Zend Framework

tutorialspoint

SIMPLY EASY LEARNING

www.tutorialspoint.com

 https://www.facebook.com/tutorialspointindia  https://twitter.com/tutorialspoint

About the Tutorial

Zend is an open source PHP framework. It is pure object-oriented and built around the MVC design pattern. Zend framework contains collection of PHP packages which can be used to develop web applications and services. Zend was started by Andi Gutmans and Zeev Suraski.

This tutorial will give you a quick introduction to Zend Framework and make you comfortable with its various components.

Audience

This tutorial has been prepared for professionals who are aspiring to make a career in Zend framework. This will give you enough understanding on how to create and develop a website using Zend.

Prerequisites

Before proceeding with the various types of components given in this tutorial, it is being assumed that the readers are already aware about what a Framework is. In addition to this, it will also be very helpful if you have sound knowledge on HTML, PHP and the OOPS concepts.

Copyright & Disclaimer

Table of Contents

1. Zend Framework – Introduction

A PHP Web Framework is a collection of classes which helps to develop a web application. Zend is one of the most popular PHP framework. It is an **open-source MVC framework** for rapidly developing, modern web applications. Zend Framework has several loosely coupled components, so it is referred to as "Component Library". Zend Framework provides any PHP stack and Zend server to run Zend framework applications.

Zend Studio is an IDE that includes features to integrate with Zend Framework. It provides MVC view and code generation. The current Zend framework 3.0 includes new components such as JSON RPC server, a XML to JSON converter, PSR-7 functionality, and compatibility with PHP 7.

Zend Framework 2 is an open source framework for developing web applications and services using PHP 5.3+. Zend Framework 2 uses 100% object oriented code and utilizes most of the new features of PHP 5.3, namely **Namespaces**, **Lambda Functions** and **Closures**.

Zend Framework 2 evolved from Zend Framework 1, a successful PHP framework with over 15 million downloads. Zend Server has a free community version and a commercial version.

Zend Framework Features

Some of the salient features of Zend Framework is as follows:

- Pure object oriented web application framework
- Advanced MVC implementation
- Supports multi databases including PostgreSQL, SQLite etc.,
- Simple cloud API
- Session management
- Data encryption
- Flexible URI Routing
- Zend provides RESTful API development support.
- Code reusable and easier to maintain.

Why Zend Framework?

What makes the Zend Framework one of the premier frameworks used by PHP developers is that – it provides clean and stable code complete with intellectual property rights. It also makes programming easier. It is fast, easy to learn and convenient framework. Zend supports strong cryptography tools and password hashing techniques.

Zend Goals

Following are the goals of the Zend Framework.

- Flexibility
- Simple and productive
- Compatibility
- Extensibility – Programmer can easily extend all the framework classes.
- Portability – Supports multiple environments

Zend Applications

The following popular products are developed by using the Zend Framework.

- McAfee Company website
- IBM Company website
- Magento - one of the popular shopping cart website.

Advantages of Zend Framework

Some of the advantages of the Zend Framework are listed below.

- **Loosely Coupled –** Zend provides the option to delete modules or components which we don't need in the application.
- **Performance –** Zend Framework is highly optimized for performance. Zend Framework 3 is 4x faster than its previous version.
- **Security –** Framework supports industry standard encryption.
- **Testing –** PHPUnit is integrated with Zend so you can easily test the framework.

In the next chapter, we will learn how to install the Zend Framework.

2. Zend Framework – Installation

To install the Zend Framework, we must first install the Composer and the latest version of PHP as shown in the following steps.

- **Install Composer:** Zend uses Composer for managing its dependencies, so make sure you have the Composer installed on your machine. If the Composer is not installed, then visit the official website of Composer and install it.

- **Install the latest version of PHP:** To get the maximum benefit of Zend Framework, install the latest version of PHP. The minimum required version for the Zend Framework 3 is PHP 5.6 or later.

Install Zend Framework

Zend Framework can be installed in two ways. They are as follows:

- Manual installation
- Composer based installation

Let us discuss both these installations in detail.

Manual Installation

Download the latest version of Zend Framework by visiting the following link – https://framework.zend.com/downloads/archives

Extract the content of the downloaded archive file to the folder you would like to keep it. Once you have a copy of Zend Framework available in your local machine, your Zend Framework based web application can access the framework classes. Though there are several ways to achieve this, your PHP **include_path** needs to contain the path to the Zend Framework classes under the /library directory in the distribution. This method applies to Zend Framework version 2.4 and earlier only.

Composer Based Installation

To easily install the Zend Framework, use the Composer tool. This is the preferred method to install the latest version of Zend Framework. To install all the components of the Zend Framework, use the following Composer command –

```
$ composer require zendframework/zendframework
```

Each Zend Framework module / component can be installed individually as well. For example, to install the **MVC component** of the Zend Framework, use the following **composer** command –

```
$ composer require zendframework/zend-mvc
```

3. Zend Framework – Skeleton Application

Let us create a skeleton application using the Zend Framework MVC layer and module systems.

Installation using Composer

The easiest way to create a new Zend Framework project is to use a Composer. It is defined as below:

```
$ cd /path/to/install
$ composer create-project -n -sdev zendframework/skeleton-application myapp
```

You would see the following result on your screen:

```
Installing zendframework/skeleton-application (dev-master
941da45b407e4f09e264f000fb537928badb96ed)
  - Installing zendframework/skeleton-application (dev-master master)
    Cloning master

Created project in myapp
Loading composer repositories with package information
Installing dependencies (including require-dev) from lock file
  - Installing zendframework/zend-component-installer (0.3.0)
    Loading from cache

  - Installing zendframework/zend-stdlib (3.0.1)
    Loading from cache

  - Installing zendframework/zend-config (2.6.0)
    Loading from cache

  - Installing zendframework/zend-loader (2.5.1)
    Loading from cache

  - Installing zendframework/zend-eventmanager (3.0.1)
```

```
    Loading from cache
  - Installing zendframework/zend-view (2.8.0)
    Loading from cache

  - Installing container-interop/container-interop (1.1.0)
    Loading from cache

  - Installing zendframework/zend-servicemanager (3.1.0)
    Loading from cache

  - Installing zendframework/zend-validator (2.8.1)
    Loading from cache

  - Installing zendframework/zend-escaper (2.5.1)
    Loading from cache

  - Installing zendframework/zend-uri (2.5.2)
    Loading from cache

  - Installing zendframework/zend-http (2.5.4)
    Loading from cache

  - Installing zendframework/zend-router (3.0.2)
    Loading from cache

  - Installing zendframework/zend-modulemanager (2.7.2)
    Loading from cache

  - Installing zendframework/zend-mvc (3.0.1)
    Loading from cache

  - Installing zendframework/zend-skeleton-installer (0.1.3)
    Loading from cache

  - Installing zfcampus/zf-development-mode (3.0.0)
```


```
    Loading from cache
zendframework/zend-config suggests installing zendframework/zend-filter
(Zend\Filter component)
zendframework/zend-config suggests installing zendframework/zend-i18n (Zend\I18n
component)
zendframework/zend-config suggests installing zendframework/zend-json (Zend\Json
to use the Json reader or writer classes)
zendframework/zend-view suggests installing zendframework/zend-authentication
(Zend\Authentication component)
zendframework/zend-view suggests installing zendframework/zend-feed (Zend\Feed
component)
zendframework/zend-view suggests installing zendframework/zend-filter (Zend\Filter
component)
zendframework/zend-view suggests installing zendframework/zend-i18n (Zend\I18n
component)
zendframework/zend-view suggests installing zendframework/zend-json (Zend\Json
component)
zendframework/zend-view suggests installing zendframework/zend-navigation
(Zend\Navigation component)
zendframework/zend-view suggests installing zendframework/zend-paginator
(Zend\Paginator component)
zendframework/zend-view suggests installing zendframework/zend-permissions-acl
(Zend\Permissions\Acl component)
zendframework/zend-servicemanager suggests installing ocramius/proxy-manager
(ProxyManager 1.* to handle lazy initialization of services)
zendframework/zend-validator suggests installing zendframework/zend-db (Zend\Db
component)
zendframework/zend-validator suggests installing zendframework/zend-filter
(Zend\Filter component, required by the Digits validator)
zendframework/zend-validator suggests installing zendframework/zend-i18n
(Zend\I18n component to allow translation of validation error messages as well as
to use the various Date validators)
zendframework/zend-validator suggests installing zendframework/zend-i18n-resources
(Translations of validator messages)
zendframework/zend-validator suggests installing zendframework/zend-math
(Zend\Math component)
zendframework/zend-validator suggests installing zendframework/zend-session
(Zend\Session component)
zendframework/zend-router suggests installing zendframework/zend-i18n (^2.6, if
defining translatable HTTP path segments)
```


```
zendframework/zend-modulemanager suggests installing zendframework/zend-console
(Zend\Console component)

zendframework/zend-mvc suggests installing zendframework/zend-json ((^2.6.1 ||
^3.0) To auto-deserialize JSON body content in AbstractRestfulController
extensions, when json_decode is unavailable)

zendframework/zend-mvc suggests installing zendframework/zend-mvc-console (zend-
mvc-console provides the ability to expose zend-mvc as a console application)

zendframework/zend-mvc suggests installing zendframework/zend-mvc-i18n (zend-mvc-
i18n provides integration with zend-i18n, including a translation bridge and
translatable route segments)

zendframework/zend-mvc suggests installing zendframework/zend-mvc-plugin-fileprg
(To provide Post/Redirect/Get functionality around forms that container file
uploads)

zendframework/zend-mvc suggests installing zendframework/zend-mvc-plugin-
flashmessenger (To provide flash messaging capabilities between requests)

zendframework/zend-mvc suggests installing zendframework/zend-mvc-plugin-identity
(To access the authenticated identity (per zend-authentication) in controllers)

zendframework/zend-mvc suggests installing zendframework/zend-mvc-plugin-prg (To
provide Post/Redirect/Get functionality within controllers)

zendframework/zend-mvc suggests installing zendframework/zend-psr7bridge ((^0.2)
To consume PSR-7 middleware within the MVC workflow)

zendframework/zend-mvc suggests installing zendframework/zend-servicemanager-di
(zend-servicemanager-di provides utilities for integrating zend-di and zend-
servicemanager in your zend-mvc application)

Generating autoload files

    Removing optional packages from composer.json

    Updating composer.json

Removing zendframework/zend-skeleton-installer...

  - Removing zendframework/zend-skeleton-installer (0.1.3)

    Removed plugin zendframework/zend-skeleton-installer.

    Removing from composer.json

    Complete!

> zf-development-mode enable

You are now in development mode.
```

Now that the application is installed, you can test it out immediately using the **PHP's built-in web server**:

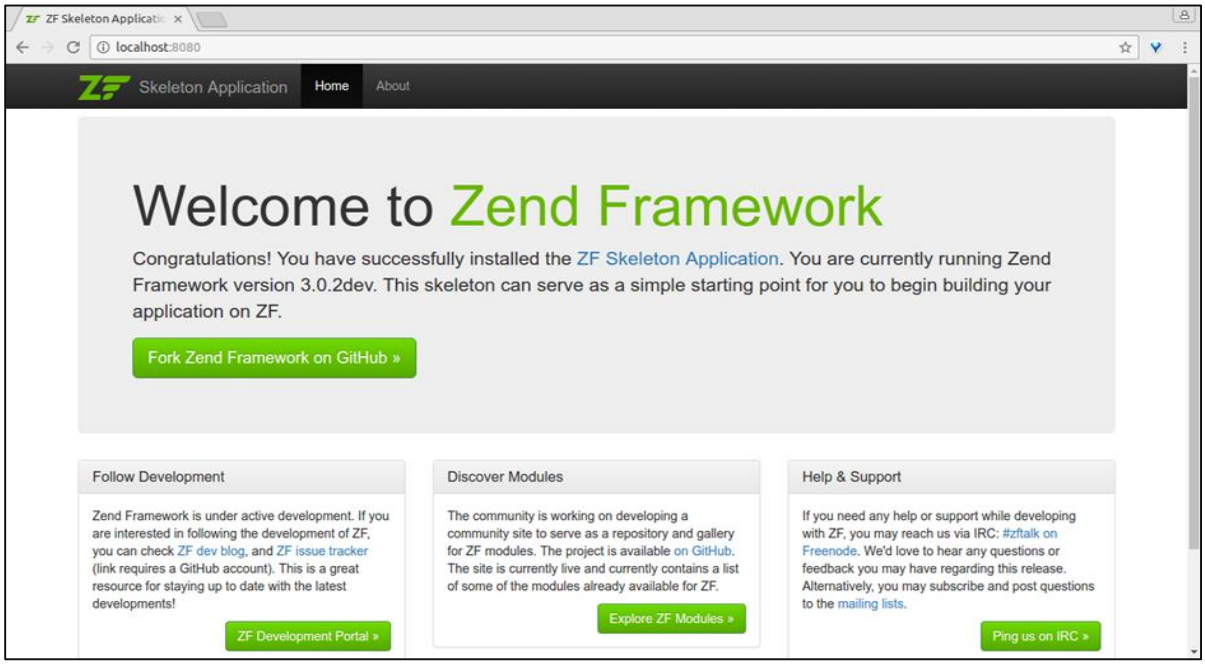
```
$ cd path/to/install/myapp

$ composer serve
```

Then you would see the following response:

```
> php -S 0.0.0.0:8080 -t public/ public/index.php
```

This will start the PHP built-in CLI server on port 8080. Once the development server is started, you can visit the site at (http://localhost:8080/). The built-in CLI server is for development only.



Unit Tests

To run the skeleton unit tests, type the following command in your terminal.

```
$ composer require --dev zendframework/zend-test
```

It will produce the following response:

```
Using version ^3.0 for zendframework/zend-test
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
  - Installing zendframework/zend-dom (2.6.0)
    Loading from cache

  - Installing zendframework/zend-console (2.6.0)
```

```
  Loading from cache

- Installing sebastian/version (2.0.1)
  Loading from cache
- Installing symfony/yaml (v3.2.1)
  Downloading: 100%

- Installing sebastian/resource-operations (1.0.0)
  Loading from cache

- Installing sebastian/recursion-context (2.0.0)
  Loading from cache

- Installing sebastian/object-enumerator (2.0.0)
  Loading from cache

- Installing sebastian/global-state (1.1.1)
  Loading from cache

- Installing sebastian/exporter (2.0.0)
  Loading from cache

- Installing sebastian/environment (2.0.0)
  Loading from cache

- Installing sebastian/diff (1.4.1)
  Loading from cache

- Installing sebastian/comparator (1.2.2)
  Loading from cache

- Installing phpunit/php-text-template (1.2.1)
  Loading from cache

- Installing doctrine/instantiator (1.0.5)
```

```
  Loading from cache

- Installing phpunit/phpunit-mock-objects (3.4.3)
  Downloading: 100%

- Installing phpunit/php-timer (1.0.8)
  Loading from cache

- Installing phpunit/php-file-iterator (1.4.2)
  Loading from cache

- Installing sebastian/code-unit-reverse-lookup (1.0.0)
  Loading from cache

- Installing phpunit/php-token-stream (1.4.9)
  Loading from cache

- Installing phpunit/php-code-coverage (4.0.4)
  Downloading: 100%

- Installing webmozart/assert (1.2.0)
  Loading from cache

- Installing phpdocumentor/reflection-common (1.0)
  Loading from cache

- Installing phpdocumentor/type-resolver (0.2.1)
  Loading from cache

- Installing phpdocumentor/reflection-docblock (3.1.1)
  Loading from cache

- Installing phpspec/prophecy (v1.6.2)
  Loading from cache
```

```
  - Installing myclabs/deep-copy (1.5.5)
    Loading from cache

  - Installing phpunit/phpunit (5.7.4)
    Downloading: 100%
  - Installing zendframework/zend-test (3.0.2)
    Loading from cache
zendframework/zend-console suggests installing zendframework/zend-filter (To
support DefaultRouteMatcher usage)
symfony/yaml suggests installing symfony/console (For validating YAML files using
the lint command)
sebastian/global-state suggests installing ext-uopz (*)
phpunit/phpunit-mock-objects suggests installing ext-soap (*)
phpunit/php-code-coverage suggests installing ext-xdebug (>=2.4.0)
phpunit/phpunit suggests installing phpunit/php-invoker (~1.1)
phpunit/phpunit suggests installing ext-xdebug (*)
zendframework/zend-test suggests installing zendframework/zend-mvc-console
(^1.1.8, to test MVC <-> console integration)
Writing lock file
Generating autoload files
```

Now the testing support is enabled so you can run the test using the following command.

```
$ ./vendor/bin/phpunit
```

Apache Web Server

Hosting the Zend Framework based application in the production environment is very simple and straight-forward. Just create a **VirtualHost** in the Apache configuration file and point the **DocumentRoot** to the **Public** folder of the Zend Framework application.

A sample configuration (myapp) is given below:

```
<VirtualHost *:80>
    ServerName myapp.localhost
    DocumentRoot /path/to/install/myapp/public
    <Directory /path/to/install/myapp/public>
        DirectoryIndex index.php
        AllowOverride All
```

```
        Order allow,deny
        Allow from all
        <IfModule mod_authz_core.c>
            Require all granted
        </IfModule>
    </Directory>
</VirtualHost>
```

4. Zend Framework – MVC Architecture

Before proceeding with this chapter, let us have a brief understanding of MVC. A **Model View Controller** is a software approach that separates the application logic from the presentation. In practice, it permits the webpages to contain minimal PHP scripting since the presentation is separate from it.

The short description of the MVC Components is as follows

- **Model:** Model represents the structure of the application data. Typically, model classes contain functions that helps to **retrieve**, **insert** and **update business data** in the back-end database (MySQL, PostgreSQL, etc.).
- **View:** View is the presentation layer of the MVC Application. It gets the models data through the Controller and display it as needed. It is loosely coupled to the **Controller** and the **Model** and so, it can be changed without affecting either the Model and the Controller.
- **Controller:** The Controller is the main component of the MVC architecture. Every request first hits the controller. In other words, the controller processes all the request and serves as an intermediary between the Model, View, and any other resources needed to **process the HTTP request** and to generate the response.

In the next chapter, we will understand the different concepts of the Zend Framework.

5. Zend Framework – Concepts

Zend Framework is a collection of 60+ components. They are loosely connected with each other. They can be used as both stand-alone component as well as a group of components working as a single unit.

Zend Framework provides three most important components, which are –

- zend-servicemanager
- zend-eventmanager and
- zend-modulemanager.

They provide Zend components the ability to integrate with other components efficiently.

- **Event Manager –** It gives the ability to create event based programming. This helps to create, inject and manage new events.
- **Service Manager –** It gives the ability to consume any services (PHP classes) from anywhere with a little effort.
- **Module Manager –** Ability to convert a collection of PHP classes with similar functionality into a single unit called as a **module**. The newly created modules can be used, maintained and configured as a single unit.

We will cover these concepts in detail in the subsequent chapters.

6. Zend Framework – Service Manager

The Zend Framework includes a powerful service locator pattern implementation called **zend-servicemanager**. Zend framework extensively uses the service manager for all its functionalities. The Service Manager provides a high-level abstraction for the Zend Framework. It also integrates nicely with all the other components of the Zend Framework.

Install Service Manager

The Service Manager component can be installed using the **composer** tool.

```
composer require zendframework/zend-servicemanager
```

Example

First, all the services need to be registered into the service manager. Once the services are registered into the server manager system, it can be accessed at any time with minimal efforts. The service manager provides a lot of options to register the service. A simple example is as follows:

```
use Zend\ServiceManager\ServiceManager;
use Zend\ServiceManager\Factory\InvokableFactory;
use stdClass;

$serviceManager = new ServiceManager([
    'factories' => [
        stdClass::class => InvokableFactory::class,
    ],
]);
```

The above code registers the **stdClass** into the system using the **Factory** option. Now, we can get an instance of the stdClass at any time using the ***get()*** method of the service manager as shown below.

```
use Zend\ServiceManager\ServiceManager;

$object = $serviceManager->get(stdClass::class);
```

The get() method shares the retrieved object and so, the object returned by calling the get() method multiple times is one and the same instance. To get a different instance every time, the service manager provides another method, which is the **build()** method.

```
use Zend\ServiceManager\ServiceManager;


$a = $serviceManager->build(stdClass::class);
$b = $serviceManager->build(stdClass::class);
```

Service Manager Registration

The service manager provides a set of methods to register a component. Some of the most important methods are as given below:

- Factory method
- Abstract factory method
- Initializer method
- Delegator factory method

We will discuss each of these in detail in the upcoming chapters.

Factory Method

A factory is basically any callable or any class that implements the **FactoryInterface** (Zend\ServiceManager\Factory\FactoryInterface).

The FactoryInterface has a single method:

```
public function __invoke(ContainerInterface $container, $requestedName, array
$options = null)
```

The arguments details of the FactoryInterface is as follows:

- **container (ContainerInterface) –** It is the base interface of the ServiceManager. It provides an option to get other services.
- **requestedName –** It is the service name.
- **options –** It gives additional options needed for the service.

Let us create a simple class implementing the FactoryInterface and see how to register the class.

Class: Test - Object to be Retrieved

```
use stdClass;
```

```
class Test
{
    public function __construct(stdClass $sc)
    {
        // use $sc
    }
}
```

The **Test** class depends on the stdClass.

Class: TestFactory - Class to Initialize Test Object

```
class TestFactory implements FactoryInterface
{
    public function __invoke(ContainerInterface $container, $requestedName, array
$options = null)
    {
        $dep = $container->get(stdClass::class);
        return new Test($dep);
    }
}
```

The TestFactory uses a container to retrieve the stdClass, creates the instance of the Test class, and returns it.

Registration and Usage of the Zend Framework

Let us now understand how to register and use the Zend Framework.

```
serviceManager $sc = new ServiceManager([
    'factories' => [
        stdClass::class => InvokableFactory::class,
        Test::class => TestFactory::class
    ]
]);
```

```
$test = $sc->get(Test::class);
```

The service manager provides a special factory called **InvokableFactory** to retrieve any class which has no dependency. For example, the **stdClass** can be configured using the InvokableFactory since the stdClass does not depend on any other class.

```
serviceManager $sc = new ServiceManager([
    'factories' => [
        stdClass::class => InvokableFactory::class
    ]
]);

$stdC = $sc->get(stdClass::class);
```

Another way to retrieve an object without implementing the **FactoryInterfac** or using the **InvokableFactory** is using the inline method as given below.

```
$serviceManager = new ServiceManager([
    'factories' => [
        stdClass::class => InvokableFactory::class,
        Test::class => function(ContainerInterface $container, $requestedName) {
            $dep= $container->get(stdClass::class);
            return new Test($dep);
        },
    ],
]);
```

Abstract Factory Method

Sometimes, we may need to create objects, which we come to know only at runtime. This situation can be handled using the **AbstractFactoryInterface**, which is derived from the FactoryInterface.

The AbstractFactoryInterface defines a method to check whether the object can be created at the requested instance or not. If object creation is possible, it will create the object using the **__invokemethod** of the FactoryInterface and return it.

The signature of the AbstractFactoryInterface is as follows:

```
public function canCreate(ContainerInterface $container, $requestedName)
```

Initializer Method

The Initializer Method is a special option to inject additional dependency for already created services. It implements the **InitializerInterface** and the signature of the sole method available is as follows:

```
public function(ContainerInterface $container, $instance)
function(ContainerInterface $container, $instance) {
    if (! $instance instanceof EventManagerAwareInterface) {
        return;
    }
    $instance->setEventManager($container->get(EventManager::class));
}
```

In the above example, the method checks whether the instance is of type **EventManagerAwareInterface**. If it is of type EventManagerAwareInterface, it sets the event manager object, otherwise not. Since, the method may or may not set the dependency, it is not reliable and produces many runtime issues.

Delegator Factory Method

Zend Framework supports delegators pattern through **DelegatorFactoryInterface**. It can be used to decorate the service.

The signature of this function is as follows:

```
public function __invoke(ContainerInterface $container,
                         $name, callable $callback, array $options = null
);
```

Here, the **$callback** is responsible for decorating the service instance.

Lazy Services

Lazy service is one of those services which will not be fully initialized at the time of creation. They are just referenced and only initialized when it is really needed. One of the best example is database connection, which may not be needed in all places. It is an expensive resource as well as have time-consuming process to create. Zend framework provides **LazyServiceFactory** derived from the **DelegatorFactoryInterface**, which can produce lazy

service with the help of the **Delegator** concept and a 3rd party proxy manager, which is called as the **ocramius proxy manager**.

Plugin Manager

Plugin Manager extends the service manager and provides additional functionality like instance validation. Zend Framework extensively uses the plugin manager.

For example, all the validation services come under the **ValidationPluginManager**.

Configuration Option

The service manager provides some options to extend the feature of a service manager. They are **shared**, **shared_by_default** and **aliases**. As we discussed earlier, retrieved objects are shared among requested objects by default and we can use the **build()** method to get a distinct object. We can also use the **shared** option to specify which service to be shared. The ***shared_by_default*** is same as the **shared** feature, except that it applies for all services.

```
$serviceManager = new ServiceManager([
    'factories' => [
        stdClass::class => InvokableFactory::class
    ],
    'shared' => [
        stdClass::class => false // will not be shared
    ],
    'shared_by_default' => false, // will not be shared and applies to all
service
]);
```

The **aliases** option can be used to provide an alternative name to the registered services. This have both advantages and disadvantages. On the positive side, we can provide alternative short names for a service. But, at the same time, the name may become out of context and introduce bugs.

```
aliases' => [
    'std' => stdClass::class,
    'standard' => 'std'
]
```

7. Zend Framework – Event Manager

All modern applications need solid and flexible event components. Zend Framework provides one such component, **zend-eventmanager**. The zend-eventmanager helps to design high level architecture and supports subject/observer pattern and aspect oriented programming.

Install Event Manager

The event manager can be installed using the **Composer** as specified below:

```
composer require zendframework/zend-eventmanager
```

Concepts of the Event Manager

The core concepts of the event manager are as follows:

- **Event -** Event is arbitrarily named action, say **greet**.
- **Listener -** Any PHP callback. They are attached to the events and gets called when the event is triggered. The default signature of Listener is –

  ```
  function(EventInterface $e)
  ```

- **EventInterface Class -** Used to specify the event itself. It has methods to set and get event information like name (set/getName), target (get/setTarget) and parameter (get/setParams).
- **EventManager class -** The instance of the EventManager tracks all the defined events in an application and its corresponding listeners. The EventManager provides a method, **attach** to attach listener to an event and it provides a method, **trigger** to trigger any pre-defined event. Once trigger is called, EventManager calls the listener attached to it.
- **EventManagerAwareInterface -** For a class to support event based programming, it needs to implement the EventManagerAwareInterface. It provides two methods, **setEventManager** and **getEventManager** to get and set the event manager.

Example

Let us write a simple PHP console application to understand the event manager concept. Follow the steps given below.

- Create a folder "eventapp".
- Install **zend-eventmanager** using the composer.

- Create a PHP file **Greeter.php** inside the "eventapp" folder.
- Create class **Greeter** and implement the **EventManagerAwareInterface**.

```
require __DIR__ . '/vendor/autoload.php';
class Greeter implements EventManagerAwareInterface
  {
      // code
  }
```

Here, **require** is used to autoload all composer installed components.

Write the **setEventManager** method in class **Greeter** as shown below:

```
  public function setEventManager(EventManagerInterface $events)
  {
      $events->setIdentifiers([
          __CLASS__,
          get_called_class(),
      ]);
      $this->events = $events;
      return $this;
  }
```

This method sets the current class into the given event manager ($events argument) and then sets the event manager in local variable $**events**.

The next step is to write the **getEventManager** method in class **Greeter** as shown below:

```
  public function getEventManager()
  {
      if (null === $this->events) {
          $this->setEventManager(new EventManager());
      }
      return $this->events;
  }
```

The method gets the event manager from a local variable. if it is not available, then it creates an instance of event manager and returns it.

Write a method, **greet**, in class **Greeter**.

```
public function greet($message)
{
    printf("\"%s\" from class\n", $message);
    $this->getEventManager()->trigger(__FUNCTION__, $this,
                                          [ $message ]);
}
```

This method gets the event manager and fires / triggers events attached to it.

The next step is to create an instance of the **Greeter** class and attach a listener to its method, **greet**.

```
$greeter = new Greeter();

$greeter->getEventManager()->attach('greet', function($e) {
    $event_name = $e->getName();
    $target_name = get_class($e->getTarget());
    $params_json = json_encode($e->getParams());

    printf("\"%s\" event of class \"%s\" is called." .
                    " The parameter supplied is %s\n",
                $event_name,
                $target_name,
                $params_json);
});
```

The listener callback just prints the name of the event, target and the supplied parameters.

The complete listing of the **Greeter.php** is as follows:

```
<?php

require __DIR__ . '/vendor/autoload.php';

use Zend\EventManager\EventManagerInterface;
use Zend\EventManager\EventManager;
use Zend\EventManager\EventManagerAwareInterface;
```

```
class Greeter implements EventManagerAwareInterface
{
    protected $events;

    public function setEventManager(EventManagerInterface $events)
    {
        $events->setIdentifiers([
            __CLASS__,
            get_called_class(),
        ]);
        $this->events = $events;
        return $this;
    }

    public function getEventManager()
    {
        if (null === $this->events) {
            $this->setEventManager(new EventManager());
        }
        return $this->events;
    }
    public function greet($message)
    {
        printf("\"%s\" from class\n", $message);
        $this->getEventManager()->trigger(__FUNCTION__, $this,
                                                [ $message ]);
    }
}
$greeter = new Greeter();
$greeter->greet("Hello");

$greeter->getEventManager()->attach('greet', function($e) {
    $event_name = $e->getName();
    $target_name = get_class($e->getTarget());
    $params_json = json_encode($e->getParams());
```

```
    printf("\"%s\" event of class \"%s\" is called." .
                    " The parameter supplied is %s\n",
                $event_name,
                $target_name,
                $params_json);
});

$greeter->greet("Hello");
```

Now, run the application in the command prompt php **Greeter.php** and the result will be as follows:

```
"Hello" from class
"Hello" from class
"greet" event of class "Greeter" is called. The parameter supplied is ["Hello"]
```

The above sample application explains only the basics of an event manager. The Event manager provides many more advanced options such as **Listener Priority**, **Custom Callback Prototype / Signature**, **Short Circuiting**, etc. The Event manager is used extensively in the Zend MVC framework.

8. Zend Framework – Module System

The Zend Framework provides a powerful module system. The module system has three components. They are as follows:

- **Module Autoloader –** A Module Autoloader is responsible for locating and loading of modules from variety of sources. It can load modules packaged as **Phar archives** as well. The implementation of the Module Autoloader is located at myapp/vendor/zendframework/zend-loader/src/ModuleAutoloader.php.
- **Module Manager –** Once the Module Autoloader locates the modules, the module manager fires a sequence of events for each module. The implementation of the Module Manager is located at myapp/vendor/zendframework/zend-modulemanager/src/ModuleManager.php.
- **Module Manager Listeners –** They can be attached to the events fired by the Module Manager. By attaching to the events of module manager, they can do everything from resolving and loading modules to performing complex work for each modules.

MVC Web Module System

The MVC Web Application in the Zend Framework is usually written as Modules. A single website can contain one or more modules grouped by functionality. The recommended structure for MVC-Oriented module is as follows:

```
module_root/
    Module.php
    autoload_classmap.php
    autoload_function.php
    autoload_register.php
    config/
        module.config.php
    public/
        images/
        css/
        js/
    src/
        <module_namespace>/
            <code files>
    test/
```

```
        phpunit.xml
```

```
        bootstrap.php
        <module_namespace>/
            <test code files>
    view/
        <dir-named-after-module-namespace>/
            <dir-named-after-a-controller>/
                <.phtml files>
```

The structure is same as discussed in the previous chapter, but here it is generic. The **autoload_ files** can be used as a default mechanism for autoloading the classes available in the module without using the advanced **Module Manager** available in the **zend-modulemanager**.

- **autoload_classmap.php –** Returns an array of class name and its corresponding filename.
- **autoload_function.php –** Returns a PHP callback. This can utilize classes returned by autoload_classmap.php.
- **autoload_register.php –** Registers the PHP callback that is returned by the autoload_function.php.

These autoload files are not required but recommended. In the skeleton application, we have not used the **autoload_** files.

Module Class

The Module class should be named **Module** and the namespace of the module class should be **Module name**. This will help the Zend Framework to resolve and load the module easily. The **Application** module code in the skeleton(myapp) application,myapp/module/Application/src/Module.php is as follows:

```
namespace Application;


class Module
{
    const VERSION = '3.0.2dev';
    public function getConfig()
    {
        return include __DIR__ . '/../config/module.config.php';
```

```
    }
}
```

The Zend Framework module manager will call the **getConfig()** function automatically and will do the necessary steps.