docs.zendframework.com

Modules - tutorials

1. Docs »
2. MVC Tutorials »
3. Getting Started with Zend Framework »
4. Modules

zend-mvc uses a module system to organise your main application-specific code within each module. The `Application` module provided by the skeleton is used to provide bootstrapping, error, and routing configuration to the whole application. It is usually used to provide application level controllers for the home page of an application, but we are not going to use the default one provided in this tutorial as we want our album list to be the home page, which will live in our own module.

We are going to put all our code into the `Album` module which will contain our controllers, models, forms and views, along with configuration. We'll also tweak the `Application` module as required.

Let's start with the directories required.

Setting up the Album module

Start by creating a directory called `Album` under `module` with the following subdirectories to hold the module's files:

```
zf-tutorial/
    /module
        /Album
            /config
            /src
                /Controller
                /Form
                /Model
            /view
                /album
```

```
                    /album
```

The `Album` module has separate directories for the different types of files we will have. The PHP files that contain classes within the `Album` namespace live in the `src/` directory. The view directory also has a sub-folder called `album` for our module's view scripts.

In order to load and configure a module, Zend Framework provides a `ModuleManager`. This will look for a `Module` class in the specified module namespace (i.e., `Album`); in the case of our new module, that means the class `Album\Module`, which will be found in `module/Album/src/Module.php`.

Let's create that file now, with the following contents:

```
namespace Album;

use Zend\ModuleManager\Feature\ConfigProviderInterface;

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

The `ModuleManager` will call `getConfig()` automatically for us.

Autoloading

While Zend Framework provides autoloading capabilities via its zend-loader component, we recommend using Composer's autoloading capabilities. As such, we need to inform Composer of our new namespace, and where its files live.

Open `composer.json` in your project root, and look for the `autoload` section; it should look like the following by default:

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

We'll now add our new module to the list, so it now reads:

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

Once you've made that change, run the following to ensure Composer updates its autoloading rules:

```
$ composer dump-autoload
```

Configuration

Having registered the autoloader, let's have a quick look at the `getConfig()` method in `Album\Module`. This method loads the `config/module.config.php` file under the module's root directory.

Create a file called `module.config.php` under `zf-tutorial/module/Album/config/`:

```
namespace Album;

use Zend\ServiceManager\Factory\InvokableFactory;

return [
    'controllers' => [
        'factories' => [
            Controller\AlbumController::class =>
InvokableFactory::class,
        ],
    ],
    'view_manager' => [
        'template_path_stack' => [
            'album' => __DIR__ . '/../view',
        ],
    ],
];
```

The config information is passed to the relevant components by the `ServiceManager`. We need two initial sections: `controllers` and `view_manager`. The controllers section provides a list of all the controllers provided by the module. We will need one controller, `AlbumController`; we'll reference it by its fully qualified class name, and use the zend-servicemanager `InvokableFactory` to create instances of it.

Within the `view_manager` section, we add our view directory to the `TemplatePathStack` configuration. This will allow it to find the view scripts for the `Album` module that are stored in our `view/` directory.

Informing the application about our new module

We now need to tell the `ModuleManager` that this new module exists. This is done in the application's `config/modules.config.php` file which is provided by the skeleton application. Update this file so that the array it returns contains the `Album` module as well, so the file now looks like this:

(Changes required are highlighted using comments; original comments from the file are omitted for brevity.)

```
return [
    'Zend\Form',
    'Zend\Db',
    'Zend\Router',
    'Zend\Validator',
    'Application',
    'Album',
];
```

As you can see, we have added our `Album` module into the list of modules after the `Application` module.

We have now set up the module ready for putting our custom code into it.