docs.zendframework.com

Introducing the Blog Module - tutorials

1. Docs »
2. MVC Tutorials »
3. In-Depth Tutorial »
4. Introducing the Blog Module

Now that we know about the basics of the zend-mvc skeleton application, let's continue and create our very own module. We will create a module named "Blog". This module will display a list of database entries that represent a single blog post. Each post will have three properties: `id` , `text` , and `title` . We will create forms to enter new posts into our database and to edit existing posts. Furthermore we will do so by using best-practices throughout the whole tutorial.

Writing a new Module

Let's start by creating a new folder under the `/module` directory called `Blog` , with the following stucture:

```
module/
   Blog/
       config/
       src/
       view/
```

To be recognized as a module by the ModuleManager, we need to do three things:

- Tell Composer how to autoload classes from our new module.
- Create a `Module` class in the `Blog` namespace.
- Notify the application of the new module.

Let's tell Composer about our new module. Open the `composer.json` file in the project root, and edit the `autoload` section to add a new PSR-4 entry for the `Blog` module; when you're done, it should read:

```
"autoload": {
  "psr-4": {
       "Application\\": "module/Application/src/",
       "Album\\": "module/Album/src/",
       "Blog\\": "module/Blog/src/"
  }
}
```

Once you're done, tell Composer to update its autoloading definitions:

```
$ composer dump-autoload
```

Next, we will create a `Module` class under the `Blog` namespace. Create the file `module/Blog/src/Module.php` with the following contents:

```
<?php
namespace Blog;

class Module
{
}
```

We now have a module that can be detected by the ModuleManager. Let's add this module to our application. Although our module doesn't do anything yet, just having the `Module.php` class allows it to be loaded by the ModuleManager. To do this, add an entry for `Blog` to the modules array inside `config/modules.config.php` :

```
<?php


return [

    'Application',
    'Album',
    'Blog',
];
```

If you refresh your application you should see no change at all (but also no errors).

At this point it's worth taking a step back to discuss what modules are for. In short, a module is an encapsulated set of features for your application. A module might add features to the application that you can see, like our `Blog` module; or it might provide background functionality for other modules in the application to use, such as interacting with a third party API.

Organizing your code into modules makes it easier for you to reuse functionality in other application, or to use modules written by the community.

Configuring the Module

The next thing we're going to do is add a route to our application so that our module can be accessed through the URL `localhost:8080/blog` . We do this by adding router configuration to our module, but first we need to let the `ModuleManager` know that our module has configuration that it needs to load.

This is done by adding a `getConfig()` method to the `Module` class that returns the configuration. (This method is defined in the `ConfigProviderInterface` , although explicitly implementing this interface in the module class is optional.) This method should return either an `array` or a `Traversable` object. Continue by editing `module/Blog/src/Module.php` :

```
class Module
{
    public function getConfig()
    {
        return [];
    }
}
```

With this, our module is now able to be configured. Configuration files can become quite big, though, and keeping everything inside the `getConfig()` method won't be optimal. To help keep our project organized, we're going to put our array configuration in a separate file. Go ahead and create this file at `module/Blog/config/module.config.php` :

```
<?php
return [];
```

Now rewrite the `getConfig()` function to include this newly created file instead of directly returning the array:

```
 <?php


public function getConfig()
{
    return include __DIR__ . '/../config/module.config.php';
}
```

Reload your application and you'll see that everything remains as it was. Next we add the new route to our configuration file:

```
namespace Blog;

use Zend\Router\Http\Literal;

return [

    'router' => [

        'routes' => [

            'blog' => [

                'type' => Literal::class,

                'options' => [

                    'route' => '/blog',


                    'defaults' => [
                        'controller' =>
Controller\ListController::class,
                        'action'     => 'index',
                    ],
                ],
            ],
        ],
    ],
];
```

We've now created a route called `post` that listens to the URL `localhost:8080/blog`. Whenever someone accesses this route, the `indexAction()` function of the class `Blog\Controller\ListController` will be executed. However, this controller does not exist yet, so if you reload the page you will see this error message:

```
 A 404 error occurred
Page not found.
The requested controller could not be mapped by routing.

Controller:
Blog\Controller\ListController(resolves to invalid controller class or
alias: Blog\Controller\ListController)
```

We now need to tell our module where to find this controller named `Blog\Controller\ListController`. To achieve this we have to add this key to the `controllers` configuration key inside your `module/Blog/config/module.config.php`.

```
 namespace Blog;

use Zend\ServiceManager\Factory\InvokableFactory;

return [
    'controllers' => [
        'factories' => [
            Controller\ListController::class => InvokableFactory::class,
        ],
    ],

];
```

This configuration defines a factory for the controller class `Blog\Controller\ListController`, using the zend-servicemanager `InvokableFactory` (which, internally, instantiates the class with no arguments). Reloading the page should then give you:

```
 Fatal error: Class 'Blog\Controller\ListController' not found in
{projectPath}/vendor/zendframework/zend-servicemanager/src/Factory
/InvokableFactory.php on line 32
```

This error tells us that the application knows what class to load, but was not able to autoload it. In our case, we've already setup autoloading, but have not yet defined the controller class!

Create the file `module/Blog/src/Controller/ListController.php` with the following contents:

```
 <?php
namespace Blog\Controller;

class ListController
{
}
```

Reloading the page now will finally result into a new screen. The new error message looks like this:

```
 A 404 error occurred
Page not found.
The requested controller was not dispatchable.

Controller:
Blog\Controller\List(resolves to invalid controller class or alias:
Blog\Controller\List)

Additional information:
Zend\ServiceManager\Exception\InvalidServiceException

File:
{projectPath}/vendor/zendframework/zend-mvc/src/Controller
/ControllerManager.php:{lineNumber}

Message:
Plugin of type "Blog\Controller\ListController" is invalid; must
implement Zend\Stdlib\DispatchableInterface
```

This happens because our controller must implement DispatchableInterface in order to be 'dispatched' (or run) by zend-mvc. zend-mvc provides a base controller implementation of it with AbstractActionController, which we are going to use. Let's modify our controller now:

```
namespace Blog\Controller;

use Zend\Mvc\Controller\AbstractActionController;

class ListController extends AbstractActionController
{
}
```

It's now time for another refresh of the site. You should now see a new error message:

```
 An error occurred

An error occurred during execution; please try again later.

Additional information:

Zend\View\Exception\RuntimeException

File:
{projectPath}/vendor/zendframework/zend-view/src/Renderer
/PhpRenderer.php:{lineNumber}

Message:
Zend\View\Renderer\PhpRenderer::render: Unable to render template
"blog/list/index"; resolver could not resolve to a file
```

Now the application tells you that a view template-file can not be rendered, which is to be expected as we've not created it yet. The application is expecting it to be at `module/Blog/view/blog/list/index.phtml`. Create this file and add some dummy content to it:

```
<h1>Blog\ListController::indexAction()</h1>
```

Before we continue let us quickly take a look at where we placed this file. Note that view files are found within the `/view` subdirectory, not `/src` as they are not PHP class files, but template files for rendering HTML. The path, however, deserves some explanation. First we have the lowercased namespace, followed by the lowercased controller name (without the suffix 'controller'), and lastly comes the name of the action that we are accessing (again without the suffix 'action'). As a templated string, you can think of it as: `view/{namespace}/{controller}/{action}.phtml`. This has become a community standard but you you have the freedom to specify custom paths if desired.

However creating this file alone is not enough and this brings as to the final topic of this part of the tutorial. We need to let the application know where to look for view files. We do this within our module's configuration file, `module.config.php` .

```
return [
    'controllers' => [  ],
    'router'      => [  ]
    'view_manager' => [
        'template_path_stack' => [
            __DIR__ . '/../view',
        ],
    ],
];
```

The above configuration tells the application that the folder `module/Blog/view/` has view files in it that match the above described default scheme. It is important to note that with this you can not only ship view files for your module, but you can also overwrite view files from other modules.

Reload your site now. Finally we are at a point where we see something different than an error being displayed! Congratulations, not only have you created a simple "Hello World" style module, you also learned about many error messages and their causes. If we didn't exhaust you too much, continue with our tutorial, and let's create a module that actually does something.