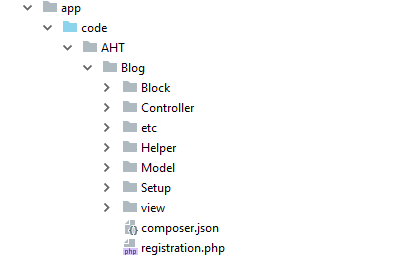
Structure of module Magento 2



* AHT - it is a vendor's name (as a rule it coincides with the company name and should be unique).
* Blog: module (extension) name.
* Block: contains PHP view classes as part of Model View Controller(MVC) vertical implementation of module logic.
* Controller: contains PHP controller classes as part of MVC vertical implementation of module logic.
* etc: contains configuration files; in particular, module.xml, which is required.
* Helper: contains the files that are responsible for performing general tasks for extension objects and variables.
* Model: contains PHP model classes as part of MVC vertical implementation of module logic.
* Setup: includes the files that are necessary to make changes in the database – i.e. creating tables, fields, or other records required for the module performance (optional directory).
* View: contains view files, including static view files, design templates, email templates, and layout files that are necessary for information output on the frontend and backend.
* registration.php: Among other things, this file specifies the directory in which the component is installed by vendors in production environments. By default, composer automatically installs components in the <Magento root dir>/vendor directory.
* etc/module.xml: This file specifies basic information about the component such as the components dependencies and its version number. This version number is used to determine schema and data updates when bin/magento setup:upgrade is run.
* composer.json: Specifies component dependencies and other metadata.

Create AHT Blog module for Magento 2

To create AHT Blog module, you need to complete the following high-level steps:

* Step 1: Create the folder of AHT Blog module
* Step 2: Create etc/module.xml file
* Step 3: Create etc/registration.php file
* Step 4: Enable the module
* Step 5: Create routes.xml file.
* Step 6: Create controller file.
* Step 7: Create View: Block, Layouts, Templates.
* Step 8: CRUD Models

Step 1: Create the folder of AHT Blog module

Name of the module is defined as “VendorName\_ModuleName”. First part is name of the vendor and last part is name of the module: For example: Magento\_Blog, AHT\_OnePageCheckout. Focus on following guide to create the folders:

app/code/AHT/Blog

Step 2: Create etc/module.xml file.

Then, it is necessary to create etc folder and add the module.xml file

app/code/AHT/Blog/etc/module.xml

Contents would be:

<?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="AHT\_Blog" setup\_version="1.0.0">

</module>

</config>

Step 3: Create registration.php file

In this step, we will add registration.php as following guide:

app/code/AHT/Blog/registration.php

Contents would be:

<?php

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

\Magento\Framework\Component\ComponentRegistrar::MODULE,

'AHT\_Blog',

\_\_DIR\_\_

);

Step 4: Enable the module

Finish the step 3, we have already created the Blog module. And we will enable this module in this step

After create the module if you run the command as:

php bin/magento module:status

You should see the module is disable now:

List of disabled modules: AHT\_Blog

Follow exact guide to enable the module right now, let run the command as:

php bin/magento module:enable AHT\_Blog

Or other way, you can access the file:

app/etc/config.php

You will see a long list of modules there, just add your module as well

...

'AHT\_Blog' => 1,

....

Your module should be available now.

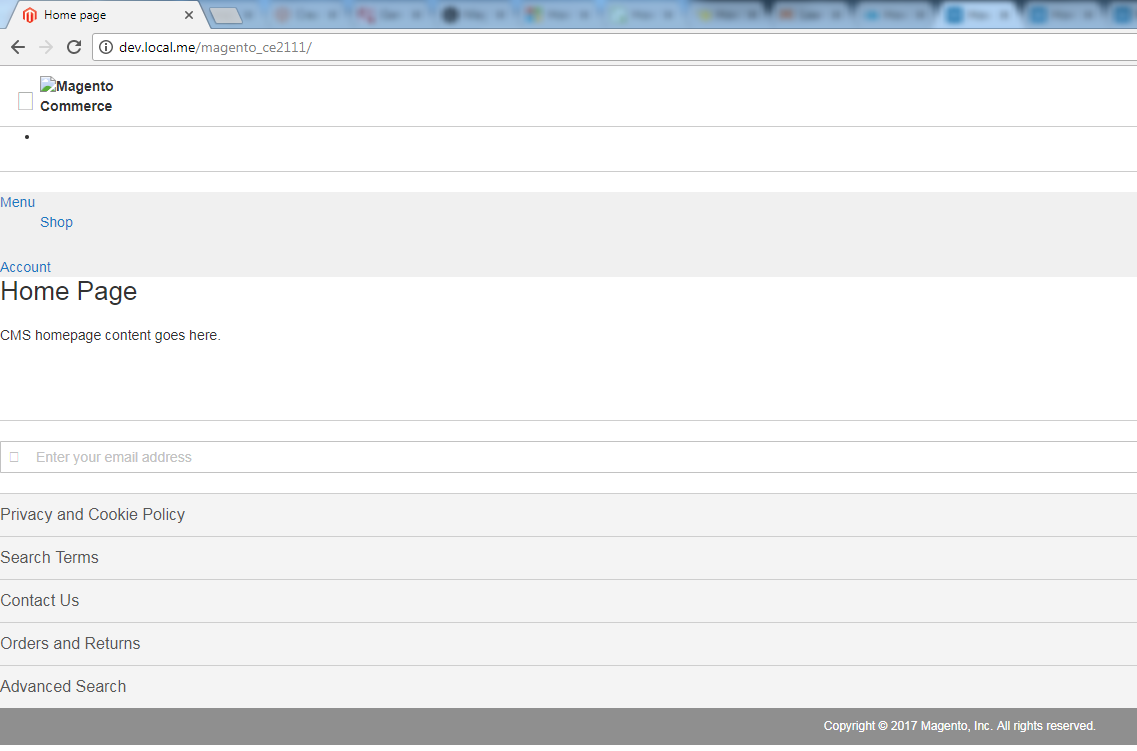
After this step, when you open your website in browser you will get an error saying

Please upgrade your database: Run “bin/magento setup:upgrade” from the Magento root directory.

Let run the command:

php bin/magento setup:upgrade

After complete,when you open your website in browser you will see the layout of the website is broken.



Please run

php bin/magento setup:static-content:deploy

to fix this.

After deploy completed,you can also see your module from backend at System Configuration -> Advanced -> Disable Modules Output.

### Step 5: Create routes.xml file.

Now, we will create a controller to test module.

Before create a controller, we will create a route for Blog module.

Route’s in magento are divided into 3 parts: Route frontname, controller and action as following example:

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

To add route, it is necessary to create routes.xml file

app/code/AHT/Blog/etc/frontend/routes.xml

since this is a frontend route, we added it in frontend/ folder else we need to add it to adminhtml/ folder

Content would be:

<?xml version="1.0" ?>

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

<router id="standard">

<route frontName="blog" id="blog">

<module name="AHT\_Blog"/>

</route>

</router>

</config>

After define the first part of the route, the URL will be displayed as:

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

### Step 6: Create test controller file

Then, we will continue the controller and action

The folder and file you need to create is:

app/code/AHT/Blog/Controller/Index/Test.php

Contents would be:

<?php

namespace AHT\Blog\Controller\Index;

class Test 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()

{

echo "AHT Blog";

exit;

}

}

After completed, please run php bin/magento cache:clean to check result.

Your URL now should be as:

http://<yourhost.com>/blog/index/test

After finish all steps, the output AHT Blog should be displayed in your browser when you open the URL.

### Step 7: Create View: Block, Layouts, Templates

Firstly, we will create a controller to call the layout file .xml

File: app/code/AHT/Blog/Controller/Index/Index.php

<?php

namespace AHT\Blog\Controller\Index;

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

{

protected $\_pageFactory;

public function \_\_construct(

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

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

{

$this->\_pageFactory = $pageFactory;

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

}

public function execute()

{

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

}

}

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

Then, create layout file .xml:

The **Layout** is the major path of view layer in  Magento 2 module. The layout file is a XML file which will define the page structure and will be locate 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 for all the page in it’s area. Otherwhile, the layout file will have name as format: {router\_name}\_{controller\_name}\_{action\_name}.xml.

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/AHT/Blog/view/frontend/layout/blog\_index\_index.xml

<?xml version="1.0"?>

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

<referenceContainer name="content">

<block class="AHT\Blog\Block\Index" name="blog\_index" template="AHT\_Blog::index.phtml" />

</referenceContainer>

</page>

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

Block class: AHT\Blog\Block\Index

Template file: AHT\_Blog::index.phtml

name: It is the required attribute and is used to identify a block as a reference

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/AHT/Blog/Block/Index.php

Contents would be:

<?php

namespace AHT\Blog\Block;

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

{

public function \_\_construct(\Magento\Framework\View\Element\Template\Context $context)

{

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

}

public function getBlogInfo()

{

return \_\_('AHT Blog module');

}

public function getPosts()

{

$post = $this->\_postFactory->create();

$collection = $post->getCollection();

return $collection;

}

}

Every block in Magento 2 must extend from Magento\Framework\View\Element\Template. In this block we will define a method getBlogInfo() to show the word “AHT Blog module”. We will use it in template file.

Create a template file call index.phtml

app/code/AHT/Blog/view/frontend/templates/index.phtml

Insert the following code:

<?php

/\*\*

\* @var \AHT\Blog\Block\Index $block

\*/

echo $block->getBlogInfo();

In the layout file, we define the template by AHT\_Blog::index.phtml. It mean that Magento will find the file name index.phtml in templates folder of module AHT\_Blog. 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 getBlogInfo() in Block. It’s done, please access to this page again (http://<yourhost>/blog/index/index) and see the result.

### Step 8: CRUD Models

CRUD Models in Magento 2 can manage data in database easily, you don’t need to write many line of code to create a CRUD. **CRUD** is 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**.

Create a table aht\_blog\_post with the following columns:

* post\_id - the post unique identifier
* name - the name of the post
* url\_key - url of the post
* image - the image of the post
* content - the content of the post
* status - the status of the post
* created\_at - the date created of the post
* updated\_at - the date updated of the post

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

app/code/AHT/Blog/Setup/InstallSchema.php

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

<?php

namespace AHT\Blog\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();

if (!$installer->tableExists('aht\_blog\_post')) {

$table = $installer->getConnection()->newTable(

$installer->getTable('aht\_blog\_post')

)

->addColumn(

'post\_id',

\Magento\Framework\DB\Ddl\Table::TYPE\_INTEGER,

null,

[

'identity' => true,

'nullable' => false,

'primary' => true,

'unsigned' => true,

],

'Post ID'

)

->addColumn(

'name',

\Magento\Framework\DB\Ddl\Table::TYPE\_TEXT,

255,

['nullable => false'],

'Name'

)

->addColumn(

'url\_key',

\Magento\Framework\DB\Ddl\Table::TYPE\_TEXT,

255,

[],

'URL Key'

)

->addColumn(

'image',

\Magento\Framework\DB\Ddl\Table::TYPE\_TEXT,

null,

['nullable' => true, 'default' => null],

'Image'

)

->addColumn(

'content',

\Magento\Framework\DB\Ddl\Table::TYPE\_TEXT,

'64k',

['nullable' => false],

'Content'

)

->addColumn(

'status',

\Magento\Framework\DB\Ddl\Table:: TYPE\_SMALLINT,

null,

['nullable' => false, 'default' => '1'],

'Status'

)

->addColumn(

'created\_at',

\Magento\Framework\DB\Ddl\Table::TYPE\_TIMESTAMP,

null,

['nullable' => false, 'default' => \Magento\Framework\DB\Ddl\Table::TIMESTAMP\_INIT],

'Created At'

)->addColumn(

'updated\_at',

\Magento\Framework\DB\Ddl\Table::TYPE\_TIMESTAMP,

null,

['nullable' => false, 'default' => \Magento\Framework\DB\Ddl\Table::TIMESTAMP\_INIT\_UPDATE],

'Updated At')

->setComment('Blog Post Table');

$installer->getConnection()->createTable($table);

$installer->getConnection()->addIndex(

$installer->getTable('aht\_blog\_post'),

$setup->getIdxName(

$installer->getTable('aht\_blog\_post'),

['name','url\_key','image','content'],

\Magento\Framework\DB\Adapter\AdapterInterface::INDEX\_TYPE\_FULLTEXT

),

['name','url\_key','image','content'],

\Magento\Framework\DB\Adapter\AdapterInterface::INDEX\_TYPE\_FULLTEXT

);

}

$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 and change attribute setup\_version greater than current setup version in module.xml at app/code/AHT/Blog/etc/module.xml.

Contents would be:

File: app/code/AHT/Blog/etc/module.xml

<?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="AHT\_Blog" setup\_version="1.0.1">

</module>

</config>

In module.xml file, we changed the attribute to 1.0.1 greater than setup\_version before

File: app/code/AHT/Blog/Setup/UpgradeSchema.php

<?php

namespace AHT\Blog\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', '<')) {

if (!$installer->tableExists('aht\_blog\_post')) {

$table = $installer->getConnection()->newTable(

$installer->getTable('aht\_blog\_post')

)

->addColumn(

'post\_id',

\Magento\Framework\DB\Ddl\Table::TYPE\_INTEGER,

null,

[

'identity' => true,

'nullable' => false,

'primary' => true,

'unsigned' => true,

],

'Post ID'

)

->addColumn(

'name',

\Magento\Framework\DB\Ddl\Table::TYPE\_TEXT,

255,

['nullable => false'],

'Name'

)

->addColumn(

'url\_key',

\Magento\Framework\DB\Ddl\Table::TYPE\_TEXT,

255,

[],

'URL Key'

)

->addColumn(

'image',

\Magento\Framework\DB\Ddl\Table::TYPE\_TEXT,

null,

['nullable' => true, 'default' => null],

'Image'

)

->addColumn(

'content',

\Magento\Framework\DB\Ddl\Table::TYPE\_TEXT,

'64k',

['nullable' => false],

'Content'

)

->addColumn(

'status',

\Magento\Framework\DB\Ddl\Table:: TYPE\_SMALLINT,

null,

['nullable' => false, 'default' => '1'],

'Status'

)

->addColumn(

'created\_at',

\Magento\Framework\DB\Ddl\Table::TYPE\_TIMESTAMP,

null,

['nullable' => false, 'default' => \Magento\Framework\DB\Ddl\Table::TIMESTAMP\_INIT],

'Created At'

)->addColumn(

'updated\_at',

\Magento\Framework\DB\Ddl\Table::TYPE\_TIMESTAMP,

null,

['nullable' => false, 'default' => \Magento\Framework\DB\Ddl\Table::TIMESTAMP\_INIT\_UPDATE],

'Updated At')

->setComment('Blog Post Table');

$installer->getConnection()->createTable($table);

$installer->getConnection()->addIndex(

$installer->getTable('aht\_blog\_post'),

$setup->getIdxName(

$installer->getTable('aht\_blog\_post'),

['name','url\_key','image','content'],

\Magento\Framework\DB\Adapter\AdapterInterface::INDEX\_TYPE\_FULLTEXT

),

['name','url\_key','image','content'],

\Magento\Framework\DB\Adapter\AdapterInterface::INDEX\_TYPE\_FULLTEXT

);

}

}

$installer->endSetup();

}

}

After this please run this command line:

php bin/magento setup:upgrade

When you run upgrade completed, please continue run deploy like this

php bin/magento setup:static-content:deploy

Now checking your database, you will see a table with name aht\_blog\_post 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 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 aht\_blog\_post. 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/AHT/Blog/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 first time install the module. If you want to change the database when upgrade module, please try to use UpgradeSchema.php and UpgradeData.php.

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. We have to create Model, Resource Model, Resource Model Conllection to manage data in table: aht\_blog\_post as I mentioned above.

Now we will create the model file:

app/code/AHT/Blog/Model/Post.php

And this is the content of that file:

<?php

namespace AHT\Blog\Model;

class Post extends \Magento\Framework\Model\AbstractModel implements \Magento\Framework\DataObject\IdentityInterface

{

const CACHE\_TAG = 'aht\_blog\_post';

protected $\_cacheTag = 'aht\_blog\_post';

protected $\_eventPrefix = 'aht\_blog\_post';

protected function \_construct()

{

$this->\_init('AHT\Blog\Model\ResourceModel\Post');

}

public function getIdentities()

{

return [self::CACHE\_TAG . '\_' . $this->getId()];

}

public function getDefaultValues()

{

$values = [];

return $values;

}

}

This model class will extends AbstractModel class Magento\Framework\Model\AbstractModel and implements \Magento\Framework\DataObject\IdentityInterface. The IdentityInterface will force Model class 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 have 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 AHT\Post\Model\ResourceModel\Post The last thing about model is some variable which you should you in your model:

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

As you know, the model file contain overall database logic, it do not execute sql queries. The resource model will do that. Now we will create the Resource Model for this table: app/code/AHT/Blog/Model/ResourceModel/Post.php

Content for this file:

<?php

namespace AHT\Blog\Model\ResourceModel;

class Post extends \Magento\Framework\Model\ResourceModel\Db\AbstractDb

{

public function \_\_construct(

\Magento\Framework\Model\ResourceModel\Db\Context $context

)

{

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

}

protected function \_construct()

{

$this->\_init('aht\_blog\_post', 'post\_id');

}

}

Every CRUD resource model in Magento must extends abstract class \Magento\Framework\Model\ResourceModel\Db\AbstractDb which contain the functions for fetching information from 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 aht\_blog\_post and the primary key post\_id.

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:

app/code/AHT/Blog/Model/ResourceModel/Post/Collection.php

The content for this file:

<?php

namespace AHT\Blog\Model\ResourceModel\Post;

class Collection extends \Magento\Framework\Model\ResourceModel\Db\Collection\AbstractCollection

{

protected $\_idFieldName = 'post\_id';

protected $\_eventPrefix = 'aht\_blog\_post\_collection';

protected $\_eventObject = 'post\_collection';

/\*\*

\* Define resource model

\*

\* @return void

\*/

protected function \_construct()

{

$this->\_init('AHT\Blog\Model\Post', 'AHT\Blog\Model\ResourceModel\Post');

}

}

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.

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 model. As you know in OOP, a factory method will be used to instantiate an object. In Magento, the Factory Object do the same thing.

The Factory class name is the name of Model class and append with the ‘Factory’ word. So for our example, we will have PostFactory 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

In this case, it will be:

var/generation/AHT/Blog/Model/PostFactory.php

For example, we will call the model to get data in controller.

app/code/AHT/Blog/Controller/Index/Index.php

Content for this file:

<?php

namespace AHT\Blog\Controller\Index;

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

{

protected $\_pageFactory;

protected $\_postFactory;

public function \_\_construct(

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

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

\AHT\Blog\Model\PostFactory $postFactory

)

{

$this->\_pageFactory = $pageFactory;

$this->\_postFactory = $postFactory;

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

}

public function execute()

{

$post = $this->\_postFactory->create();

$collection = $post->getCollection();

foreach($collection as $item){

echo "<pre>";

print\_r($item->getData());

echo "</pre>";

}

exit();

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

}

}

As you see in this controller, the PostFactory object will be created in the \_construct() function. In the execute() function, we use $post = $this->\_postFactory->create(); to create the model object.

Now, You need go to phpmyadmin and open aht\_blog\_post table to add some record to test post model work.

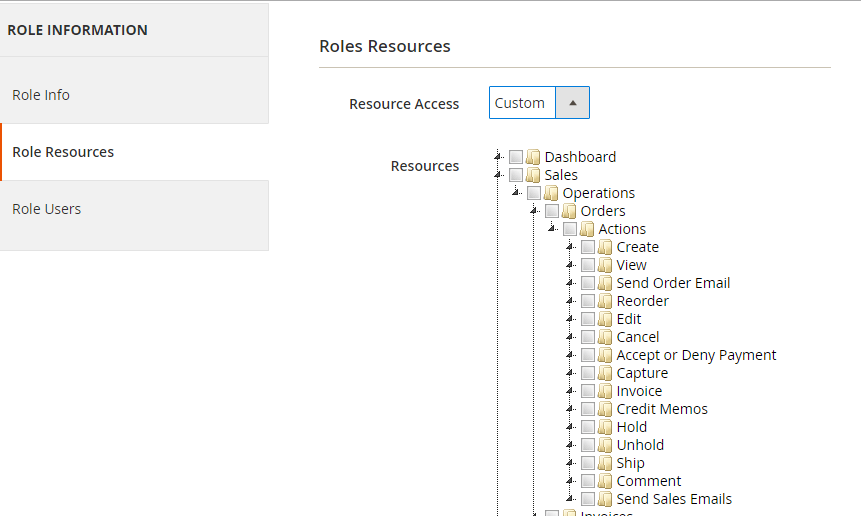
After completed, let’s open browser and navigate to

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

and see the result.

## 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:



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 user who belong to this role will be limit access to the resource which you choose. They cannot see and access to other one.

## To Create Admin Admin ACL

* Step 1: Create ACL rule
* Step 2: Flush Magento cache

## Step 1: Create ACL rule

Now, we will see how to add our module to ACL role. We will use a previous simple module Blog to do this. As in the Admin Menu and System Configuration article, you saw that we alway have a resource attribute when create 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 app/code/{namespace}/{module}/etc/acl.xml. Let’s create this file for our simple Module:

File: app/code/AHT/Blog/etc/acl.xml

Contents would be:

<?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="AHT\_Blog::blog" title="Blog" sortOrder="10">

<resource id="AHT\_Blog::post" title="Posts" sortOrder="10"/>

<resource id="AHT\_Blog::blog\_configuration" title="Configuration" sortOrder="100" />

</resource>

<resource id="Magento\_Backend::stores">

<resource id="Magento\_Backend::stores\_settings">

<resource id="Magento\_Config::config">

<resource id="AHT\_Blog::blog\_config" title="Blog"/>

</resource>

</resource>

</resource>

</resource>

</resources>

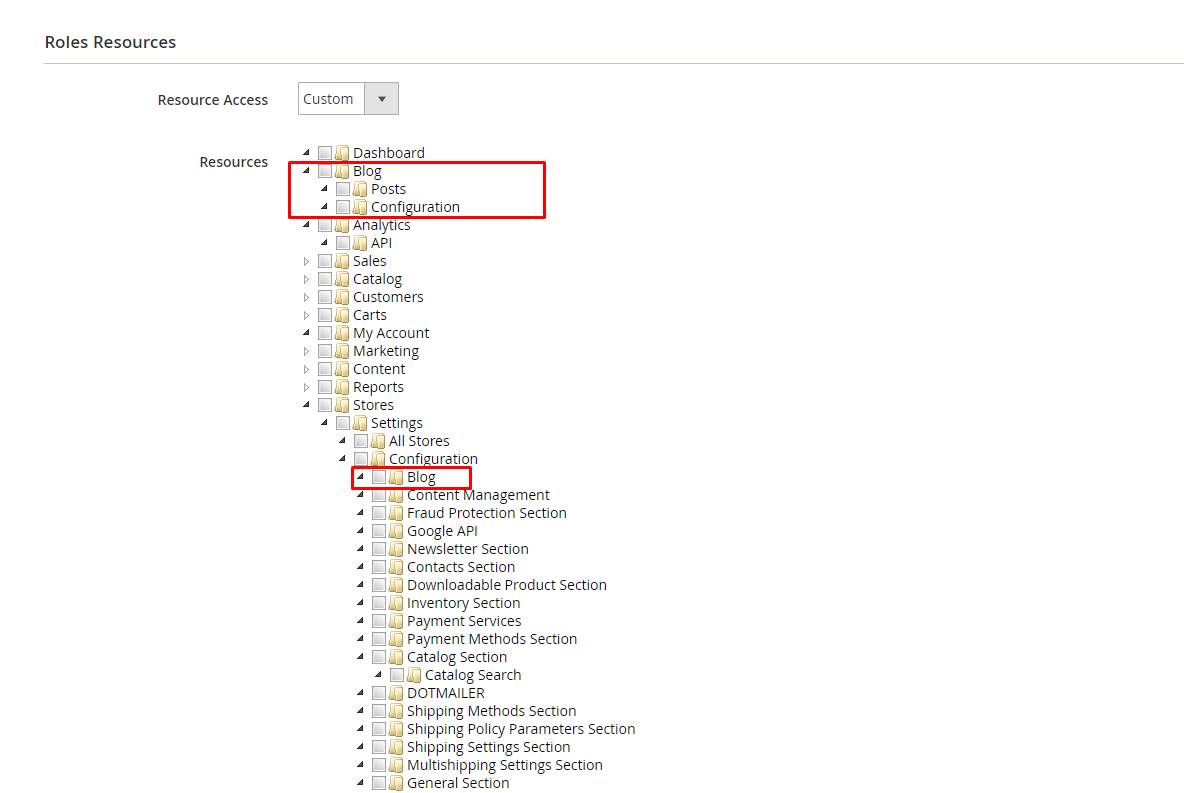
</acl>

</config>

Our resource will be placed as child of Magento\_Backend::admin. Each resource will have an id, title and sortOrder attribute:

* Id attribute is the identify of this resource. You can use this when define 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 define the position of this resource in tree.

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



## Step 2: Flush Magento cache

Make sure it admin menu items are displayed on Magento 2 admin.

## 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/AHT/Blog/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="AHT\_Blog::blog" title="Blog" module="AHT\_Blog" sortOrder="10" resource="AHT\_Blog::blog"/>

<add id="AHT\_Blog::post" title="Manage Posts" module="AHT\_Blog" sortOrder="10" action="blog/post" resource="AHT\_Blog::post" parent="AHT\_Blog::blog"/>

<add id="AHT\_Blog::blog\_configuration" title="Configuration" module="AHT\_Blog" sortOrder="99" parent="AHT\_Blog::blog" action="adminhtml/system\_config/edit/section/blog" resource="AHT\_Blog::blog\_configuration"/>

</menu>

</config>

In this example, we will create a level-0 menu named “Blog” and two sub-menus named “Manage Posts” 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:

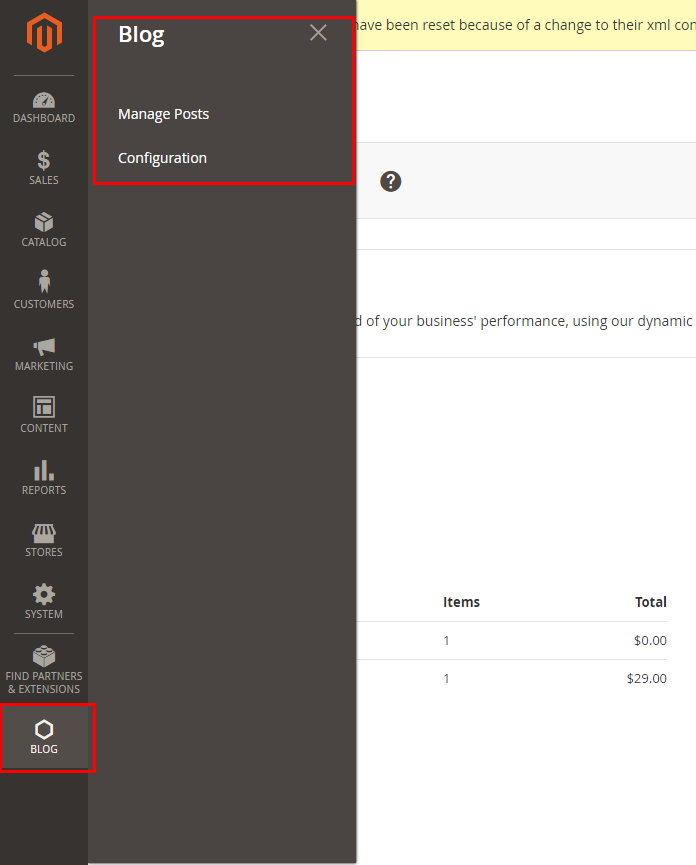
<add id="AHT\_Blog::post" title="Manage Posts" module="AHT\_Blog" sortOrder="10" action="AHT\_Blog/post" resource="AHT\_Blog::post" parent="AHT\_Blog::blog"/>

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 is defined the module which this menu is belong to.
* The sortOrder attribute is defined the position of the menu. Lower value will display on top of menu.
* The parent attribute is an Id of other menu node. It will tell Magento that this menu is a child of another menu. In this example, we have parent=”AHT\_Blog::blog”, so we - know this menu “Manage Posts” is a child of “Blog” menu and it will show inside of Blog 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 Blog, controller Post 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.

### Step 3: Flush Magento cache

Now to go Magento 2 Admin and see result:

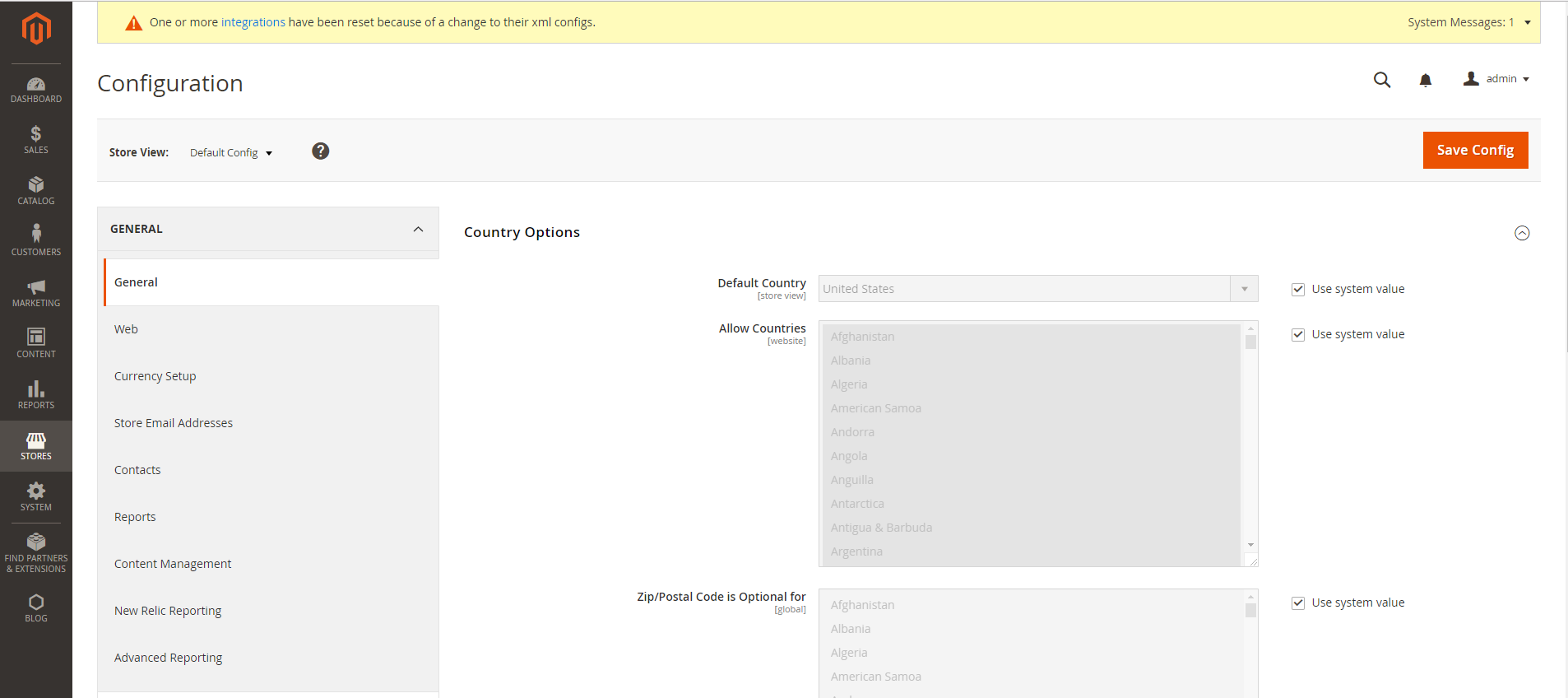


## To Create system.xml

* Step 1: Create System.xml
* Step 2: Set default value
* Step 3: Flush Magento cache
* Step 4: Get value from configuration

### Step 1: Create System.xml

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



So let’s start to create a **simple configuration** for the simple module AHT Blog. The system.xml is located in etc/adminhtml folder of the module, we will create it a new Tab for our vendor “AHT”, a new Section for our module Blog, a Group to contain some simple fields: enable module and text.

File: app/code/AHT/Blog/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="aht" translate="label" sortOrder="10">

<label>AHT</label>

</tab>

<section id="blog" translate="label" sortOrder="130" showInDefault="1" showInWebsite="1" showInStore="1">

<class>separator-top</class>

<label>Blog</label>

<tab>aht</tab>

<resource>AHT\_Blog::blog\_config</resource>

<group id="general" translate="label" type="text" sortOrder="10" showInDefault="1" showInWebsite="1" showInStore="1">

<label>General Configuration</label>

<field id="enabled" translate="label" type="select" sortOrder="1" showInDefault="1" showInWebsite="1" showInStore="1">

<label>Enabled</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="1" showInStore="1">

<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 let Magento know which title need to translate
  + Label element child is the text which will show as tab title.
* The Section element will have 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 group and some other child elements:

* Class: this value will be added as class for this element. You should you 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 is belong 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 field and some attributes which is same as Sections.
* Fields: is the main path of this page. It will save the data which we want to setting. 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 source\_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/AHT/Blog/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>

<blog>

<general>

<enabled>1</enabled>

<display\_text>AHT Blog Module Text</display\_text>

</general>

</Blog>

</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>

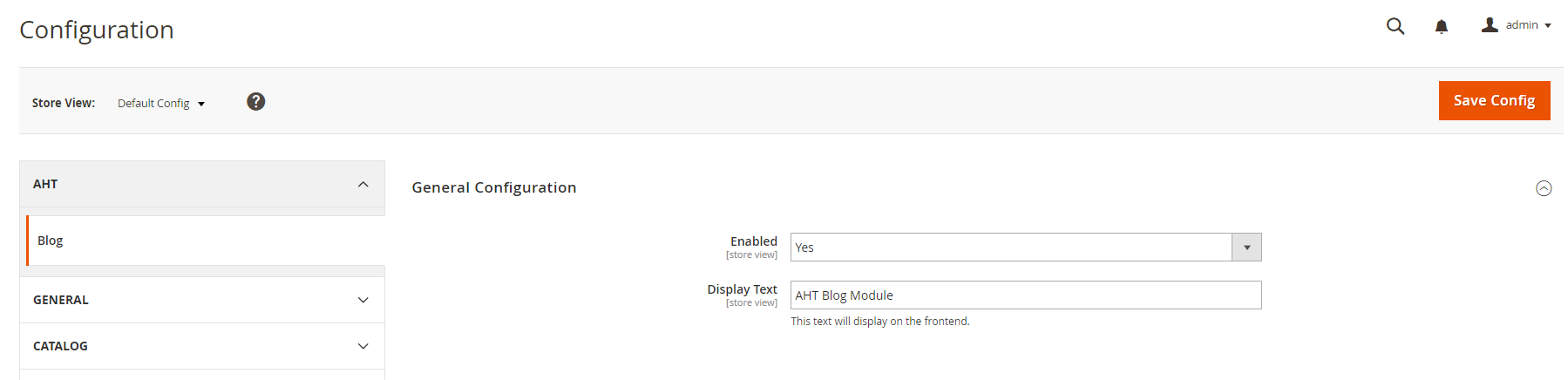
<field>{value}</field>

</group>

</section>

</default>

### Step 3: Flush Magento Cache



### Step 4: Get value from configuration

First all of let’s save value and flush cache, then you can get saved value from database.

In the system.xml, we have added 2 fields: enabled and display\_text. So the path should be:

* blog/general/enable
* blog/general/display\_text

#### 4.1 Simple calling:

$this->scopeConfig->getValue('blog/general/enable', \Magento\Store\Model\ScopeInterface::SCOPE\_STORE);

$this->scopeConfig->getValue('blog/general/display\_text', \Magento\Store\Model\ScopeInterface::SCOPE\_STORE);

#### 4.2 Create a helper file (standard)

Create file: app/code/AHT/Blog/Helper/Data.php

<?php

namespace AHT\Blog\Helper;

use Magento\Framework\App\Helper\AbstractHelper;

use Magento\Store\Model\ScopeInterface;

class Data extends AbstractHelper

{

public function getConfig($configPath, $storeId = null)

{

return $this->scopeConfig->getValue(

$configPath, ScopeInterface::SCOPE\_STORE, $storeId

);

}

}

Now, we try get it in phtml file.

File: app\code\AHT\Blog\view\frontend\templates\index.phtml

<?php

$helper = $this->helper('AHT\Blog\Helper\Data');

?>

<p><?php echo $helper->getConfig('blog/general/enabled'); ?></p>

<p><?php echo $helper->getConfig('blog/general/display\_text'); ?></p>

Please run php bin/magento cache:clean to clear cache and check the result.

## To Create Admin Grid

* Step 1: Create routes admin
* Step 2: Create admin menu
* Step 3: Create Controller
* Step 4: Create Admin Grid

### Step 1: Create routes admin

File: app/code/AHT/Blog/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">

<router id="admin">

<route id="blog" frontName="blog">

<module name="AHT\_Blog"/>

</route>

</router>

</config>

### Step 2: Create admin menu

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

blog/post/index

### Step 3: Create Controller

Create controller file called Index.php

app/code/AHT/Blog/Controller/Adminhtml/Post/Index.php

With the following content:

<?php

namespace AHT\Blog\Controller\Adminhtml\Post;

class Index extends \AHT\Blog\Controller\Adminhtml\Post

{

public function execute()

{

$this->\_initAction();

$this->\_view->getPage()->getConfig()->getTitle()->prepend(\_\_('Posts'));

$this->\_view->renderLayout();

}

}

### Step 4: Create Admin Grid

#### Step 4.1: 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/AHT/Blog/etc/di.xml

With the following content:

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

<type name="AHT\Blog\Model\Post">

<arguments>

<argument name="resource" xsi:type="object">AHT\Blog\Model\ResourceModel\Post</argument>

</arguments>

</type>

</config>

#### Step 4.2: Create block for this grid

File: app/code/AHT/Block/Adminhtml/Post.php

<?php

namespace AHT\Blog\Block\Adminhtml;

class Post extends \Magento\Backend\Block\Widget\Grid\Container

{

protected function \_construct()

{

$this->\_controller = 'adminhtml\_post';

$this->\_blockGroup = 'AHT\_Blog';

$this->\_headerText = \_\_('Posts');

$this->\_addButtonLabel = \_\_('Create New Post');

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 Blog, I put the Grid.php file inside of the Adminhtml/Post folder
* \_headerText is the Grid page title
* \_addButtonLabel is the label of the Add new button.

#### Step 4.3: 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/AHT/Blog/view/adminhtml/layout/blog\_post\_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\_configuration.xsd">

<body>

<referenceContainer name="content">

<block class="AHT\Blog\Block\Adminhtml\Post" name="adminhtml.post.container">

<block class="Magento\Backend\Block\Widget\Grid" name="adminhtml.post.container.grid" as="grid">

<arguments>

<argument name="id" xsi:type="string">postGrid</argument>

<argument name="dataSource" xsi:type="object">AHT\Blog\Model\ResourceModel\Post\Collection</argument>

<argument name="default\_sort" xsi:type="string">name</argument>

<argument name="default\_dir" xsi:type="string">ASC</argument>

<argument name="save\_parameters\_in\_session" xsi:type="boolean">true</argument>

<argument name="use\_ajax" xsi:type="boolean">false</argument>

<argument name="grid\_url" xsi:type="url" path="blog/post/index">

<param name="\_current">1</param>

</argument>

</arguments>

<block class="Magento\Backend\Block\Widget\Grid\Massaction" name="adminhtml.post.grid.massaction" as="grid.massaction">

<arguments>

<argument name="massaction\_id\_field" xsi:type="string">post\_id</argument>

<argument name="form\_field\_name" xsi:type="string">ids</argument>

<argument name="options" xsi:type="array">

<item name="delete" xsi:type="array">

<item name="label" xsi:type="string" translate="true">Delete</item>

<item name="url" xsi:type="string">blog/post/massDelete</item>

<item name="confirm" xsi:type="string" translate="true">Are you sure you want to delete?</item>

</item>

<item name="change\_status" xsi:type="array">

<item name="label" xsi:type="string" translate="true">Change Status</item>

<item name="url" xsi:type="string">blog/post/massStatus</item>

<item name="block\_name" xsi:type="string">grid.status</item>

</item>

</argument>

</arguments>

<block class="Magento\Backend\Block\Widget\Grid\Massaction\Additional" name="adminhtml.post.grid.status" as="grid.status">

<arguments>

<argument name="fields" xsi:type="array">

<item name="visibility" xsi:type="array">

<item name="name" xsi:type="string">status</item>

<item name="type" xsi:type="string">select</item>

<item name="class" xsi:type="string">required-entry</item>

<item name="label" xsi:type="string" translate="true">Status</item>

<item name="values" xsi:type="array">

<item name="disable" xsi:type="array">

<item name="value" xsi:type="string">0</item>

<item name="label" xsi:type="string" translate="true">Disabled</item>

</item>

<item name="enable" xsi:type="array">

<item name="value" xsi:type="string">1</item>

<item name="label" xsi:type="string" translate="true">Enabled</item>

</item>

</item>

</item>

</argument>

</arguments>

</block>

</block>

<block class="Magento\Backend\Block\Widget\Grid\ColumnSet" name="adminhtml.post.container.grid.columnSet" as="grid.columnSet">

<arguments>

<argument name="id" xsi:type="string">postGrid</argument>

<argument name="rowUrl" xsi:type="array">

<item name="path" xsi:type="string">\*/\*/edit</item>

<item name="extraParamsTemplate" xsi:type="array">

<item name="id" xsi:type="string">getId</item>

</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">post\_id</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="name">

<arguments>

<argument name="header" xsi:type="string" translate="true">Name</argument>

<argument name="index" xsi:type="string">name</argument>

</arguments>

</block>

<block class="Magento\Backend\Block\Widget\Grid\Column" as="created\_at">

<arguments>

<argument name="header" xsi:type="string" translate="true">Created</argument>

<argument name="index" xsi:type="string">created\_at</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 class="Magento\Backend\Block\Widget\Grid\Column" as="updated\_at">

<arguments>

<argument name="header" xsi:type="string" translate="true">Modified</argument>

<argument name="index" xsi:type="string">updated\_at</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 class="Magento\Backend\Block\Widget\Grid\Column" as="status">

<arguments>

<argument name="header" xsi:type="string" translate="true">Status</argument>

<argument name="index" xsi:type="string">status</argument>

<argument name="type" xsi:type="string">options</argument>

<argument name="options" xsi:type="array">

<item name="disable" xsi:type="array">

<item name="value" xsi:type="string">0</item>

<item name="label" xsi:type="string" translate="true">Disabled</item>

</item>

<item name="enable" xsi:type="array">

<item name="value" xsi:type="string">1</item>

<item name="label" xsi:type="string" translate="true">Enabled</item>

</item>

</argument>

</arguments>

</block>

<block class="Magento\Backend\Block\Widget\Grid\Column" as="action">

<arguments>

<argument name="id" xsi:type="string">action</argument>

<argument name="header" xsi:type="string" translate="true">Action</argument>

<argument name="type" xsi:type="string">action</argument>

<argument name="getter" xsi:type="string">getId</argument>

<argument name="filter" xsi:type="boolean">false</argument>

<argument name="sortable" xsi:type="boolean">false</argument>

<argument name="is\_system" xsi:type="boolean">true</argument>

<argument name="actions" xsi:type="array">

<item name="view\_action" xsi:type="array">

<item name="caption" xsi:type="string" translate="true">Edit</item>

<item name="url" xsi:type="array">

<item name="base" xsi:type="string">blog/post/edit</item>

</item>

<item name="field" xsi:type="string">id</item>

</item>

</argument>

<argument name="header\_css\_class" xsi:type="string">col-actions</argument>

<argument name="column\_css\_class" xsi:type="string">col-actions</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.

## To Create Admin CRUD

* Step 1: Create abtract controller
* Step 2: Create NewAction controller
* Step 3: Create Edit controller
* Step 4: Create Block Edit
* Step 5: Create Block Edit Form
* Step 6: Create Save controller
* Step 7: Create massDelete controller
* Step 8: Create massStatus controller

### Step 1: Create abstract controller

<?php

namespace AHT\Blog\Controller\Adminhtml;

use Magento\Backend\App\Action;

abstract class Post extends Action

{

protected function \_initAction()

{

$this->\_view->loadLayout();

$this->\_setActiveMenu(

'AHT\_Blog::blog'

)->\_addBreadcrumb(

\_\_('Blog'),

\_\_('Blog')

);

return $this;

}

protected function \_isAllowed()

{

return $this->\_authorization->isAllowed('AHT\_Blog::blog');

}

}

### Step 2: Create NewAction controller

<?php

namespace AHT\Blog\Controller\Adminhtml\Post;

class NewAction extends \AHT\Blog\Controller\Adminhtml\Post

{

public function execute()

{

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

}

}

### Step 3: Create Edit controller

<?php

namespace AHT\Blog\Controller\Adminhtml\Post;

class Edit extends \AHT\Blog\Controller\Adminhtml\Post

{

protected $\_coreRegistry = null;

public function \_\_construct(\Magento\Backend\App\Action\Context $context, \Magento\Framework\Registry $coreRegistry)

{

$this->\_coreRegistry = $coreRegistry;

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

}

public function execute()

{

$id = $this->getRequest()->getParam('id');

$model = $this->\_objectManager->create('AHT\Blog\Model\Post');

if ($id) {

$model->load($id);

if (!$model->getId()) {

$this->messageManager->addError(\_\_('This item no longer exists.'));

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

return;

}

}

$data = $this->\_objectManager->get('Magento\Backend\Model\Session')->getFormData(true);

if (!empty($data)) {

$model->setData($data);

}

$this->\_coreRegistry->register('blog\_post', $model);

$this->\_initAction()->\_addBreadcrumb(

$id ? \_\_('Edit %1', $model->getName()) : \_\_('New Item'),

$id ? \_\_('Edit %1', $model->getName()) : \_\_('New Item')

)->\_addContent(

$this->\_view->getLayout()->createBlock('AHT\Blog\Block\Adminhtml\Edit')

);

$this->\_view->getPage()->getConfig()->getTitle()->prepend(\_\_('Posts'));

$this->\_view->getPage()->getConfig()->getTitle()->prepend(

$model->getId() ? $model->getName() : \_\_('New Item')

);

$this->\_view->renderLayout();

}

}

### Step 4: Create Edit Block

<?php

namespace AHT\Blog\Block\Adminhtml;

class Edit extends \Magento\Backend\Block\Widget\Form\Container

{

protected $\_coreRegistry = null;

public function \_\_construct(

\Magento\Backend\Block\Widget\Context $context,

\Magento\Framework\Registry $registry,

array $data = []

)

{

$this->\_coreRegistry = $registry;

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

}

protected function \_construct()

{

$this->\_objectId = 'id';

$this->\_controller = 'adminhtml';

$this->\_blockGroup = 'AHT\_Blog';

parent::\_construct();

$this->buttonList->add(

'saveandcontinue',

[

'label' => \_\_('Save and Continue Edit'),

'class' => 'save',

'data\_attribute' => [

'mage-init' => ['button' => ['event' => 'saveAndContinueEdit', 'target' => '#edit\_form']],

]

],

-100

);

}

public function getHeaderText()

{

if ($this->\_coreRegistry->registry('blog\_post')->getId()) {

return \_\_('Edit %1', $this->\_coreRegistry->registry('blog\_post')->getName());

} else {

return \_\_('New Item');

}

}

protected function \_isAllowedAction($resourceId)

{

return $this->\_authorization->isAllowed($resourceId);

}

}

### Step 5: Create Block Edit Form

<?php

namespace AHT\Blog\Block\Adminhtml\Edit;

class Form extends \Magento\Backend\Block\Widget\Form\Generic

{

protected $\_wysiwygConfig;

protected $\_systemStore;

public function \_\_construct(

\Magento\Backend\Block\Template\Context $context,

\Magento\Framework\Registry $registry,

\Magento\Framework\Data\FormFactory $formFactory,

\Magento\Cms\Model\Wysiwyg\Config $wysiwygConfig,

\Magento\Store\Model\System\Store $systemStore,

array $data = []

)

{

$this->\_wysiwygConfig = $wysiwygConfig;

$this->\_systemStore = $systemStore;

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

}

protected function \_construct()

{

parent::\_construct();

$this->setId('post\_\_form');

$this->setTitle(\_\_('Information'));

}

protected function \_prepareForm()

{

$model = $this->\_coreRegistry->registry('blog\_post');

$form = $this->\_formFactory->create(

['data' => ['id' => 'edit\_form', 'action' => $this->getData('action'), 'method' => 'post', 'enctype' => 'multipart/form-data']]

);

$fieldset = $form->addFieldset('add\_post\_form', ['legend' => \_\_('Post')]);

if ($model->getId()) {

$fieldset->addField('post\_id', 'hidden', ['name' => 'post\_id']);

}

$fieldset->addField(

'name',

'text',

[

'label' => \_\_('Name'),

'name' => 'name',

'required' => true,

'value' => $model->getName()

]

);

$fieldset->addField(

'url\_key',

'text',

[

'label' => \_\_('Url Key'),

'name' => 'url\_key',

'required' => true,

'value' => $model->getUrlKey()

]

);

$fieldset->addField(

'image',

'image',

[

'label' => \_\_('Image'),

'name' => 'image',

'required' => true,

'value' => $model->getImage()

]

);

$wysiwygConfig = $this->\_wysiwygConfig->getConfig();

$fieldset->addField(

'content',

'editor',

['name' => 'content', 'label' => \_\_('Content'), 'title' => \_\_('Content'), 'required' => true, 'config' => $wysiwygConfig]

);

$fieldset->addField(

'status',

'select',

[

'label' => \_\_('Status'),

'name' => 'status',

'required' => false,

'options' => ['1' => \_\_('Enabled'), '0' => \_\_('Disabled')]

]

);

$form->setValues($model->getData());

$form->setUseContainer(true);

$this->setForm($form);

return parent::\_prepareForm();

}

}

### Step 6: Create Save controller

<?php

namespace AHT\Blog\Controller\Adminhtml\Post;

use Magento\Framework\App\Filesystem\DirectoryList;

class Save extends \AHT\Blog\Controller\Adminhtml\Post

{

public function execute()

{

$resultRedirect = $this->resultRedirectFactory->create();

$data = $this->getRequest()->getPostValue();

if ($data) {

$id = $this->getRequest()->getParam('post\_id');

$model = $this->\_objectManager->create('AHT\Blog\Model\Post')->load($id);

if (!$model->getId() && $id) {

$this->messageManager->addError(\_\_('This item no longer exists.'));

return $resultRedirect->setPath('\*/\*/');

}

if (isset($\_FILES['image']['name']) && $\_FILES['image']['name'] != '') {

$uploader = $this->\_objectManager->create(

'Magento\MediaStorage\Model\File\Uploader',

['fileId' => 'image']

);

$uploader->setAllowedExtensions(['jpg', 'jpeg', 'gif', 'png', 'svg', 'JPG', 'JPEG', 'GIF', 'PNG', 'SVG']);

$uploader->setAllowRenameFiles(true);

$uploader->setAllowCreateFolders(true);

$uploader->setFilesDispersion(true);

$ext = pathinfo($\_FILES['image']['name'], PATHINFO\_EXTENSION);

if ($uploader->checkAllowedExtension($ext)) {

$path = $this->\_objectManager->get('Magento\Framework\Filesystem')->getDirectoryRead(DirectoryList::MEDIA)

->getAbsolutePath('blog/images/');

$uploader->save($path);

$fileName = $uploader->getUploadedFileName();

if ($fileName) {

$data['image'] = 'blog/images' . $fileName;

}

} else {

$this->messageManager->addError(\_\_('Disallowed file type.'));

return $this->redirectToEdit($model, $data);

}

} else {

if (isset($data['image']['delete']) && $data['image']['delete'] == 1) {

$data['image'] = '';

} else {

unset($data['image']);

}

}

$model->setData($data);

try {

$model->save();

$this->messageManager->addSuccess(\_\_('You saved the item.'));

$this->\_objectManager->get('Magento\Backend\Model\Session')->setFormData(false);

if ($this->getRequest()->getParam('back')) {

return $resultRedirect->setPath('\*/\*/edit', ['post\_id' => $model->getId()]);

}

return $resultRedirect->setPath('\*/\*/');

} catch (\Exception $e) {

$this->messageManager->addError($e->getMessage());

$this->\_objectManager->get('Magento\Backend\Model\Session')->setFormData($data);

return $resultRedirect->setPath('\*/\*/edit', ['post\_id' => $this->getRequest()->getParam('post\_id')]);

}

}

return $resultRedirect->setPath('\*/\*/');

}

}

### Step 7: Create massDelete controller

<?php

namespace AHT\Blog\Controller\Adminhtml\Post;

class massDelete extends \AHT\Blog\Controller\Adminhtml\Post

{

public function execute()

{

$resultRedirect = $this->resultRedirectFactory->create();

$ids = $this->getRequest()->getPost('ids');

if (!is\_array($ids)) {

$this->messageManager->addError(\_\_('Please select item(s).'));

} else {

try {

foreach ($ids as $id) {

$model = $this->\_objectManager->create('AHT\Blog\Model\Post')

->load($id)

->delete();

}

$this->messageManager->addSuccess(\_\_('Total of %1 record(s) were successfully deleted.', count($ids)));

} catch (\Exception $e) {

$this->messageManager->addError($e->getMessage());

}

}

return $resultRedirect->setPath('\*/\*/');

}

}

### Step 8: Create massStatus controller

<?php

namespace AHT\Blog\Controller\Adminhtml\Post;

class MassStatus extends \AHT\Blog\Controller\Adminhtml\Post

{

public function execute()

{

$resultRedirect = $this->resultRedirectFactory->create();

$ids = $this->getRequest()->getPost('ids');

if (!is\_array($ids)) {

$this->messageManager->addError(\_\_('Please select item(s).'));

} else {

try {

foreach ($ids as $id) {

$model = $this->\_objectManager->create('AHT\Blog\Model\Post')

->load($id)

->setStatus($this->getRequest()->getPost('status'))

->save();

}

$this->messageManager->addSuccess(\_\_('Total of %1 record(s) were successfully updated.', count($ids)));

} catch (\Exception $e) {

$this->messageManager->addError($e->getMessage());

}

}

return $resultRedirect->setPath('\*/\*/');

}

}