docs.zendframework.com

Advanced Configuration - tutorials

1. Docs »
2. MVC Tutorials »
3. Advanced Configuration

Configuration of zend-mvc applications happens in several steps:

- Initial configuration is passed to the `Application` instance and used to seed the `ModuleManager` and `ServiceManager`. In this tutorial, we will call this configuration **system configuration**.
- The `ModuleManager`'s `ConfigListener` aggregates configuration and merges it while modules are being loaded. In this tutorial, we will call this configuration **application configuration**.
- Once configuration is aggregated from all modules, the `ConfigListener` will also merge application configuration globbed in specified directories (typically `config/autoload/`).
- Finally, immediately prior to the merged application configuration being passed to the `ServiceManager`, it is passed to a special `EVENT_MERGE_CONFIG` event to allow further modification.

In this tutorial, we'll look at the exact sequence, and how you can tie into it.

System configuration

To begin module loading, we have to tell the `Application` instance about the available modules and where they live, optionally provide some information to the default module listeners (e.g., where application configuration lives, and what files to load; whether to cache merged configuration, and where; etc.), and optionally seed the `ServiceManager`. For purposes of this tutorial we will call this the **system configuration**.

When using the skeleton application, the **system configuration** is by default in `config/application.config.php`. The defaults look like this:

```
return [
    'modules' => require __DIR__ . '/modules.config.php',

    'module_listener_options' => [


        'module_paths' => [
            './module',
            './vendor',
        ],
```

```
        'config_glob_paths' => [
            realpath(__DIR__) . '/autoload/{{,*.}global,{,*.}local}.php',
        ],




        'config_cache_enabled' => true,


        'config_cache_key' => 'application.config.cache',




        'module_map_cache_enabled' => true,


        'module_map_cache_key' => 'application.module.cache',


        'cache_dir' => 'data/cache/',




    ],
```

```
];
```

The system configuration is for the bits and pieces related to the MVC that run before your application is ready. The configuration is usually brief, and quite minimal.

Also, system configuration is used *immediately*, and is not merged with any other configuration — which means, with the exception of the values under the `service_manager` key, it cannot be overridden by a module.

This leads us to our first trick: how do you provide environment-specific system configuration?

Environment-specific system configuration

What happens when you want to change the set of modules you use based on the environment? Or if the configuration caching should be enabled based on environment?

It is for this reason that the default system configuration we provide in the skeleton application is in PHP; providing it in PHP means you can programmatically manipulate it.

As an example, let's make the following requirements:

- We want to use the `ZendDeveloperTools` module in development only.
- We want to have configuration caching on in production only.

zfcampus/zf-development-mode provides a concise and conventions-based approach to switching between specifically production and development. The package is installed by default with version 3+ skeletons, and can be installed with existing v2 skeletons using the following:

```
$ composer require zfcampus/zf-development-mode
```

The approach it takes is as follows:

- The user provides production settings in `config/application.config.php`.
- The user provides development settings in `config/development.config.php.dist` to override bootstrap-level settings such as modules and configuration caching, and optionally also in `config/autoload/development.local.php.dist` (to override application settings).
- The bootstrap script (`public/index.php`) checks for `config/development.config.php`, and, if found, merges its configuration with the application configuration prior to configuring the `Application` instance.

When you execute:

```
$ ./vendor/bin/zf-development-mode enable
```

The `.dist` files are copied to versions removing the suffix; doing so ensures they will then be used when invoking the application.

As such, to accomplish our goals, we will do the following:

- In `config/development.config.php.dist`, add `ZendDeveloperTools` to the list of modules:

```
'modules' => [
    'ZendDeveloperTools',
],
```

- Also in `config/development.config.php.dist`, we will disable config caching:

```
'config_cache_enable' => false,
```

- In `config/application.config.php`, we will enable config caching:

```
'config_cache_enable' => true,
```

Enabling development mode now enables the selected module, and disables configuration caching; disabling development mode enables configuration caching. (Also, either operation clears the configuration cache.)

If you require additional environments, you can extend zf-development-mode to address them using the same workflow.

Environment-specific application configuration

Sometimes you want to change application configuration to load things such as database adapters, log writers, cache adapters, and more based on the environment. These are typically managed in the service manager, and may be defined by modules. You can override them at the application level via `Zend\ModuleManager\Listener\ConfigListener`, by specifying a glob path in the **system configuration** — the `module_listener_options.config_glob_paths` key from the previous examples.

The default value for this is `config/autoload/{{,*.}global,{,*.}local}.php`. What this means is that it will look for **application configuration** files in the `config/autoload` directory, in the following order:

- `global.php`
- `*.global.php`
- `local.php`
- `*.local.php`

This allows you to define application-level defaults in "global" configuration files, which you would then commit to your version control system, and environment-specific overrides in your "local" configuration files, which you would *omit* from version control.

Additional glob patterns for development mode

When using zf-development-mode, as detailed in the previous section, the shipped `config/development.config.php.dist` file provides an additional glob pattern for specifying development configuration:

- `config/autoload/{,*.}{global,local}-development.php`

This will match files such as:

- database.global-development.php
- database.local-development.php

These will only be considered when development mode is enabled!

This is a great solution for development, as it allows you to specify alternate configuration that's specific to your development environment without worrying about accidently deploying it. However, what if you have more environments — such as a "testing" or "staging" environment — and they each have their own specific overrides?

To accomplish this, we'll provide an *environment variable* via our web server configuration, `APP_ENV`. In Apache, you'd put a directive like the following in either your system-wide apache.conf or httpd.conf, or in the definition for your virtual host; alternately, it can be placed in an .htaccess file.

```
SetEnv "APP_ENV" "development"
```

For other web servers, consult the web server documentation to determine how to set environment variables.

To simplify matters, we'll assume the environment is "production" if no environment variable is present.

With that in place, We can alter the glob path in the system configuration slightly:

```
 'config_glob_paths' => [
    realpath(__DIR__) . sprintf('config/autoload/{,*.}{global,%s,local}.php', getenv('APP_ENV')
?: 'production')
],
```

The above will allow you to define an additional set of application configuration files per environment; furthermore, these will be loaded *only* if that environment is detected!

As an example, consider the following tree of configuration files:

```
 config/
    autoload/
        global.php
        local.php
        users.development.php
        users.testing.php
        users.local.php
```

If `$env` evaluates to `testing`, then the following files will be merged, in the following order:

```
 global.php
users.testing.php
local.php
users.local.php
```

Note that `users.development.php` is not loaded — this is because it will not match the glob pattern!

Also, because of the order in which they are loaded, you can predict which values will overwrite the others, allowing you to both selectively overwrite as well as debug later.

> **Order of config merging**
>
> The files under `config/autoload/` are merged *after* your module configuration, detailed in next section. We have detailed it here, however, as setting up the **application configuration** glob path happens within the **system configuration** (`config/application.config.php`).

Module Configuration

One responsibility of modules is to provide their own configuration to the application. Modules have two general mechanisms for doing this.

First, modules that either implement `Zend\ModuleManager\Feature\ConfigProviderInterface` and/or a `getConfig()` method can return their configuration. The default, recommended implementation of the `getConfig()` method is:

```
 public function getConfig()
{
```

```
    return include __DIR__ . '/config/module.config.php';
}
```

where `module.config.php` returns a PHP array. From that PHP array you can provide general configuration as well as configuration for all the available `Manager` classes provided by the ServiceManager. Please refer to the Configuration mapping table to see which configuration key is used for each specific `Manager` .

Second, modules can implement a number of interfaces and/or methods related to specific service manager or plugin manager configuration. You will find an overview of all interfaces and their matching Module Configuration functions inside the Configuration mapping table.

Most interfaces are in the `Zend\ModuleManager\Feature` namespace (some have moved to the individual components), and each is expected to return an array of configuration for a service manager, as denoted in the section on default service configuration.

Configuration mapping table

Manager name	Interface name	Module method name	Config ke
`ControllerPluginManager`	`ControllerPluginProviderInterface`	`getControllerPluginConfig()`	`controller`
`ControllerManager`	`ControllerProviderInterface`	`getControllerConfig()`	`controller`
`FilterManager`	`FilterProviderInterface`	`getFilterConfig()`	`filters`
`FormElementManager`	`FormElementProviderInterface`	`getFormElementConfig()`	`form_eleme`
`HydratorManager`	`HydratorProviderInterface`	`getHydratorConfig()`	`hydrators`
`InputFilterManager`	`InputFilterProviderInterface`	`getInputFilterConfig()`	`input_filt`
`RoutePluginManager`	`RouteProviderInterface`	`getRouteConfig()`	`route_mana`
`SerializerAdapterManager`	`SerializerProviderInterface`	`getSerializerConfig()`	`serializer`
`ServiceLocator`	`ServiceProviderInterface`	`getServiceConfig()`	`service_ma`
`ValidatorManager`	`ValidatorProviderInterface`	`getValidatorConfig()`	`validators`
`ViewHelperManager`	`ViewHelperProviderInterface`	`getViewHelperConfig()`	`view_helpe`
`LogProcessorManager`	`LogProcessorProviderInterface`	`getLogProcessorConfig`	`log_proces`
`LogWriterManager`	`LogWriterProviderInterface`	`getLogWriterConfig`	`log_writer`

Configuration Priority

Considering that you may have service configuration in your module configuration file, what has precedence?

The order in which they are merged is:

- configuration returned by the various service configuration methods in a module class
- configuration returned by `getConfig()`

In other words, your `getConfig()` wins over the various service configuration methods. Additionally, and of particular note: the configuration returned from those methods will *not* be cached.

> **Use cases for service configuration methods**
>
> Use the various service configuration methods when you need to define closures or instance callbacks for factories, abstract factories, and initializers. This prevents caching problems, and also allows you to write your configuration files in other markup formats.

Manipulating merged configuration

Occasionally you will want to not just override an application configuration key, but actually remove it. Since merging will not remove keys, how can

you handle this?

`Zend\ModuleManager\Listener\ConfigListener` triggers a special event, `Zend\ModuleManager\ModuleEvent::EVENT_MERGE_CONFIG`, after merging all configuration, but prior to it being passed to the `ServiceManager`. By listening to this event, you can inspect the merged configuration and manipulate it.

The `ConfigListener` itself listens to the event at priority 1000 (i.e., very high), which is when the configuration is merged. You can tie into this to modify the merged configuration from your module, via the `init()` method.

```
namespace Foo;

use Zend\ModuleManager\ModuleEvent;
use Zend\ModuleManager\ModuleManager;

class Module
{
    public function init(ModuleManager $moduleManager)
    {
        $events = $moduleManager->getEventManager();



        $events->attach(ModuleEvent::EVENT_MERGE_CONFIG, [$this, 'onMergeConfig']);
    }

    public function onMergeConfig(ModuleEvent $e)
    {
        $configListener = $e->getConfigListener();
        $config         = $configListener->getMergedConfig(false);


        if (isset($config['some_key'])) {
            unset($config['some_key']);
        }


        $configListener->setMergedConfig($config);
    }
}
```

At this point, the merged application configuration will no longer contain the key `some_key`.

> **Cached configuration and merging**
>
> If a cached config is used by the `ModuleManager`, the `EVENT_MERGE_CONFIG` event will not be triggered. However, typically that means that what is cached will be what was originally manipulated by your listener.

Configuration merging workflow

To cap off the tutorial, let's review how and when configuration is defined and merged.

- **System configuration**
- Defined in `config/application.config.php`

- No merging occurs
- Allows manipulation programmatically, which allows the ability to:
  - Alter flags based on computed values
  - Alter the configuration glob path based on computed values
  - Configuration is passed to the `Application` instance, and then the `ModuleManager` in order to initialize the system.

- **Application configuration**
- The `ModuleManager` loops through each module class in the order defined in the **system configuration**
  - Service configuration defined in `Module` class methods is aggregated
  - Configuration returned by `Module::getConfig()` is aggregated
  - Files detected from the **service configuration** `config_glob_paths` setting are merged, based on the order they resolve in the glob path.
  - `ConfigListener` triggers `EVENT_MERGE_CONFIG` :
  - `ConfigListener` merges configuration
  - Any other event listeners manipulate the configuration
  - Merged configuration is finally passed to the `ServiceManager`
