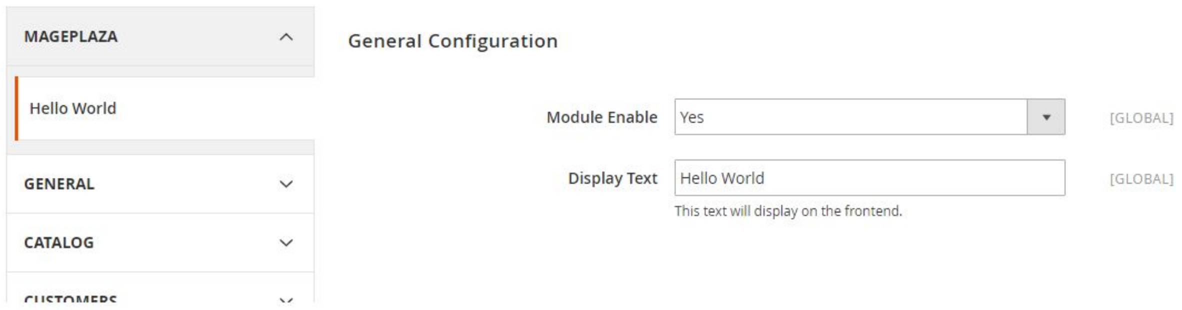
```
<?xml version="1.0"?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:magento:module:Magento_Store:etc/config.xsd">
    <default>
        <hello>
            <general>
                <enable>1</enable>
                <display_text>Hello World</display_text>
            </general>
        </hello>
    </default>
</config>
```

You can put the path to the field in the `<default>` element to set value default for it. The format is:

```
<default>
    <section>
        <group>
            <field>{value}</field>
        </group>
    </section>
</default>
```

Step 3: Flush Magento Cache

Now, please refresh your cache and see the result:



General Configuration

Module Enable: Yes [GLOBAL]

Display Text: Hello World [GLOBAL]

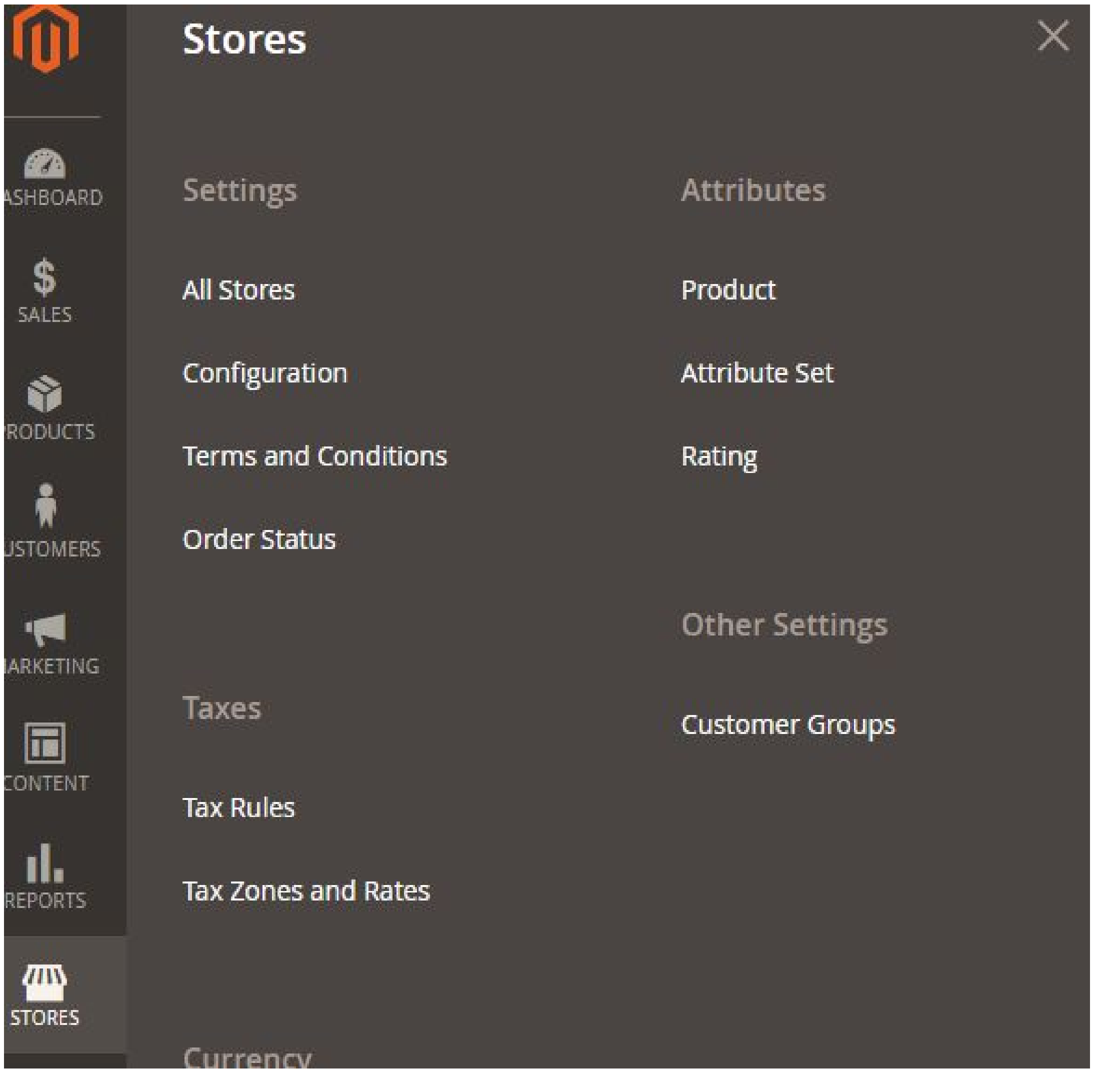
This text will display on the frontend.

Note that, if you might get an Error 404 Page Not Found the first time, just logout and login again to fix this.

TOPIC 6: MAGENTO 2 CREATE ADMIN MENU

In this article, we will find how to add a link to admin menu in Magneto 2, which shown on the left site of Admin pages of Magento 2.

Firstly, we will find out the structure of the admin menu and how the action in each menu like. The structure of the menu is separated by level. You will see the level-0 on the left bar and the higher level is grouped and shown when you click on the level-0 menu. For example, this image is a menu of Stores. You will see the Stores is a level-0 and show on the left bar. When you click on it, the sub-menu will show up like: Setting, Attributes, Taxes… and that sub-menu has some sub-sub-menu also (Setting has All Stores, Configuration, Terms and Conditions, Order Status).



Checking the url of each menu we will see it like this:

```
http://example.com/admin/catalog/product_attribute/index/key/aa6db988cd4bf9fb91c5b3fe4
f5e49ffc0df4c5666aa13f5912b50752fafbea5/
```

Like on the frontend, we will have this format `{router_name}_{controller_folder}_{action_name}`. But in the admin menu, we will have an admin router name (this can be customized) before which made Magento know this's a Backend area.

So how the menu created? We will use the simple module Hello World which was created in the previous topic to create a menu.

In Magento 2, we use the menu.xml to add this menu. Let create it.

To Create Admin Menu in Magento 2

- Step 1: Create menu.xml
- Step 2: Add menu item
- Step 3: Flush Magento cache

Step 1: Create menu.xml

Create admin menu file called: menu.xml file

```
app/code/Mageplaza/HelloWorld/etc/adminhtml/menu.xml
```

with the following content:

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

Step 2: Add menu item

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

```
xsi:noNamespaceSchemaLocation="urn:magento:module:Magento_Backend:etc/menu.xsd">
    <menu>
        <add id="Mageplaza_HelloWorld::hello" title="Hello World"
module="Mageplaza_HelloWorld" sortOrder="50" resource="Mageplaza_HelloWorld::hello"/>
        <add id="Mageplaza_HelloWorld::hello_manage_items" title="Manage Items"
module="Mageplaza_HelloWorld" sortOrder="50" parent="Mageplaza_HelloWorld::hello"
action="adminhtml/hello" resource="Mageplaza_HelloWorld::hello_manage_items"/>
        <add id="Mageplaza_HelloWorld::hello_configuration" title="Configuration"
module="Mageplaza_HelloWorld" sortOrder="50" parent="Mageplaza_HelloWorld::hello"
action="adminhtml/system_config/edit/section/hello"
resource="Mageplaza_HelloWorld::hello_configuration"/>
    </menu>
</config>
```

In this example, we will create a level-0 menu named “Hello World” and two sub-menus named “Manage Items” and “Configuration”. The menu.xml file will define a collection of ‘add’ note which will add a menu item to Magento backend. We will see its structure:

```
<add id="Mageplaza_HelloWorld::hello_manage_items" title="Manage Items"
module="Mageplaza_HelloWorld" sortOrder="50" parent="Mageplaza_HelloWorld::hello"
action="helloworld/hello" resource="Mageplaza_HelloWorld::hello_manage_items"/>
```

Let’s explain some attributes:

- The `id` attribute is the identifier for this note. It’s a unique string and should follow the format: {Vendor_ModuleName}::{menu_description}.
- The `title` attribute is the text which will be shown on the menu bar.
- The `module` attribute defines the module which this menu belongs to.
- The `sortOrder` attribute defines the position of the menu. A lower value will display on top of the menu.
- The `parent` attribute is an Id of another menu node. It will tell Magento that this menu is a child of another menu. In this example, we have parent=”Mageplaza_HelloWorld::hello”, so we - know this menu “Manage Items” is a child of “Hello World” menu and it will show inside of Hello World menu.
- The `action` attribute will define the url of the page which this menu link to. As we talk above, the url will be followed this format {router_name}*{controller_folder}*{action_name}. - In this example, this menu will link to the module HelloWorld, controller Hello, and action Index

- The `resource` attribute is used to defined the ACL rule which the admin user must have in order to see and access this menu. We will find more detail about ACL in another topic.

You can also create more child menus and it will show like Store menu above.

I want to talk about the icon on the top menu level. You can see them above the level-0 menu title. This icon is generated by 'Admin Icons' font in Magento. You can see all of the icon and how to create an icon in this link.

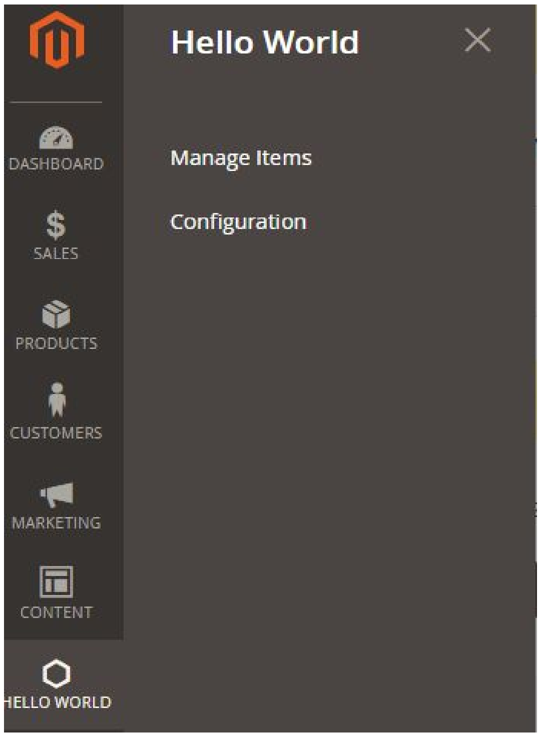
Step 3: Flush Magento cache

Make sure it admin menu items are displayed on Magento 2 admin, you should try to flush Magento 2 cache.

Run the following command line:

```
php bin/magento cache:clean
```

Now to go Magento 2 Admin and see result:



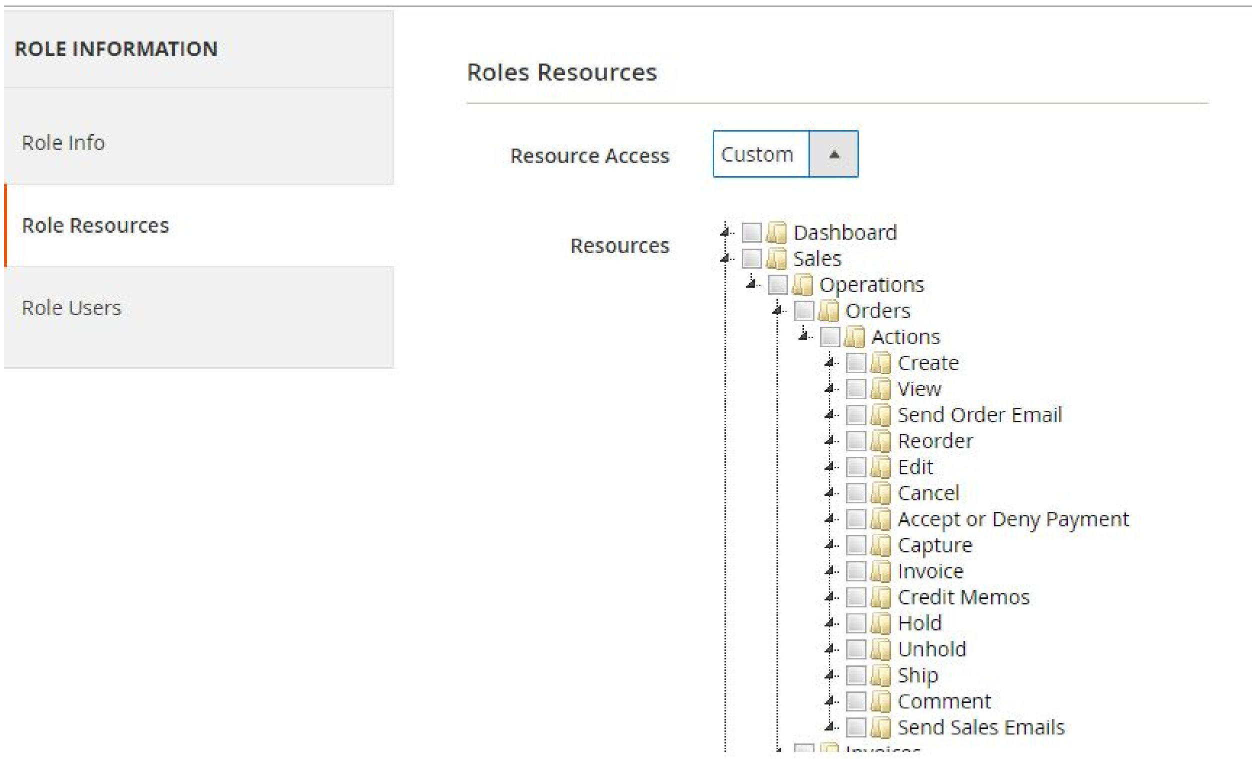
You may get `404 Not Found` message, just logout then login again to fix this.

TOPIC 7: MAGENTO 2 ADMIN ACL ACCESS CONTROL LISTS

Magento 2 admin panel use an authentication system and a robust system for creating Access Control List Rules (ACL) which allows a store owner to create fine grained roles for each and every user in their system. In this article, we will find how it work and how to add ACL for our custom module.

Magento 2 Access Control List Rules

The Magento 2 Admin ACL resources are visible under the Magento 2 admin `System > Permissions > User Roles` area. When we click on the Add New Role button or access to a role, you will see the page look like:



Resource Access: Custom

Resources:
- Dashboard
- Sales
  - Operations
    - Orders
      - Actions
        - Create
        - View
        - Send Order Email
        - Reorder
        - Edit
        - Cancel
        - Accept or Deny Payment
        - Capture
        - Invoice
        - Credit Memos
        - Hold
        - Unhold
        - Ship
        - Comment
        - Send Sales Emails

In this resources tab, you can see a tree-list of all the available resources in your system. You can choose all Resource or some of them for this role and select the user for this role in Role Users tab. All of the users who belong to this role will be limit access to the resource which you choose. They cannot see and access to another one.

Create ACL rule

Now, we will see how to add our module to ACL role. We will use a previous simple module Hello World to do this. As in the Admin Menu and System Configuration article, you saw that we alway have a resource attribute when creating it. Now we will register that resources to the system, so Magento can realize and let us set a role for them. To register the resource, we use the acl.xml file which located in `{module_root}/etc/acl.xml`. Let's create this file for our simple Module:

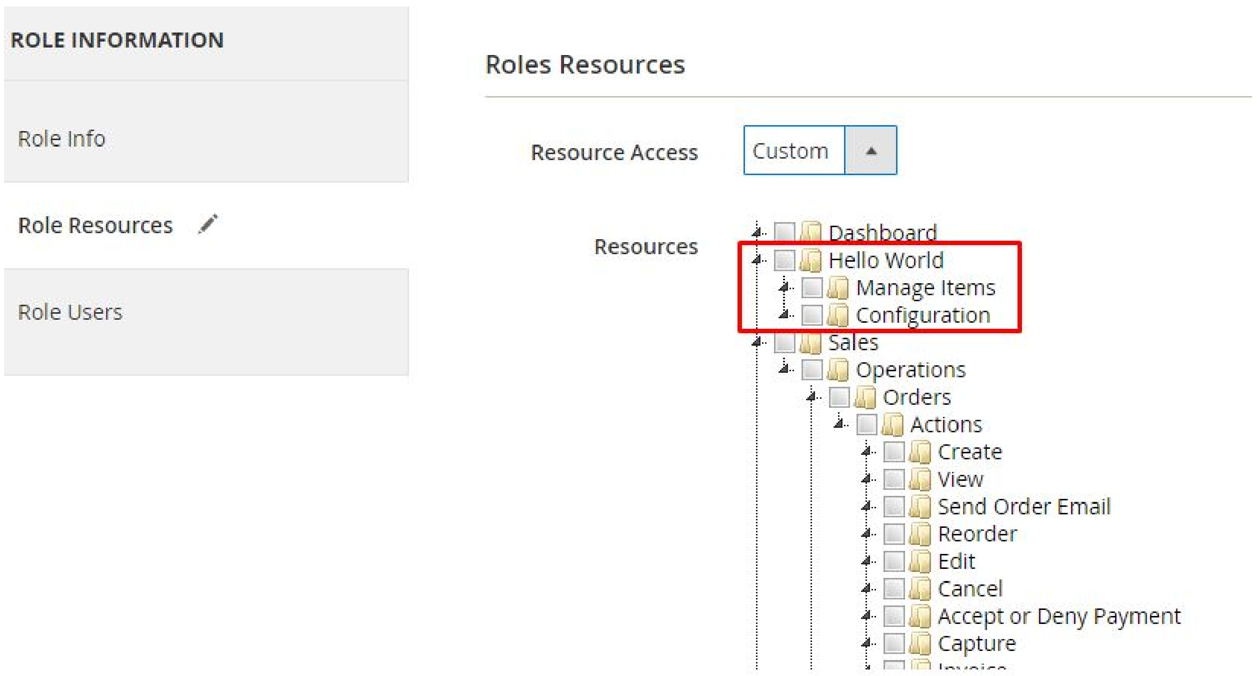
File: `app/code/Mageplaza/HelloWorld/etc/acl.xml`

```
<?xml version="1.0"?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:magento:framework:Acl/etc/acl.xsd">
    <acl>
        <resources>
            <resource id="Magento_Backend::admin">
                <resource id="Mageplaza_HelloWorld::hello" title="Hello World"
sortOrder="10" >
                    <resource id="Mageplaza_HelloWorld::hello_manage_items"
title="Manage Items" sortOrder="0" />
                    <resource id="Mageplaza_HelloWorld::hello_configuration"
title="Configuration" sortOrder="100" />
                </resource>
            </resource>
        </resources>
    </acl>
</config>
```

Our resource will be placed as a child of `Magento_Backend::admin`. Each resource will have an `Id, title and sortOrder` attribute:

- Id attribute is the identity of this resource. You can use this when defining resource in Admin menu, configuration and limit access to your module controller. This is a unique string and should be in this format: Vendor_ModuleName::resource_name.
- Title attribute is the label of this resource when showing in resource tree.
- sortOrder attribute defines the position of this resource in the tree.

After this done, please refresh the cache and see the result on resource tree.



Checking ACL rule

There are some places where we put the ACL resource to make it limit the access:

Admin menu: Put the ACL resource to hide the menu if it's not allowed by the store owner.

File: `app/code/Mageplaza/HelloWorld/etc/adminhtml/menu.xml`

```
<add id="Mageplaza_HelloWorld::hello" title="Hello World"
module="Mageplaza_HelloWorld" sortOrder="50" resource="Mageplaza_HelloWorld::hello"/>
```

System configuration: Put the ACL resource to limit access to this section page.

File: `app/code/Mageplaza/HelloWorld/etc/adminhtml/system.xml`

```
<section id="hello" translate="label" sortOrder="130" showInDefault="1"
showInWebsite="1" showInStore="1">
            ….
            <resource>Mageplaza_HelloWorld::hello_configuration</resource>
            ….
</section>
```

Admin controllers: Magento provides an abstract type `Magento\Framework\AuthorizationInterface` which you can use to validate the currently logged in user against a specific ACL. You can call that object by use the variable: `$this->_authorization`. In the controller, you have to write a protected function to check the resource:

File: `vendor/magento/module-customer/Controller/Adminhtml/Index.php`

```
protected function _isAllowed()
{
 return $this->_authorization->isAllowed('Magento_Customer::manage');
}
```

TOPIC 8: MAGENTO 2 EVENTS

This article will talk about Event in Magento 2. As you know, Magento 2 is using the event-driven architecture which will help too much to extend the Magento functionality. We can understand this event as a kind of flag that rises when a specific situation happens. We will use an example module Mageplaza_Example to exercise this lesson.

Dispatch event

In Magento 2 Events, we can use the class Magento\Framework\Event\Manager to dispatch event. For example, we create a controller action in Mageplaza_Example to show the word "Hello World" on the screen:

File: `app/code/Mageplaza/Example/Controller/Hello/World.php`

```
<?php
namespace Mageplaza\Example\Controller\Hello;

class World extends \Magento\Framework\App\Action\Action
{
  public function execute()
  {
      echo 'Hello World';
      exit;
  }
}
```

Now we want to dispatch a Magento 2 event list which allows other modules to change the word displayed. We will change the controller like this:

File: `app/code/Mageplaza/Example/Controller/Hello/World.php`

```
<?php
namespace Mageplaza\Example\Controller\Hello;

class World extends \Magento\Framework\App\Action\Action
{
  public function execute()
  {
      $textDisplay = new \Magento\Framework\DataObject(array('text' => 'Mageplaza'));
```

```
     $this->_eventManager->dispatch('example_hello_world_display_text', ['text' =>
$textDisplay]);
     echo $textDisplay->getText();
     exit;
  }
}
```

The dispatch method will receive 2 arguments: an unique event name and an array data. In this example, we add the data object to the event and call it back to display the text.

Catch and handle event

Event area

Magento use area definition to manage the store. We will have a frontend area and admin area. With the configuration file, they can be put in 3 places:

- Under `etc/` folder is the configuration which can be used in both admin and frontend.
- Under `etc/frontend` folder will be used for frontend area.
- Under `etc/adminhtml` folder will be used for admin area.

The same with the event configuration file. You can create events configuration file for each area like this:

- Admin area: `app/code/Mageplaza/Example/etc/adminhtml/events.xml`
- Frontend area: `app/code/Mageplaza/Example/etc/frontend/events.xml`
- Global area: `app/code/Mageplaza/Example/etc/events.xml`

Create events.xml

In this example, we only catch the event to show the word `Mageplaza` on the frontend so we should create an events.xml file in `etc/frontend` folder.

File: `app/code/Mageplaza/Example/etc/frontend/events.xml`

```
<?xml version="1.0"?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:magento:framework:Event/etc/events.xsd">
   <event name="example_hello_world_display_text">
       <observer name="hello_world_display"
instance="Mageplaza\Example\Observer\ChangeDisplayText" />
```

```
   </event>
</config>
```

In this file, under config element, we define an event element with the name is the event name which was dispatch above. The class which will execute this event will be defined in the observer element by instance attribute. The name of the observer is used to identify this with other observers of this event.

With this `events.xml` file, Magento will execute class `Mageplaza\Example\Observer\ChangeDisplayText` whenever the dispatch method of this event was called on frontend area. Please note that we place `events.xml` in the frontend area, so if you dispatch that event in the admin area (like admin controller), it will not run.

Observer

Now we will create a class to execute above event.

File: `app/code/Mageplaza/Example/Observer/ChangeDisplayText.php`

```
<?php
namespace Mageplaza\Example\Observer;

class ChangeDisplayText implements \Magento\Framework\Event\ObserverInterface
{
  public function execute(\Magento\Framework\Event\Observer $observer)
  {
     $displayText = $observer->getData('text');
     $displayText->setText('Execute event successfully.');

     return $this;
  }
}
```

This class will implement the `ObserverInterface` and declare the execute method. You can see this simple method to know how it works.

Let’s flush cache and see the result.

TOPIC 9: HOW TO CREATE SQL SETUP SCRIPT IN MAGENTO 2

In this article, we will find out how to install and upgrade sql script for the module in Magento 2. When you install or upgrade a module, you may need to change the database structure or add some new data for the current table. To do this, Magento 2 provide you some classes which you can do all of them.

- `InstallSchema` - this class will run when the module is installed to setup the database structure
- `InstallData` - this class will run when the module is installed to initial the data for the database table
- `UpgradeSchema` - this class will run when the module is upgraded to setup the database structure
- `UpgradeData` - this class will run when the module is upgraded to add/remove data from table
- `Recurring`
- `Uninstall`

All of the classes will be located in `app/code/Vendor/Module/Setup` folder. The module install/upgrade script will run when you run the following command line:

```
php bin/magento setup:upgrade
```

In this article, we will use the sample module Mageplaza_HelloWorld to create some demo table and data.

InstallSchema / InstallData

The InstallSchema and InstallData classes will be run during the module install.

The InstallSchema setup script in magento 2 will be use to change the database schema (create or change database table). This's the setup script to create the mageplaza_blog table:

File: `app/code/Mageplaza/HelloWorld/Setup/InstallSchema.php`

```
<?php
namespace Mageplaza\HelloWorld\Setup;
class InstallSchema implements \Magento\Framework\Setup\InstallSchemaInterface
{
	public function install(
		\Magento\Framework\Setup\SchemaSetupInterface $setup,
		\Magento\Framework\Setup\ModuleContextInterface $context
	){
```

```
                $installer = $setup;
                $installer->startSetup();

                $table = $installer->getConnection()
                        ->newTable($installer->getTable('mageplaza_blog'))
                        ->addColumn(
                                'blog_id',
                                \Magento\Framework\Db\Ddl\Table::TYPE_INTEGER,
                                null,
                                ['identity' => true, 'nullable' => false, 'primary' =>
true, 'unsigned' => true],
                                'Blog Id'
                        )->addColumn(
                                'title',
                                \Magento\Framework\Db\Ddl\Table::TYPE_TEXT,
                                255,
                                ['nullable' => false],
                                'Blog Title'
                        )->addColumn(
                                'content',
                                \Magento\Framework\Db\Ddl\Table::TYPE_TEXT,
                                '2M',
                                [],
                                'Blog Content'
                        )->addColumn(
                                'creation_time',
                                \Magento\Framework\Db\Ddl\Table::TYPE_TIMESTAMP,
                                null,
                                ['nullable' => false, 'default' =>
\Magento\Framework\Db\Ddl\Table::TIMESTAMP_INIT],
                                'Blog Creation Time'
                        )->setComment(
                                'Mageplaza Blog Table'
                        );
                $installer->getConnection()->createTable($table);
                $installer->endSetup();
        }
}
```

Looking into this file we will see:

The class must extend `\Magento\Framework\Setup\InstallSchemaInterface`

The class must have `install()` method with 2 arguments `SchemaSetupInterface` and `ModuleContextInterface`. The `SchemaSetupInterface` is the setup object which provide many function to interact with database server. The `ModuleContextInterface` has only 1 method `getVersion()` which will return the current version of your module.

In the example above, we create a table named `mageplaza_blog` with 4 columns: `blog_id, title, content and creation_time`.

`InstallData` will be run after the `InstallSchema` class to add data to the database table.

File: `app/code/Mageplaza/HelloWorld/Setup/InstallData.php`

```
<?php
namespace Mageplaza\HelloWorld\Setup;

use Magento\Framework\Setup\InstallDataInterface;
use Magento\Framework\Setup\ModuleContextInterface;
use Magento\Framework\Setup\ModuleDataSetupInterface;

class InstallData implements InstallDataInterface
{
    protected $_blogFactory;

    public function __construct(\Mageplaza\HelloWorld\Model\BlogFactory $blogFactory)
    {
        $this->_blogFactory = $blogFactory;
    }

    public function install(ModuleDataSetupInterface $setup, ModuleContextInterface
$context)
    {
        $data = [
            'title' => "Sample title 1",
            'content' => "Sample content 1"
        ];
        $blog = $this->_blogFactory->create();
        $blog->addData($data)->save();
    }
}
```

This class will have the same concept as `InstallSchema`.

UpgradeSchema/UpgradeData

The both of these files will run when the module is installed or upgraded. These classes are different with the Install classes because they will run every time the module upgrade. So we will need to check the version and separate the script by each version.

Upgrade Schema:

File: `app/code/Mageplaza/HelloWorld/Setup/UpgradeSchema.php`

```
<?php
namespace Mageplaza\HelloWorld\Setup;

use Magento\Framework\Setup\UpgradeSchemaInterface;
use Magento\Framework\Setup\SchemaSetupInterface;
use Magento\Framework\Setup\ModuleContextInterface;

class UpgradeSchema implements UpgradeSchemaInterface
{
	public function upgrade( SchemaSetupInterface $setup, ModuleContextInterface
$context ) {
		$installer = $setup;

		$installer->startSetup();
		if(version_compare($context->getVersion(), '1.0.1', '<')) {
			$installer->getConnection()->dropColumn(
				$installer->getTable( 'mageplaza_blog' ),
				'creation_time'
			);
		}

		$installer->endSetup();
	}
}
```

In this class, we use the `upgrade()` method which will be run every time the module is upgraded. We also have to compare the version to add the script for each version.

Upgrade Data:

This will same with the `UpgradeSchema` class

File: `app/code/Mageplaza/HelloWorld/Setup/UpgradeData.php`

```
<?php
namespace Mageplaza\HelloWorld\Setup;

use Magento\Framework\Setup\UpgradeDataInterface;
use Magento\Framework\Setup\ModuleDataSetupInterface;
use Magento\Framework\Setup\ModuleContextInterface;

class UpgradeData implements UpgradeDataInterface
{
	protected $_blogFactory;

	public function __construct(\Mageplaza\HelloWorld\Model\BlogFactory
$blogFactory)
	{
		$this->_blogFactory = $blogFactory;
	}

	public function upgrade( ModuleDataSetupInterface $setup,
ModuleContextInterface $context ) {
		if ( version_compare($context->getVersion(), '1.0.1', '<' )) {
			$data = [
				'title' => "Sample title 2",
				'content' => "Sample content 2"
			];
			$blog = $this->_blogFactory->create();
			$blog->addData($data)->save();
		}
	}
}
```

Recurring

The recurring script is a script which will be ran after the module setup script every time the command line `php bin/magento setup:upgrade` run.

This script will be defined as same as `InstallSchema` class and only different in the name of the class. The example for this class you can see in `vendor/magento/module-indexer/Setup/Recurring.php`

Uninstall

Magento 2 provide us the uninstall module feature which will remove all of the tables, data like it hadn't installed yet. This's the example for this class:

File: `app/code/Mageplaza/Example/Setup/Uninstall.php`

```
<?php
namespace Mageplaza\Example\Setup;

use Magento\Framework\Setup\UninstallInterface;
use Magento\Framework\Setup\SchemaSetupInterface;
use Magento\Framework\Setup\ModuleContextInterface;

class Uninstall implements UninstallInterface
{
	public function uninstall(SchemaSetupInterface $setup, ModuleContextInterface
$context)
	{
		$installer = $setup;
		$installer->startSetup();


$installer->getConnection()->dropTable($installer->getTable('mageplaza_blog'));

		$installer->endSetup();
	}
}
```

TOPIC 10: MAGENTO 2 ROUTING

In this article, we will talk about an important part in Magento 2 Routing. The Route will define the name for a module which we can use in the url to find the module and execute the controller action.

Magento 2 request flow

In Magento 2, the request url will be like this:

```
http://example.com/index.php/front_name/controller/action
```

In that url, you will see the front_name which will be used to find the module. The router define this name for each module by defining in router.xml which we will see more detail bellow.

When you make a request in Magento 2, it will follow this flow to find the `controller/action`: `index.php → HTTP app → FrontController → Routing → Controller processing → etc`

The `FrontController` will be call in Http class to routing the request which will find the `controller/action` match.

File: `vendor/magento/framework/App/FrontController.php`

```
public function dispatch(RequestInterface $request)
{
   \Magento\Framework\Profiler::start('routers_match');
   $routingCycleCounter = 0;
   $result = null;
   while (!$request->isDispatched() && $routingCycleCounter++ < 100) {
       /** @var \Magento\Framework\App\RouterInterface $router */
       foreach ($this->_routerList as $router) {
           try {
               $actionInstance = $router->match($request);
               if ($actionInstance) {
                   $request->setDispatched(true);
                   $this->response->setNoCacheHeaders();
                   if ($actionInstance instanceof
\Magento\Framework\App\Action\AbstractAction) {
```

```
                        $result = $actionInstance->dispatch($request);
                    } else {
                        $result = $actionInstance->execute();
                    }
                    break;
                }
            } catch (\Magento\Framework\Exception\NotFoundException $e) {
                $request->initForward();
                $request->setActionName('noroute');
                $request->setDispatched(false);
                break;
            }
        }
    }
    \Magento\Framework\Profiler::stop('routers_match');
    if ($routingCycleCounter > 100) {
        throw new \LogicException('Front controller reached 100 router match
iterations');
    }
    return $result;
}
```

As you can see in this `dispatch()` method, the router list will be a loop to find the match one with this request. If it finds out the controller action for this request, that action will be called and executed.

Create custom route on frontend/admin

In this part, we will use a simple module Mageplaza_HelloWorld. Please follow the previous article to know how to create and register a Module in Magento 2.

We will find how to create a frontend route, admin route and how to use route to rewrite controller.

Frontend route

To register a frontend route, we must create a 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">
   <!--Use router 'standard' for frontend route-->
   <router id="standard">
       <!--Define a custom route with id and frontName-->
       <route id="example" frontName="example">
           <!--The module which this route match to-->
           <module name="Mageplaza_HelloWorld" />
       </route>
   </router>
</config>
```

Please look into the code, you will see it's very simple to register a route. You must use the standard router for the frontend. This route will have a child which define the module for it and 2 attributes:

- The id attribute is a unique string which will identify this route. You will use this string to declare the layout handle for the action of this module
- The frontName attribute is also a unique string which will be shown on the url request. For example, if you declare a route like this:

```
<route id="exampleid" frontName="examplefront">
```

The url to this module should be:

```
http://example.com/index.php/examplefront/controller/action
```

And the layout handle for this action is: `exampleid_controller_action.xml` So with this example path, you must create the action class in this folder:`{module_path}/Controller/Controller/Action.php`

Admin route

This route will be same as the frontend route but you must declare it in adminhtml folder with router id is `admin`.

File: `app/code/Mageplaza/HelloWorld/etc/adminhtml/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">
   <!--Use router 'standard' for frontend route-->
   <router id="admin">
```

```
        <!--Define a custom route with id and frontName-->
        <route id="example" frontName="example">
            <!--The module which this route match to-->
            <module name="Mageplaza_HelloWorld"/>
        </route>
    </router>
</config>
```

The url of the admin page is the same structure with frontend page, but the `admin_area` name will be added before `route_frontName` to recognize this is an admin router. For example, the url of admin cms page:

```
http://example.com/index.php/admin/example/blog/index
```

The controller action for admin page will be added inside of the folder `Controller/Adminhtml`. For example for above url:

```
{module_path}/Controller/Adminhtml/Blog/Index.php
```

Use route to rewrite controller

In this path, we will see how to rewrite a controller with router. As above path, you can see each route will have an id attribute to identify. So what happen if we define 2 route with the same id attribute?

The answer is that the controller action will be found in both of that modules. Magento system provides the attribute before/after to config the module sort order which defines what module controller will be found first. This's the logic for the controller rewrite.

For example, if we want to rewrite the controller customer/account/login, we will define more route in the route.xml like this:

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">
    <!--Use router 'standard' for frontend route-->
    <router id="standard">
        <!--Define a custom route with id and frontName-->
        <route id="example" frontName="example">
            <!--The module which this route match to-->
            <module name="Mageplaza_HelloWorld" />
        </route>
        <route id="customer">
            <module name="Mageplaza_HelloWorld" before="Magento_Customer" />
        </route>
    </router>
</config>
```

And the controller file: `app/code/Mageplaza/HelloWorld/Controller/Account/Login.php`

So the frontController will find the Login action in our module first, if it's found, it will run and the Login action of Magento_Customer will not be run. We are successful rewrite a controller.

You can also use this to have a second module with the same router as another module. For example, with above declare, you can use route 'customer' for your controller action. If you have controller 'Blog' and action 'Index.php' you can use this url:

```
http://example.com/customer/blog/index
```

TOPIC 11: HOW TO CREATE ADMIN GRID IN MAGENTO 2

In this article, we will find how to create an Admin Grid in Magento 2 backend. As you know, Magento 2 Grid is a kind of table which listing the items in your database table and provide you some features like: sort, filter, delete, update item, etc. The example for this is the grid of products, grid of customer.

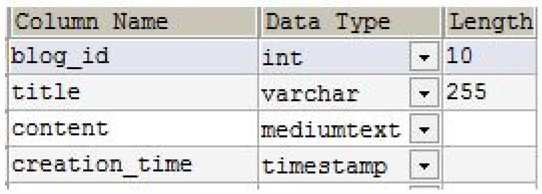
Magento 2 provide two ways to create Admin Grid: using layout and using component. We will find out the detail for both of them. Before we continue please follow this articles to create a simple module with admin menu, the router which we will use to learn about grid. In this article, I will use the sample module `Mageplaza_Example` with some demo data:

To Create Admin Grid

- Step 1: Create database schema
- Step 2: Create admin menu
- Step 3: Create Controller
- Step 4: Declare resource
- Step 5: Create Admin Grid using Component
- Step 6: Create Admin Grid using Layout

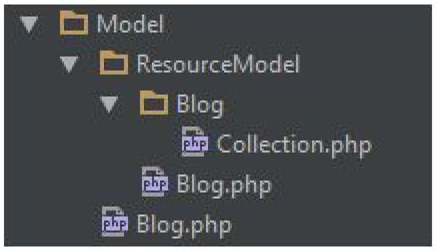
Step 1: Create database schema

Database: We will use a simple database



Column Name	Data Type	Length
blog_id	int	10
title	varchar	255
content	mediumtext	
creation_time	timestamp	

Create Resource Model and Model Collection - `Model/Resource` `Model/Collection` - like this


```
Model
    ResourceModel
        Blog
            Collection.php
        Blog.php
    Blog.php
```

Step 2: Create admin menu

Admin menu/Route: we will use the route `example` for our admin page and the menu link to:

```
example/blog/index
```

Step 3: Create Controller

Create controller file: please read the Create Controller article for the detail.

Create controller file called index.php

```
app/code/Mageplaza/Example/Controller/Adminhtml/Blog/Index.php
```

With the following content:

```
<?php
namespace Mageplaza\Example\Controller\Adminhtml\Blog;

class Index extends \Magento\Backend\App\Action
{
        protected $resultPageFactory = false;
        public function __construct(
                \Magento\Backend\App\Action\Context $context,
                \Magento\Framework\View\Result\PageFactory $resultPageFactory
        ) {
                parent::__construct($context);
                $this->resultPageFactory = $resultPageFactory;
        }

        public function execute()
        {
                //Call page factory to render layout and page content
                $resultPage = $this->resultPageFactory->create();
```

```
            //Set the menu which will be active for this page
            $resultPage->setActiveMenu('Mageplaza_Example::blog_manage');

            //Set the header title of grid
            $resultPage->getConfig()->getTitle()->prepend(__('Manage Blogs'));

            //Add bread crumb
            $resultPage->addBreadcrumb(__('Mageplaza'), __('Mageplaza'));
            $resultPage->addBreadcrumb(__('Hello World'), __('Manage Blogs'));

            return $resultPage;
        }

        /*
         * Check permission via ACL resource
         */
        protected function _isAllowed()
        {
            return
$this->_authorization->isAllowed('Mageplaza_Example::blog_manage');
        }
}
```

Step 4: Declare resource

Declare resource in dependency injection file Now we will create `di.xml` file which will connect to the Model to get the data for our grid.

File: `app/code/Mageplaza/Example/etc/di.xml`

```
<?xml version="1.0"?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:magento:framework:ObjectManager/etc/config.xsd">
    <virtualType name="Mageplaza\Example\Model\ResourceModel\Blog\Collection"
type="Magento\Framework\View\Element\UiComponent\DataProvider\SearchResult">
        <arguments>
            <argument name="mainTable" xsi:type="string">mageplaza_blog</argument>
            <argument name="resourceModel"
xsi:type="string">Mageplaza\Example\Model\ResourceModel\Blog</argument>
        </arguments>
    </virtualType>
    <type
name="Magento\Framework\View\Element\UiComponent\DataProvider\CollectionFactory">
        <arguments>
            <argument name="collections" xsi:type="array">
                <item name="example_blog_grid_data_source"
xsi:type="string">Mageplaza\Example\Model\ResourceModel\Blog\Collection</item>
            </argument>
        </arguments>
    </type>
</config>
```

This file will declare the blog collection class, table and resourceModel for the table. This source will be called in the layout file to get data for grid.

There are 2 ways to create admin grid, in this post's scope, we will talk about both of them.

Step 5: Create Admin Grid using Component

Step 5.1: Create layout file

For the action `example/blog/index`, we will create a layout file named `example_blog_index.xml`

File: `app/code/Mageplaza/Example/view/adminhtml/layout/example_blog_index.xml`

```
<?xml version="1.0"?>
<page xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:magento:framework:View/Layout/etc/page_configuratio
n.xsd">
    <update handle="styles"/>
    <body>
        <referenceContainer name="content">
            <uiComponent name="mageplaza_blog_grid"/>
        </referenceContainer>
    </body>
</page>
```

In this layout file, we declare an *uiComponent* for the content of this page.

Step 5.2: Create component layout file

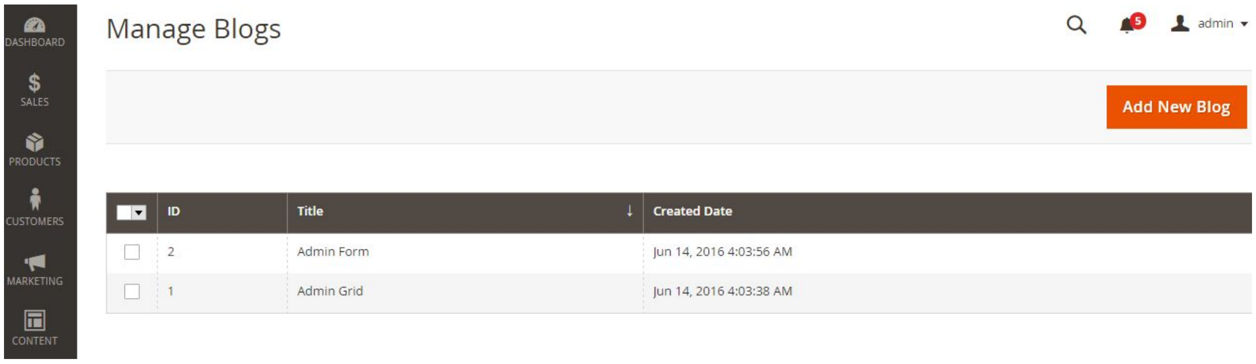
As declaration in layout file, we will create a component file `mageplaza_blog_grid.xml`

File:
app/code/Mageplaza/Example/view/adminhtml/ui_component/mageplaza_blog_grid.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<listing xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:magento:module:Magento_Ui:etc/ui_configuration.xsd"
>
    <!--Declare data source, columns list, button...-->
    <argument name="data" xsi:type="array">
        <item name="js_config" xsi:type="array">
            <item name="provider"
xsi:type="string">mageplaza_blog_grid.example_blog_grid_data_source</item>
            <item name="deps"
xsi:type="string">mageplaza_blog_grid.example_blog_grid_data_source</item>
            <!--Declare the data source name which will be defined below-->
        </item>
        <item name="spinner" xsi:type="string">example_blog_columns</item>
        <!--Declare the listing of columns which will be defined below-->
        <item name="buttons" xsi:type="array">
            <item name="add" xsi:type="array">
                <item name="name" xsi:type="string">add</item>
                <item name="label" xsi:type="string" translate="true">Add New
Blog</item>
                <item name="class" xsi:type="string">primary</item>
                <item name="url" xsi:type="string">*/*/new</item>
            </item>
            <!--The button on the top of the Grid-->
        </item>
    </argument>
    <dataSource name="example_blog_grid_data_source">
        <!--The data source-->
        <argument name="dataProvider" xsi:type="configurableObject">
            <argument name="class"
xsi:type="string">Magento\Framework\View\Element\UiComponent\DataProvider\DataProvider
</argument>
            <argument name="name"
xsi:type="string">example_blog_grid_data_source</argument>
            <argument name="primaryFieldName" xsi:type="string">blog_id</argument>
            <argument name="requestFieldName" xsi:type="string">id</argument>
            <argument name="data" xsi:type="array">
                <item name="config" xsi:type="array">
                    <item name="component"
xsi:type="string">Magento_Ui/js/grid/provider</item>
                    <item name="update_url" xsi:type="url" path="mui/index/render"/>
                    <item name="storageConfig" xsi:type="array">
                        <item name="indexField" xsi:type="string">blog_id</item>
                    </item>
                </item>
            </argument>
        </argument>
    </dataSource>
    <columns name="example_blog_columns">
        <!--The list of columns-->
        <selectionsColumn name="ids">
            <argument name="data" xsi:type="array">
                <item name="config" xsi:type="array">
                    <item name="indexField" xsi:type="string">blog_id</item>
                </item>
            </argument>
        </selectionsColumn>
        <column name="blog_id">
            <argument name="data" xsi:type="array">
                <item name="config" xsi:type="array">
                    <item name="filter" xsi:type="string">text</item>
                    <item name="sorting" xsi:type="string">asc</item>
                    <item name="label" xsi:type="string" translate="true">ID</item>
                </item>
            </argument>
        </column>
        <column name="title">
            <argument name="data" xsi:type="array">
                <item name="config" xsi:type="array">
                    <item name="filter" xsi:type="string">text</item>
                    <item name="sorting" xsi:type="string">asc</item>
                    <item name="label" xsi:type="string" translate="true">Title</item>
                </item>
            </argument>
        </column>
        <column name="creation_time"
class="Magento\Ui\Component\Listing\Columns\Date">
            <argument name="data" xsi:type="array">
```

```
                <item name="config" xsi:type="array">
                    <item name="filter" xsi:type="string">dateRange</item>
                    <item name="component"
xsi:type="string">Magento_Ui/js/grid/columns/date</item>
                    <item name="dataType" xsi:type="string">date</item>
                    <item name="label" xsi:type="string" translate="true">Created
Date</item>
                </item>
            </argument>
        </column>
    </columns>
</listing>
```

With this code, you will know how to declare Grid layout (button, columns), call data source. Please refresh the cache, and access to this grid page, the admin grid will show up like this:



ID	Title	Created Date
2	Admin Form	Jun 14, 2016 4:03:56 AM
1	Admin Grid	Jun 14, 2016 4:03:38 AM

Step 5.3: Create a container

As I said on the top of this page, the Magento 2 Grid will support some actions to interact with grid like: sort, filter, action delete/update etc. The sort feature is a default action for the grid. You can click on the column header to sorting the items. We will find out how to built the other features for our grid.

Prepare for this, we will create a container element under the parent listing in the component layout file:

File:
`app/code/Mageplaza/Example/view/adminhtml/ui_component/mageplaza_blog_grid.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<listing xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:magento:module:Magento_Ui:etc/ui_configuration.xsd"
```

```
>

    <!-- ... other block of code -->

    <container name="listing_top">
        <argument name="data" xsi:type="array">
            <item name="config" xsi:type="array">
                <item name="template" xsi:type="string">ui/grid/toolbar</item>
            </item>
        </argument>
    </container>
</listing>
```

Step 5.4: Create a Bookmark

This argument is used to define the template `Magento/Ui/view/base/web/templates/grid/toolbar.html` which will be loaded to define the knockout js for handling all ajax update action in this grid. We will define above features inside of this container. You can place this container element before or after the columns element to define the position of the toolbar (above or below the columns). Let's see the detail for each action: `Bookmark`

```
<bookmark name="bookmarks">
    <argument name="data" xsi:type="array">
         <item name="config" xsi:type="array">
             <item name="storageConfig" xsi:type="array">
                 <item name="namespace" xsi:type="string">mageplaza_blog_grid</item>
             </item>
          </item>
     </argument>
 </bookmark>
```

This will add the bookmark feature which allows admin setup difference `state` of the grid. Each `state` may have a difference columns list. So with each admin user, they can choose to show the information which are relevant to him.

Step 5.5: Column controls

This node will add a columns list box which allows the admin user can choose which columns can be shown up on the grid. After changing this list, admin can save that state as a bookmark which allows accessing this state quickly.

```
<component name="columns_controls">
    <argument name="data" xsi:type="array">
        <item name="config" xsi:type="array">
            <item name="columnsData" xsi:type="array">
                <item name="provider"
xsi:type="string">mageplaza_blog_grid.mageplaza_blog_grid.example_blog_columns</item>
            </item>
            <item name="component"
xsi:type="string">Magento_Ui/js/grid/controls/columns</item>
            <item name="displayArea" xsi:type="string">dataGridActions</item>
        </item>
    </argument>
</component>
```

Step 5.6: Full text search

This node will add a search box at the top of Grid. You can use this to search all the data in the table.

```
<filterSearch name="fulltext">
    <argument name="data" xsi:type="array">
        <item name="config" xsi:type="array">
            <item name="provider"
xsi:type="string">mageplaza_blog_grid.example_blog_grid_data_source</item>
            <item name="chipsProvider"
xsi:type="string">mageplaza_blog_grid.mageplaza_blog_grid.listing_top.listing_filters_
chips</item>
            <item name="storageConfig" xsi:type="array">
                <item name="provider"
xsi:type="string">mageplaza_blog_grid.mageplaza_blog_grid.listing_top.bookmarks</item>
                <item name="namespace" xsi:type="string">current.search</item>
            </item>
        </item>
    </argument>
</filterSearch>
```

Step 5.7: Filter

This node define the filter box for each column. You can see this by click to the Filter button at the top of the grid.

```
<filters name="listing_filters">
    <argument name="data" xsi:type="array">
        <item name="config" xsi:type="array">
            <item name="columnsProvider"
xsi:type="string">mageplaza_blog_grid.mageplaza_blog_grid.example_blog_columns</item>
            <item name="storageConfig" xsi:type="array">
                <item name="provider"
xsi:type="string">mageplaza_blog_grid.mageplaza_blog_grid.listing_top.bookmarks</item>
                <item name="namespace" xsi:type="string">current.filters</item>
            </item>
            <item name="childDefaults" xsi:type="array">
                <item name="provider"
xsi:type="string">mageplaza_blog_grid.mageplaza_blog_grid.listing_top.listing_filters<
/item>
                <item name="imports" xsi:type="array">
                    <item name="visible"
xsi:type="string">mageplaza_blog_grid.mageplaza_blog_grid.example_blog_columns.${
$.index }:visible</item>
                </item>
            </item>
        </item>
    </argument>
</filters>
```

Step 5.8: Mass actions

This node will add the mass action select to the grid. The Admin can use this action to take some action quickly on multiple items.

```
<massaction name="listing_massaction">
    <argument name="data" xsi:type="array">
        <item name="config" xsi:type="array">
            <item name="selectProvider"
xsi:type="string">mageplaza_blog_grid.mageplaza_blog_grid.example_blog_columns.ids</it
em>
            <item name="component"
xsi:type="string">Magento_Ui/js/grid/tree-massactions</item>
            <item name="indexField" xsi:type="string">entity_id</item>
        </item>
    </argument>
    <action name="delete">
        <argument name="data" xsi:type="array">
            <item name="config" xsi:type="array">
                <item name="type" xsi:type="string">delete</item>
                <item name="label" xsi:type="string" translate="true">Delete</item>
                <item name="url" xsi:type="url" path="*/*/massDelete"/>
                <item name="confirm" xsi:type="array">
                    <item name="title" xsi:type="string" translate="true">Delete
items</item>
                    <item name="message" xsi:type="string" translate="true">Are you
sure to delete selected blogs?</item>
                </item>
            </item>
        </argument>
    </action>
</massaction>
```

Step 5.9: Paging

This node will add the pagination for the grid. This is useful if you have a large of data in the table.

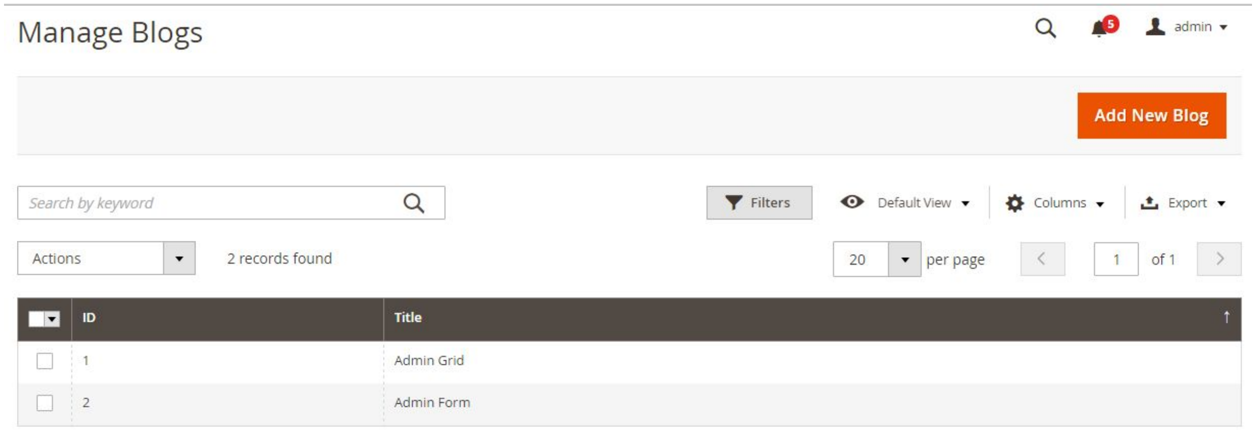
```
<paging name="listing_paging">
    <argument name="data" xsi:type="array">
        <item name="config" xsi:type="array">
            <item name="storageConfig" xsi:type="array">
                <item name="provider"
xsi:type="string">mageplaza_blog_grid.mageplaza_blog_grid.listing_top.bookmarks</item>
                <item name="namespace" xsi:type="string">current.paging</item>
            </item>
            <item name="selectProvider"
xsi:type="string">mageplaza_blog_grid.mageplaza_blog_grid.example_blog_columns.ids</it
em>
        </item>
    </argument>
</paging>
```

Step 5.10: Export

This node will add an export button which you can export the data of the grid.

```
<exportButton name="export_button">
    <argument name="data" xsi:type="array">
        <item name="config" xsi:type="array">
            <item name="selectProvider"
xsi:type="string">mageplaza_blog_grid.mageplaza_blog_grid.example_blog_columns.ids</it
em>
        </item>
    </argument>
</exportButton>
```

Try to clean the cache and go to the grid page, you will see a grid like this:



ID	Title
1	Admin Grid
2	Admin Form

Step 6: Create Admin Grid using Layout

Important! Skip this step if you have ready done Step 5

You have just found how to add a Magento 2 Grid by using Component. Now we will see how to do it by using normal layout/block file.

Step 6.1: Create block for this grid

File: app/code/Mageplaza/Example/Block/Adminhtml/Blog/Grid.php

```
<?php
namespace Mageplaza\Example\Block\Adminhtml\Blog;
class Grid extends \Magento\Backend\Block\Widget\Grid\Container
{
        protected function _construct()
        {
                $this->_blockGroup = 'Mageplaza_Example';
                $this->_controller = 'adminhtml_blog';
                $this->_headerText = __('Manage Blogs');
                $this->_addButtonLabel = __('Add New Blog');

                parent::_construct();
        }
}
```

The Grid block will extend `\Magento\Backend\Block\Widget\Grid\Container` and define some variable in the `_construct()` method. - `_blockGroup` is the name of our module with format VendorName_ModuleName - `_controller` is the path to the Grid block inside the Block folder. In this example, I put the Grid.php file inside of the `Adminhtml/Blog` folder - `_headerText` is the Grid page title - `_addButtonLabel` is the label of the Add new button.

Step 6.2: Create layout file

Now we will need a layout file to connect with Grid Block and render the grid. Let's create this file:

File: `app/code/Mageplaza/Example/view/adminhtml/layout/example_blog_index.xml`

```
<?xml version="1.0"?>
<page xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:magento:framework:View/Layout/etc/page_configuratio
n.xsd">
    <update handle="styles"/>
    <body>
        <referenceContainer name="content">
            <!--<uiComponent name="mageplaza_blog_grid"/>-->
            <block class="Mageplaza\Example\Block\Adminhtml\Blog\Grid"
name="mageplaza_blog_grid">
                <block class="Magento\Backend\Block\Widget\Grid"
name="mageplaza_blog_grid.grid" as="grid">
                    <arguments>
                        <argument name="id" xsi:type="string">blog_id</argument>
                        <argument name="dataSource"
xsi:type="object">Mageplaza\Example\Model\ResourceModel\Blog\Collection</argument>
                        <argument name="default_sort" xsi:type="string">id</argument>
                        <argument name="default_dir" xsi:type="string">ASC</argument>
                        <argument name="save_parameters_in_session"
```

```
xsi:type="string">1</argument>
                    </arguments>
                    <block class="Magento\Backend\Block\Widget\Grid\ColumnSet"
name="mageplaza_blog_grid.grid.columnSet" as="grid.columnSet">
                        <arguments>
                            <argument name="rowUrl" xsi:type="array">
                                <item name="path" xsi:type="string">*/*/edit</item>
                            </argument>
                        </arguments>
                        <block class="Magento\Backend\Block\Widget\Grid\Column"
as="id">
                            <arguments>
                                <argument name="header" xsi:type="string"
translate="true">ID</argument>
                                <argument name="index"
xsi:type="string">blog_id</argument>
                                <argument name="type"
xsi:type="string">text</argument>
                                <argument name="column_css_class"
xsi:type="string">col-id</argument>
                                <argument name="header_css_class"
xsi:type="string">col-id</argument>
                            </arguments>
                        </block>
                        <block class="Magento\Backend\Block\Widget\Grid\Column"
as="title">
                            <arguments>
                                <argument name="header" xsi:type="string"
translate="true">Title</argument>
                                <argument name="index"
xsi:type="string">title</argument>
                                <argument name="type"
xsi:type="string">text</argument>
                                <argument name="column_css_class"
xsi:type="string">col-id</argument>
                                <argument name="header_css_class"
xsi:type="string">col-id</argument>
                            </arguments>
                        </block>
                        <block class="Magento\Backend\Block\Widget\Grid\Column"
as="creation_time">
                            <arguments>
```

```
                                <argument name="header" xsi:type="string"
translate="true">Created Time</argument>
                                <argument name="index"
xsi:type="string">creation_time</argument>
                                <argument name="type"
xsi:type="string">date</argument>
                                <argument name="column_css_class"
xsi:type="string">col-id</argument>
                                <argument name="header_css_class"
xsi:type="string">col-id</argument>
                            </arguments>
                        </block>
                    </block>
                </block>
            </block>
        </referenceContainer>
    </body>
</page>
```

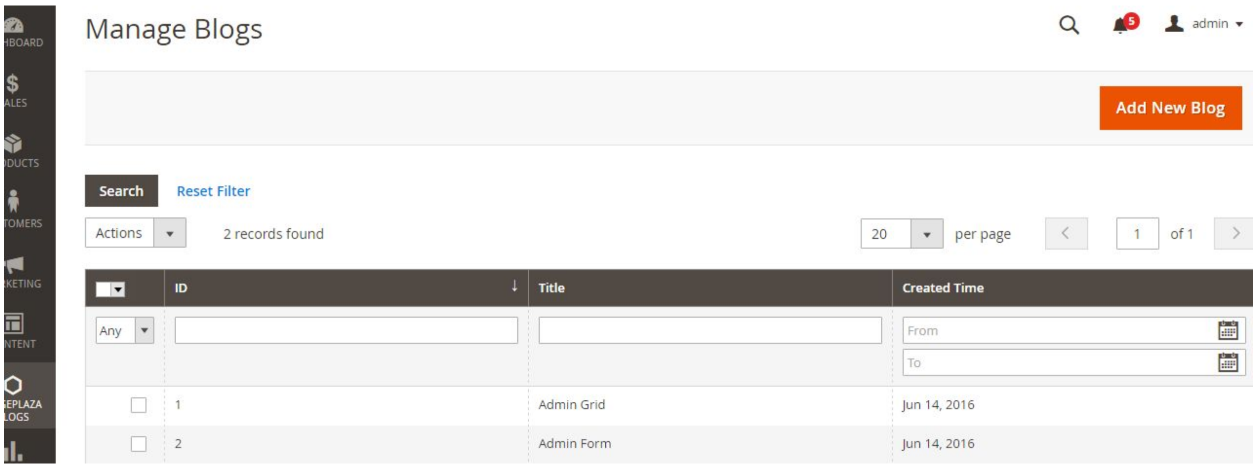
In this layout file, we will define some argument for the grid. The main argument is the dataSource. This argument will link with the dataSource which we declare in the di.xml file above to connect to the database and get data for this grid.

Step 6.4: Add Column

The Column set will define the columns which will be display in the grid. If you want to use massAction, you can add this block to the grid element:

```
<block class="Magento\Backend\Block\Widget\Grid\Massaction"
name="mageplaza.example.massaction" as="grid.massaction">
    <arguments>
        <argument name="massaction_id_field" xsi:type="string">blog_id</argument>
        <argument name="form_field_name" xsi:type="string">ids</argument>
        <argument name="use_select_all" xsi:type="string">1</argument>
        <argument name="options" xsi:type="array">
            <item name="disable" xsi:type="array">
                <item name="label" xsi:type="string" translate="true">Delete</item>
                <item name="url" xsi:type="string">*/*/massDelete</item>
            </item>
        </argument>
    </arguments>
</block>
```

After this, please refresh the cache and go to grid page to see the result. It may display like this:



	ID	Title	Created Time
☐	1	Admin Grid	Jun 14, 2016
☐	2	Admin Form	Jun 14, 2016

TOPIC 12: MAGENTO 2 INDEXING

In this article we will learn how to create an Indexer in Magento 2. Indexer is an important feature in Magento 2 Indexing. To understand how to create a Hello World module, you can read it here

We will use the example module Mageplaza_HelloWorld for this exercise. Please check our previous post to know how to create a sample module in Magento 2.

Let's start to create a custom indexer.

Create Indexer configuration file

This configuration file will define the indexer.

File `app/code/Mageplaza/HelloWorld/etc/indexer.xml`

```
<?xml version="1.0"?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:magento:framework:Indexer/etc/indexer.xsd">
   <indexer id="example_indexer" view_id="example_indexer"
class="Mageplaza\HelloWorld\Model\Indexer\Test">
```

```
        <title translate="true">HelloWorld Indexer</title>
        <description translate="true">HelloWorld of custom indexer</description>
    </indexer>
</config>
```

In this file, we declare a new indexer process with the attribute:

- The id attribute is used to identify this indexer. You can call it when you want to check status, mode or reindex this indexer by command line.
- The view_id is the id of view element which will be defined in the mview configuration file.
- The class attribute is the name to the class which we process indexer method.

The simple magento 2 indexing will have some child elements:

- The title element is used to define the Title of this when showing in indexer grid.
- The description element is used to define the Description of this when showing in indexer grid.

Create Mview configuration file

The `mview.xml` file is used to track database changes for a certain entity and running change handle (execute() method).

File: `app/code/Mageplaza/HelloWorld/etc/mview.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:magento:framework:Mview/etc/mview.xsd">
    <view id="example_indexer" class="Mageplaza\HelloWorld\Model\Indexer\Test"
group="indexer">
        <subscriptions>
            <table name="catalog_product_entity" entity_column="entity_id" />
        </subscriptions>
    </view>
</config>
```

In this file, we define a view element with an id attribute to call from indexer and a class which contain the `execute()` method. This method will run when the table in subscriptions is changed.

To declare the table, we use the table name and the column of this table which will be sent to the `execute()` method. In this example, we declare the table ‘catalog_product_entity’. So whenever one or more products is saved, the execute() method in class “Mageplaza\HelloWorld\Model\Indexer\Test” will be called.

Create Indexer class

Follow the `indexer.xml` and `mview.xml` above, we will define an Indexer class for both of them: `Mageplaza\HelloWorld\Model\Indexer\Test`

File: `app/code/Mageplaza/HelloWorld/Model/Indexer/Test.php`

```
<?php
namespace Mageplaza\HelloWorld\Model\Indexer;

class Test implements \Magento\Framework\Indexer\ActionInterface,
\Magento\Framework\Mview\ActionInterface
{
  /*
   * Used by mview, allows process indexer in the "Update on schedule" mode
   */
   public function execute($ids){}
  /*
   * Will take all of the data and reindex
   * Will run when reindex via command line
   */
   public function executeFull(){}
  /*
   * Works with a set of entity changed (may be massaction)
   */
   public function executeList(array $ids){}
  /*
   * Works in runtime for a single entity using plugins
   */
```

```
    public function executeRow($id){}
}
```

You can write the code to add data to your indexer table in the methods in Indexer class.

TOPIC 13: HOW TO ADD COMMAND LINE INTO CONSOLE CLI IN MAGENTO

In this article, we will find how to add a command line into magento 2 console CLI. Magento 2 add command line use an interface to quick change some features like enable/disable cache, setup sample data… Before we start, please take some minutes to know about the naming in Magento 2 CLI.

We will use an example module Mageplaza_Example to demo for this lesson. To add an option to Magento 2 CLI, we will follow some steps:

Step 1: Define command in di.xml

In `di.xml` file, you can use a type with name `Magento\Framework\Console\CommandList` to define the command option.

File: `app/code/Mageplaza/HelloWorld/etc/di.xml`

```
<?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\Framework\Console\CommandList">
       <arguments>
           <argument name="commands" xsi:type="array">
               <item name="exampleSayHello"
xsi:type="object">Mageplaza\HelloWorld\Console\Sayhello</item>
           </argument>
       </arguments>
   </type>
</config>
```

This config will declare a command class `Sayhello`. This class will define the command name and `execute()` method for this command.

Step 2: Create command class

As define in di.xml, we will create a command class:

File: `app/code/Mageplaza/HelloWorld/Console/Sayhello.php`

```
<?php
namespace Mageplaza\HelloWorld\Console;

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

class Sayhello extends Command
{
   protected function configure()
   {
       $this->setName('example:sayhello');
       $this->setDescription('Demo command line');
   }
   protected function execute(InputInterface $input, OutputInterface $output)
   {
       $output->writeln("Hello World");
   }
}
```

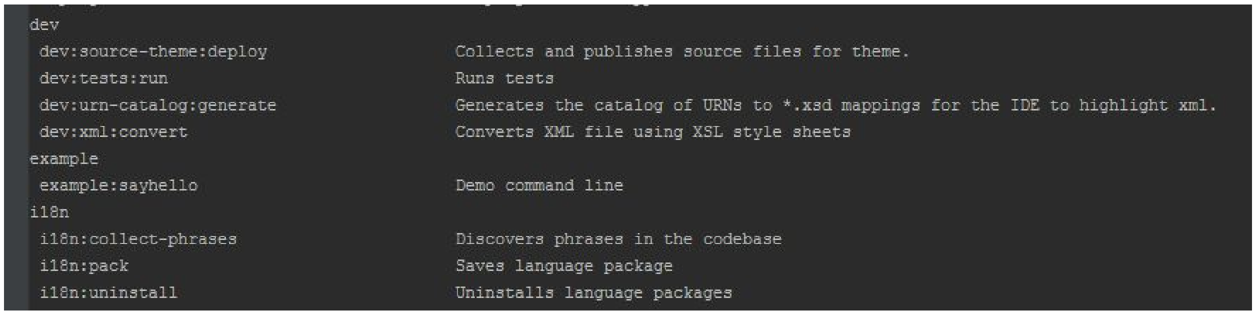
In this function, we will define 2 methods:

- `configure()` method is used to set the name and the description of the magento 2 add command line
- `execute()` method will run when we call this command line via console.

After declaring this class, please flush Magento cache and type this command:

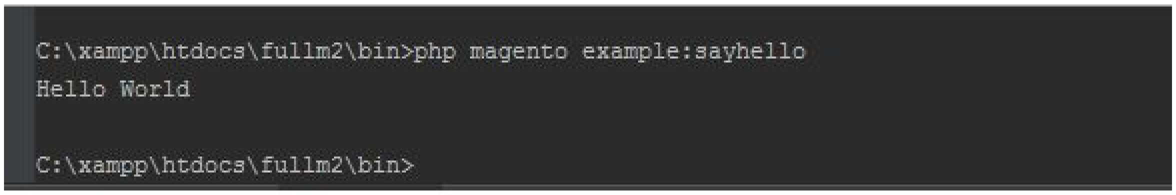
```
php magento --list
```

You will see the list of all commands. Our command will be shown here



```
dev
 dev:source-theme:deploy              Collects and publishes source files for theme.
 dev:tests:run                        Runs tests
 dev:urn-catalog:generate             Generates the catalog of URNs to *.xsd mappings for the IDE to highlight xml.
 dev:xml:convert                      Converts XML file using XSL style sheets
example
 example:sayhello                     Demo command line
i18n
 i18n:collect-phrases                 Discovers phrases in the codebase
 i18n:pack                            Saves language package
 i18n:uninstall                       Uninstalls language packages
```

Now you can run the command to see the result



```
C:\xampp\htdocs\fullm2\bin>php magento example:sayhello
Hello World

C:\xampp\htdocs\fullm2\bin>
```

TOPIC 14: MAGENTO 2 ADD CUSTOMER ATTRIBUTE PROGRAMMATICALLY

This article will guide you how to create a customer attribute in Magento 2 programmatically. Please follow our previous article to create a simple module which we will use to demo coding for this lesson and how to create the setup script classes. In this article, we will use the sample module `Mageplaza_HelloWorld` and the InstallDataclass.

Firstly, we will create the `InstallData.php` file.

File: `app/code/Mageplaza/Example/Setup/InstallData.php`

```
<?php
namespace Mageplaza\Example\Setup;

use Magento\Eav\Setup\EavSetup;
use Magento\Eav\Setup\EavSetupFactory;
use Magento\Framework\Setup\InstallDataInterface;
use Magento\Framework\Setup\ModuleContextInterface;
use Magento\Framework\Setup\ModuleDataSetupInterface;

class InstallData implements InstallDataInterface
{
        private $eavSetupFactory;

        public function __construct(EavSetupFactory $eavSetupFactory)
        {
                $this->eavSetupFactory = $eavSetupFactory;
        }

}
```

In this class, we define the EAV setup model which will be used to interact with Magento 2 attribute.

After that, we have to define the `install()` method and create eav setup model:

```
public   function   install(ModuleDataSetupInterface   $setup,   ModuleContextInterface
$context)
        {
            $eavSetup = $this->eavSetupFactory->create(['setup' => $setup]);
        }
        Next, we will use eavSetup object to add attribute:
public    function    install(ModuleDataSetupInterface    $setup,    ModuleContextInterface
$context)
        {
            $eavSetup = $this->eavSetupFactory->create(['setup' => $setup]);
            $eavSetup->addAttribute(
                \Magento\Customer\Model\Customer::ENTITY,
                'sample_attribute',
                [
                    'type' => 'int',
                    'label' => 'Sample Attribute',
                    'input' => 'select',
                    'source'                                                   =>
'Magento\Eav\Model\Entity\Attribute\Source\Boolean',
                    'required' => true,
                    'default' => '0',
                    'sort_order' => 100,
                    'system' => false,
                    'position' => 100
                ]
            );
}
```

Finally, we need to set the forms in which the attributes will be used. In this step, we need define the eavConfig object which allow us to call the attribute back and set the data for it. And the full code to create customer attribute is:

File: `app/code/Mageplaza/Example/Setup/InstallData.php`

```
<?php
namespace Mageplaza\Example\Setup;

use Magento\Eav\Setup\EavSetup;
use Magento\Eav\Setup\EavSetupFactory;
use Magento\Framework\Setup\InstallDataInterface;
use Magento\Framework\Setup\ModuleContextInterface;
use Magento\Framework\Setup\ModuleDataSetupInterface;
use Magento\Eav\Model\Config;
```

```
class InstallData implements InstallDataInterface
{
	private $eavSetupFactory;

	public function __construct(EavSetupFactory $eavSetupFactory, Config
$eavConfig)
	{
		$this->eavSetupFactory = $eavSetupFactory;
		$this->eavConfig = $eavConfig;
	}

	public function install(ModuleDataSetupInterface $setup, ModuleContextInterface
$context)
	{
		$eavSetup = $this->eavSetupFactory->create(['setup' => $setup]);
		$eavSetup->addAttribute(
			\Magento\Customer\Model\Customer::ENTITY,
			'sample_attribute',
			[
				'type' => 'int',
				'label' => 'Sample Attribute',
				'input' => 'select',
				'source' =>
'Magento\Eav\Model\Entity\Attribute\Source\Boolean',
				'required' => true,
				'default' => '0',
				'sort_order' => 100,
				'system' => false,
				'position' => 100
			]
		);
		$sampleAttribute =
$this->eavConfig->getAttribute(\Magento\Customer\Model\Customer::ENTITY,
'sample_attribute');
		$sampleAttribute->setData(
			'used_in_forms',
			['adminhtml_customer_address', 'customer_address_edit',
'customer_register_address']
		);
		$sampleAttribute->save();
```

```
        }
}
```

Now, let run command line to install the module: `php magento setup:upgrade`. Then checking the customer page to see the new attribute.

TOPIC 15: MAGENTO 2 ADD PRODUCT ATTRIBUTE PROGRAMMATICALLY

In this article, we will find out how to create a product attribute in Magento 2 programmatically. As you know, Magento 2 manage Product by EAV model, so we cannot simply add an attribute for product by adding a column forthe product table.

In this article, we will use the Mageplaza HelloWorld module to learn how to add a product attribute. We will start with the InstallData class which located in `app/code/Mageplaza/HelloWorld/Setup/InstallData.php`. The content for this file:

```
<?php
namespace Mageplaza\HelloWorld\Setup;

use Magento\Eav\Setup\EavSetup;
use Magento\Eav\Setup\EavSetupFactory;
use Magento\Framework\Setup\InstallDataInterface;
use Magento\Framework\Setup\ModuleContextInterface;
use Magento\Framework\Setup\ModuleDataSetupInterface;

class InstallData implements InstallDataInterface
{
	private $eavSetupFactory;

	public function __construct(EavSetupFactory $eavSetupFactory)
	{
		$this->eavSetupFactory = $eavSetupFactory;
	}

	public function install(ModuleDataSetupInterface $setup, ModuleContextInterface
$context)
	{
		$eavSetup = $this->eavSetupFactory->create(['setup' => $setup]);
		$eavSetup->addAttribute(
			\Magento\Catalog\Model\Product::ENTITY,
			'sample_attribute',
			[
				'type' => 'int',
				'backend' => '',
				'frontend' => '',
                    'label' => 'Sample Atrribute',
                    'input' => '',
                    'class' => '',
                    'source' => '',
                    'global'                                                     =>
\Magento\Eav\Model\Entity\Attribute\ScopedAttributeInterface::SCOPE_GLOBAL,
                    'visible' => true,
                    'required' => true,
                    'user_defined' => false,
                    'default' => '',
                    'searchable' => false,
                    'filterable' => false,
                    'comparable' => false,
                    'visible_on_front' => false,
                    'used_in_product_listing' => true,
                    'unique' => false,
                    'apply_to' => ''
                ]
            );
        }
}
```

As you can see, all the addAttribute method requires is:

- The type id of the entity which we want to add attribute
- The name of the attribute
- An array of key value pairs to define the attribute such as group, input type, source, label…

All done, please run the upgrade script `php bin/magento setup:upgrade` to install the module and the product attribute `sample_attribute` will be created.

If you want to remove product attribute, you can use method removeAttribute instead of addAttribute. It will be like this:

```
public function install(ModuleDataSetupInterface $setup, ModuleContextInterface
$context)
    {
        $eavSetup = $this->eavSetupFactory->create(['setup' => $setup]);
        $eavSetup->removeAttribute(
          \Magento\Catalog\Model\Product::ENTITY,
            'sample_attribute');
    }
```

REFERENCE

- Magento 2 Developer Doc
- Magento.stackexchange.com
- Community.magento.com
- Github.com/magento2