Zend Framework 2

Certification Study Guide

version 1.0

Introduction

The Zend Framework 2 Certification is designed to measure your expertise in both understanding the concepts, rules, and code behind the framework, and equally important, your ability to take this knowledge and apply it to your development projects.

The certification was designed by the Zend Framework Education Advisory Board, an elite, global set of web application development experts (across a variety of companies) who established the criteria for knowledge and job task competencies and developed the related questions that assess Zend Framework expertise.

The Process

In order to become certified in the use of Zend Framework 2(ZF2), you will need to successfully pass an examination. The exam is administered world-wide by Pearson VUE. You will be required to take the exam at one of their VUE Testing Centers, available in over 3500 locations around the world. The exam is taken in an isolated room, using a specially configured computer, and is “closed-book”, so you will not be able to consult any reference material or use the Internet while taking it.

The Exam

The ZF2 Certification exam itself is very similar to most other IT exams offered, including the exam for Zend PHP 5 Certification. The exam is composed of approximately 75 randomly-generated questions, which must be answered in 90 minutes. Each question can be formulated in one of three ways:

- As a multiple-choice question with only one right answer
- As a multiple-choice question with multiple correct answers
- As a free-form question for which the answer must be typed in

The 13 areas of expertise created by the ZF Certification Education Advisory Board are:

- Service Manager
- Event Manager
- Module Manager
- MVC (Model-View-Controller)
- Performance
- Security
- Database
- Utility
- Authentication and Authorization
- Forms
- Filtering
- Internationalization
- Web Services

The Study Guide

This Study Guide provides guidance as to the topics to be covered, and an indication of the depth of knowledge required to pass the exam. It does not teach Zend Framework 2.

This guide assumes that anyone about to undertake the Certification exam is very familiar with the Framework and has extensive experience in utilizing its structure and components in application development.

The Guide therefore presents a concise treatment of the 13 required knowledge areas for certification, with reminders and hints on some essential aspects of the components. It is by no means a complete discussion of the entire framework, nor a complete presentation of everything you might find on the exam. It is a guide to the types of facts and coding practices that are considered essential for passing the exam.

Guide Structure

Each of the 13 topic areas tested by the ZF2 Certification exam has its own section within the Study Guide.

Note that the content for areas that are listed multiple times under the sub-topics (Ex: Front Controllers) will appear only once.

At the end of each topic will be a couple of representative questions, similar to those you will find on the examination. If you feel confident that you know the exam topic well you can take them first, or you can take these questions as a wrap-up to your studying the content.

If, in working through this guide, you discover that you are weak in some areas, you should utilize the online Programmers Reference Guide to Zend Framework, available at http://framework.zend.com/manual. This extensive documentation presents highly detailed discussions on various topics along with multiple code examples. Please note that any suggestion within this guide to consult the "Reference Guide" is referring to this document.

If you would like more practice in answering questions about the exam, including taking a simulated exam, and have a live instructor to answer your questions and guide your preparation studies, consider taking Zend's online Zend Framework 2 Certification course.

Zend Framework 2 is abbreviated as "ZF2" for brevity throughout this guide.

About Zend Framework 2

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, late static binding, lambda functions and closures.

Zend Framework 2 evolved from Zend Framework 1, a successful PHP framework with over 15 million downloads.

The component structure of Zend Framework 2 is unique; each component is designed with few dependencies on other components. ZF2 follows the SOLID object oriented design principle. This loosely coupled architecture allows developers to use whichever components they want. We call this a "use-at-will" design. We support **Pyrus** and **Composer** as installation and dependency tracking mechanisms for the framework as a whole and for each component, further enhancing this design.

We use **PHPUnit** to test our code and **Travis CI** as a Continuous Integration service.

While they can be used separately, Zend Framework 2 components in the standard library form a powerful and extensible web application framework when combined. Also, it offers a robust, high performance MVC implementation, a database abstraction that is simple to use, and a forms component that implements HTML5 form rendering, validation, and filtering so that developers can consolidate all of these operations using one easy-to-use, object oriented interface. Other components, such as Zend\Authentication and Zend\Permissions\Acl, provide user authentication and authorization against all common credential stores.

Still others, with the ZendService namespace, implement client libraries to simply access the most popular web services available. Whatever your application needs are, you're likely to find a Zend Framework 2 component that can be used to dramatically reduce development time with a thoroughly tested foundation.

The principal sponsor of the project 'Zend Framework 2' is **Zend Technologies**, but many companies have contributed components or significant features to the framework. Companies such as **Google**, **Microsoft**, and **StrikeIron** have partnered with Zend to provide interfaces to web services and other technologies they wish to make available to Zend Framework 2 developers.

Zend Framework 2 could not deliver and support all of these features without the help of the vibrant Zend Framework 2 community. Community members, including contributors, make themselves available on **mailing lists**, **IRC** channels and other forums. Whatever question you have about Zend Framework 2, the community is always available to address it.

SERVICE MANAGER TOPIC OVERVIEW

- The Service Managers in Zend Framework 2 are used to find and instantiate services (classes), including fulfilling any dependencies that a service may require

- Zend Framework 2 uses Service Managers throughout the entire framework, with specialized Service Managers used to find specific types of services including:
  - Controllers
  - Plugins
  - View Helpers
  - Input Filters
  - Filters
  - Validators
  - Form Elements
  - Hydrators
  - Application Services (user created)

Service Manager

Service Locator

The **Service Locator** design pattern describes a method of obtaining a service via an abstraction layer.

Service Managers in Zend Framework 2 implement the **`ServiceLocatorInterface`** interface. The interface prescribes two methods, has() to determine if a given service has been registered with the Service Manager, and get() to return an instance of that service.

When asked to return a service, **`Zend\ServiceManager\ServiceManager`** objects will lazily create new instances of the service requested, using typically using service factories to create the services and meet their dependencies. Services should be created with dependencies defined in the constructor (or with setters). Wherever possible, **dependency injection** should be used to fulfill dependencies in services created by the ServiceManager.

Plugin Managers

There are multiple implementations of the `ServiceLocatorInterface` that are tasked with locating different services and plugins.

Abstract Plugin Manager

The **`AbstractPluginManager`** is a Service Manager implementation for managing plugins. The abstract class provides methods to set and get plugins, along with creating services from invokable classes, factories, and callable objects such as closures.

Controller Manager

The `Zend\Mvc\Controller\ControllerManager` extends the `AbstractPluginManager`, and is a plugin manager that is responsible for the retrieval and creation of controller classes. It is used internally by the MVC router to locate and retrieve controllers based on the key provided by the matched route. It is very important that controllers are in a separate plugin manager for speed and security reasons.

Note: If the controllers were in a generic plugin manager, it would be theoretically possible to create and dispatch controller methods by simply crafting URLs that dynamically matched service names not intended to be used as controllers.

Other Plugin Managers

There are also a number of plugin managers that are tasked with the creation and retrieval of specific types of services and plugins. Each of these plugin managers extends the `AbstractPluginManager`:

- `ControllerPluginManager` - creation and retrieval of controller plugins
- `ViewHelperPluginManager` - creation and retrieval of view helpers
- `InputFilterPluginManager` - creation and retrieval of input filters
- `FilterPluginManager` - creation and retrieval of filters
- `ValidatorPluginManager` - creation and retrieval of validators
- `FormElementManager` - creation and retrieval of form elements
- `HydratorPluginManager` - creation and retrieval of hydrators

These plugin managers will implement the `validatePlugin()` method to ensure that the items registered are of the correct type for this plugin manager.

Shared Service Manager

While it is expected that users will contribute their own plugins via the plugin managers already mentioned, users will also want to create and retrieve other services not included in the specialized plugin managers. The Shared Service Manager exists for this reason. `Zend\ServiceManager\ServiceManager` is an implementation of the `ServiceLocatorInterface` used for storing custom services, as well as to internally to store services used through the Framework. It is typically written by the application developer.

The Shared Service Manager is available from any of the specialized plugin managers via the `getServiceLocator()` method of the `AbstractPluginManager`.

Note: When using the phrase "service manager" in the rest of this study guide, we will be referring to both service managers and plugin managers.

Configuration

Service managers (and plugin managers) can be configured by passing a class that implements the **`Zend\ServiceManager\ConfigInterface`** into the manager as a constructor argument. The **`Zend\ServiceManager\Config`** class is a concrete implementation of this interface that is shipped with the framework and can be used to configure service managers.

There are multiple ways that service managers can be configured to create and retrieve services, including factories (and abstract factories), invokables, and aliases. In addition, initalizers can be registered to automatically run setter injection as required.

The `Config` object takes a well-formed array as a constructor parameter, and uses this array to configure the service manager with factories, abstract factories, invokables, services, aliases and initializers. The array is in the format of key-value pairs, where the key is the unique service name that identifies this service, and the value is the means of creating that service. Note that service names are normalized, spaces and special characters are removed, and the string is converted to lowercase, so `user-service` is the same as `User\Service`, which are both normalized to ‘`userservice`\`.

Invokables

The simplest way to create services through a service manager is by using an invokable. An invokable is simply a class that can be instantiated with no dependencies (or with dependencies that are fulfilled by an initializer). Invokables can be defined using the 'invokables' key of the configuration array passed to the Config object:

```
$config = new Config(array(
    'invokables' => array(
        'hydrator' => 'Zend\Stdlib\Hydrator\ClassMethods'
    ),
));
$serviceManager = new ServiceManager($config);
```

In this example, the hydrator has no dependencies (either constructor or setter), and so can simply be created using an invokable.

Factories

Factories are classes or callables that can return a configured instance of the service requested. Callables should return the configured service, while classes should implement the **FactoryInterface** interface, and should return the configured service from the prescribed createService() method.

Both callables and classes get the instance of the service manager that called them passed as a parameter to the method called. This allows the factory to fulfil any dependencies of the service by retrieving those dependencies by either the calling service manager, or the shared service manager (via the getServiceLocator() method).

Factories are defined in the factories key of the configuration array:

```
$config = new Config(array(
    'factories' => array(
        'UserService' => 'User\Factory\UserServiceFactory',
                        // class implements the FactoryInterface
        'user' => function(ServiceLocatorInterface $serviceManager) {
            $service = $serviceManager->get('UserService');
                        // defined above
            $user = new User($service); // constructor dependency injection
            return $user;
        }
    ),
));
$serviceManager = new ServiceManager($config);
```

Abstract Factories

Abstract Factories are factories that are capable of multiple services. Abstract factories implement the **AbstractFactoryInterface** and are queried after factories and invokables. If there is no entry for this service as either a factory or invokable then registered abstract factories are asked if they can create this service via the prescribed canCreateServiceWithName() method. If this method returns true, then the service is requested via the createServiceWithName() method.

As with the standard factory, the service manager that is being used to request this service is passed in as a parameter so that dependencies of the service can be fulfilled. Abstract factories are registered using the abstract_factories key of the configuration array.

```
class AbstractTableGatewayFactory implements AbstractFactoryInterface
{
    public function createServiceWithName(ServiceLocatorInterface
$serviceLocator, $name, $requestedName)
    {
        // does this classname end with Table?
        return fnmatch('*Table', $requestedName);
    }

    public function canCreateServiceWithName(ServiceLocatorInterface
$serviceLocator, $name, $requestedName)
    {
        $adapter = $serviceLocator->get('Zend\Db\Adapter\Adapter');
        return new TableGateway(str_replace('Table', '', $requestedName),
                                                    $adapter);
    }
}
$config = new Config(array(
    'abstract_factories' => array(
        'AbstractTableGatewayFactory'
    ),
));
$serviceManager = new ServiceManager($config);

$userTable = $serviceManager->get('userTable');
```

Aliases

Aliases can be used to point a given service name to another service that has already been defined by another method. Aliases can be registered using the aliases key of the configuration array.

```
$config = new Config(array(
    'aliases' => array(
        'user' => 'UserService',
    );
));
$serviceManager = new ServiceManager($config);
```

Initializers

Initializers are used to run additional initialization tasks after a service has been created. The newly created service is passed to all registered initializers after creation. This is useful for meeting dependencies using setter injection. Initializers are also used to check against Aware interfaces (both user defined and shipped with the factory) so that those dependencies can be met.

Inititialzers can be registered using the initializers key of the configuration array, and can be any callable or class that implements the `InitializerInterface`.

```
interface UserServiceAwareInterface
{
    public function setUserService(UserService $userService);
}
class ProductService implement UserServiceAwareInterface
{
    protected $userService;

    public function setUserService(UserService $userService)
    {
        $this->userService = $userService;
    }
}
$config = new Config(array(
    'initializers' = array(
        'userService' => function($service, ServiceLocatorInterface
$serviceManager) {
            if($service instanceof UserServiceAwareInterface) {
                $userService = $serviceManager->get('UserService');
                $service->setUserService($userService);
            }
        }
    ),
));
$serviceManager = new ServiceManager($config);
```

Test Your Knowledge: Questions

1

Given the following list of configuration files containing service manager configuration, which one will not be automatically loaded?

- `config/application.config.php`
- `config/autoload/local.php`
- `module/Application/config/autoload/local.php`
- `module/Application/config/module.config.php`

2

Which methods does a factory need to define when implementing the AbstractFactoryInterface? (Choose 2)

- `canCreateServiceWithName()`
- `canCreateServiceByName()`
- `createServiceWithName()`
- `createServiceByName()`
- `isCreatableWithName()`
- `isCreatableByName()`

Test Your Knowledge: Answers

1

Given the following list of configuration files containing service manager configuration, which one will not be automatically loaded?

- `config/application.config.php`
- `config/autoload/local.php`
- `module/Application/config/autoload/local.php`
- `module/Application/config/module.config.php`

2

Which methods does a factory need to define when implementing the AbstractFactoryInterface? (Choose 2)

- `canCreateServiceWithName()`
- `canCreateServiceByName()`
- `createServiceWithName()`
- `createServiceByName()`
- `isCreatableWithName()`
- `isCreatableByName()`

EVENT MANAGER TOPIC OVERVIEW

- The **Event Manager** is a component used to implement the **observer pattern** (**aspect-oriented design**) and **event-driven programming** in Zend Framework 2 applications

- It functions by attaching event listeners to an event manager, which then triggers events

- Event managers implement the `EventManagerInterface`

- ZF2 comes with a concrete implantation of this interface: `Zend\EventManager\EventManager`

- An EventManager can be created by simply instantiating a new instance: `$eventManager = new EventManager();`

Zend\EventManager

Attaching Event Listeners

You use the `attach()` method to attach event listeners to the event manager. This method takes three parameters, the **event name** to attach to, a **callable** to run when the event is triggered, and the **priority** of the event.

Parameters:

Event Name Parameter

The Event name is a unique identifier for this event. It will be used to trigger the event at a later time.

Callable Parameter

The callable parameter will be called when the named event is triggered. The callable will be passed a single parameter which will be an instance of the **EventInterface** - usually an **Event** which is a concrete implementation of this interface that is provided with the Framework. The `EventInterface` provides information about the triggered Event through the `getName()` and `getTarget()` methods. Optional parameters passed through triggering the event can be accessed using the `getParams()` method.

Priority Parameter

The optional priority parameter is used to determine the order in which event listeners are called when multiple listeners are registered to the same event (with the highest priorities getting triggered first). Priorities can be positive or negative, and the default priority is 1. If two listeners share the same priority, they are triggered in the order in which they were registered.

Example

```
// attach a listener to the item-added event
$logger = new Logger();
$eventManager->attach('item-added', function(EventInterface $event) use
($logger) {
    $params = $event->getParams();
    $logger->log('User ' . $params['userId'] .
                 ' added item ' . $params['itemId']);
}, 100);
```

Triggering Events

Events are trigged through the Event Manager using the `trigger()` method. The trigger method takes 4 parameters: the event name, an optional target, an optional array of arguments, and an optional callback.

Parameters:

Event Name Parameter

The Event Name is the unique identifier for this event. In order for listeners to fire, this should be identical to the name used to attach listeners in the `attach()` method.

Target Parameter

The `$target` parameter will typically be the current object instance, or the name of the function if triggering from outside an object (or a static function). This allows the instance of the triggering class to be used from within the listener.

Parameters Parameter

The Arguments parameter is an key/value array of arguments that will be passed to the event and is available in the listener via the `getParams()` method.

Callback Parameter

The callback should be a callable, and will be called after the registered listener has completed. The callback will be passed the results returned from that listener.

Example

```
$params = array('userId' => $user->getUserId(),
                'productId' => $product->getProductId());
$eventManager->trigger('item-added', $this, $params);
```

Short Circuiting

When multiple listeners are registered it is possible to stop all listeners firing by stopping event propagation from within a listener. This is done using the `stopPropagation()` method of the Event object from within the listener.

In this example, we only allow the second logging listener to fire if the email service returns false.

```
// email the user
$eventManager->attach('item-added', function(EventInterface $event) use
($email) {
    $params = $event->getParams();
    $result = $email->emailUser($params['userId']);
    if($result) {
        $event->stopPropagation(true);
    }
}, 100);

// log the item added
$eventManager->attach('item-added', function(EventInterface $event) use
($logger) {
    $params = $event->getParams();
    $logger->log('User ' . $params['userId'] .
                 ' added item ' . $params['itemId']);
}, -100);

$params = array('userId' => $user->getUserId(), 'productId' =>
                                                $product->getProductId());
$eventManager->trigger('item-added', $params, $this);
```

`It is also possible to short-circuit the event listeners from within a trigger callback(), by returning true from the callable.`

```
$eventManager->trigger('item-added', $params, $this, function($result){
    // short circuit if error returned
    return ($result instanceof Error);
});
```

Aggregate Listeners

The `ListenerAggregateInterface` provides an alternative method to attach multiple listeners via a single class.

Aggregate listeners allow multiple listeners to be attached, but more importantly detached from a single place. Listeners are also triggered within scope of the class that implements the interface, which can be convenient for listeners that need to make use of external services. Aggregate listeners are an excellent choice if you need to detach listeners as they allow you to store the `Zend\Stdlib\Callbackhandler` statefully within the aggregate listener, which makes removing much easier.

Listeners are attached using the `attach()` method, passing an `EventManagerInterface` instance as the single parameter. Listeners are detached using the `detach()` method, which accepts an `EventManagerInterface` instance as its parameter.

```
class MailAggregateListener implements ListenerAggregateInterface
{
    protected $listeners = array();
    protected $emailService;

    public function __construct(EmailService $emailService)
    {
        $this->emailService = $emailService;
    }

    public function handleEvent(EventInterface $event)
    {
        $this->emailService->emailEvent($event);
    }

    public function attach(EventManagerInterface $eventManager)
    {
        $listenTo('add-cart','remove-cart','empty-cart','checkout-cart');
        foreach($listenTo as $listen) {
            $this->listeners[$listen] = $this->attach($listen,
                                                  array($this,'handleEvent');
        }
    }

    public function detach(EventManagerInterface $eventManager)
    {
        // just detach remove-cart listener
        $eventManager->detach($this->listeners['remove-cart']);
    }
}
```

Event Managers Inside Zend Framework 2

The event manager is used extensively inside ZF2. Events are triggered during bootstrap to allow customization of how the bootstrap process occurs. Routing happens via an event listener; an event is triggered to allow changing of the outcome (or catching errors). Controller execution happens as part of a dispatch listener, and the entire view rendering process can be skipped by short circuiting the propagation of events.

Shared Event Manager

The `SharedEventManagerInterface` interface provides a way to attach listeners to many event managers, or to an event manager not in the scope of this class. It does not get triggered directly itself; rather, any `EventManager` that has a `SharedEventManagerInterface` attached will query that `SharedEventManager` for events to trigger when events are triggered within itself. This means that a module can attach a listener to another module's events without needing direct access to any of that module's class instances.

The `SharedEventManager` is a concrete implementation of the `SharedEventManagerInterface` that is shipped with the Framework.

Attaching to event listeners to a `SharedEventManager` is similar to a standard `EventManager`. The `attach()` method is used but with an extra parameter at the start; `$id` - the identifier for the event triggered. Each `EventManager` instance can compose zero or more *identifiers*. These are used by the `EventManager` instance to look up listeners in the `SharedEventManager` instance for the event that is being triggered.

```
$sharedEventManager = new SharedEventManager();
$sharedEventManager->attach('Product', 'add', function(Event $event) {
     echo 'Product Added!';
});

$eventManager = new EventManager('Product');
$eventManager->setSharedManager($sharedEventManager);
$eventManager->trigger('add', new Product());
```

This will only trigger when the **identifier** first attach parameter is the same identifier as the **target** second trigger parameter.

Test Your Knowledge: Questions

1

Which methods must be allowed when you implement Zend\EventManager\ListenerAggregateInterface? (Choose 2)

- **attach**
- **listen**
- **detach**
- **trigger**

2

What is the expected result of the following code?

```
<php>
$sem = new Zend\EventManager\SharedEventManager();
$sem->attach('UserService', 'register', function() {
   echo 'registered!';
});
$evm = new Zend\EventManager\EventManager();
$evm->setSharedManager($sem);
$evm->trigger('register');
</php>
```

- **Nothing is emitted**
- **The string “registered!” is emitted**
- **An exception is triggered by the event manager**

Test Your Knowledge: Answers

1

Which methods must be allowed when you implement Zend\EventManager\ListenerAggregateInterface? (Choose 2)

- attach
- listen
- detach
- trigger

2

What is the expected result of the following code?

```
<php>
$sem = new Zend\EventManager\SharedEventManager();
$sem->attach('UserService', 'register', function() {
   echo 'registered!';
});
$evm = new Zend\EventManager\EventManager();
$evm->setSharedManager($sem);
$evm->trigger('register');
</php>
```

- Nothing is emitted
- The string “registered!” is emitted
- An exception is triggered by the event manager

MODULE MANAGER TOPIC OVERVIEW

- The ZF2 Module Manager is a powerful way to create re-usable application code that can be plugged in to multiple projects easily

- A ZF2 Module is a discrete group of re-usable code that typically represents a single recycling concept from an application. The type of code that can be presented is anything you would use in a normal MVC application, including services, configuration, controllers and view scripts

- Because modules can provide configuration, it effectively means that modules can provide routes, load services into the various service managers, and attach listeners to events

Zend\ModuleManager

Module Manager

`Zend\ModuleManager\ModuleManager` is an implementation of the `Zend\ModuleManager\ModuleManagerInterface` that is responsible for adding and configuring modules, and making their namespaced code available to the application.

The Module Manager takes a well formed configuration array as a constructor argument, which includes a list of modules to load, where to look for modules, and the path from which to merge autoload config files.

The modules can then be loaded with the `loadModules()` method.

```
$moduleManager = new ModuleManager(array(
    'modules' => array( // which modules to load
        'Application',
        'Blog',
        'Catalog',
        'Cart',
        'Checkout',
    ),
    'module_listener_options' => array(
        'module_paths' => array( // where to look for modules
            './module',
            './vendor',
        ),
        'config_glob_paths' => array( // where to merge autoload config
files
            'config/autoload/{,*.}{global,local}.php',
        ),
    ),
));
$moduleManager->loadModules();
```

In the context of the ZF2 Skeleton Application, the config file found at `config/application.config.php` is the config file passed to the Module Manager on instantiation.

Module Autoloading

Modules are usually autoloaded by the **`Zend\Loader\ModuleAutoloader`**, which is a separate autoloader responsible only for loading modules and the `Module` class from configured locations.

The Module.php Class

In its simplest form, a `Module.php` class simply needs to be a class named `Module` that resides in your module's namespace.

```
namespace Catalog;

class Module
{
}
```

Zend Framework 2 ships with a number of **Feature** interfaces that allow modules to easily provide configuration and functionality. The interfaces describe methods that can be implemented to provide the described functionality, but the interfaces don't need to be actually implemented (only the methods written) for the functionality to be loaded.

Modules can provide configuration both through the Feature interfaces, or through an array config file that is provided by the `getConfig()` method.

Config Merging

Configuration for various components can be provided in multiple places, but all configuration is merged recursively into one large configuration file by the `Zend\Stdlib\ArrayUtils::merge` method. Because configurations are merged recursively, the order that they are added to the merged configuration array is really important to avoid unexpected overwriting.

Configurations are merged in the following order:

1. The array returned by the module's `getConfig()` method

2. `config/autoload/global.*.php` -- global autoload files

3. `config/autoload/local.*.php` -- local autoload files

4. The methods described by the Feature interface -- `get*Config()`

Once steps 1-3 have been completed, the semi-merged config can be cached by the module manager, so if you are providing configuration using closers, these should be placed in the Feature methods (step 4) rather than the config provided by step 1.

Module Events

In order to make modules as flexible as possible, the `ModuleManager` triggers a number of events during the process of loading modules. Initially, the `loadModules` event is triggered, then as each module is loaded (in the order passed in through the configuration array), a number of events are triggered for each module loaded. The list of events is extensive, and is comprehensively covered in the **manual**.

The `Module.php` file may implement an `onBootstrap()` method, which (if defined) is called immediately after the all modules have been bootstrapped on a module-by-module basis. This method is just an easy way of attaching to the `onBootstrap` event and, just like a typical event, is passed the `Event` object as the only parameter. Attaching to other events from within the module is typically done here.

The `Module` class may also implement an `init()` method that is called *before* the module is bootstrapped, and should be used to configure anything that this module needs during the bootstrap process.

Test Your Knowledge: Questions

1

Which statements are INCORRECT regarding Zend Framework 2 modules? (Choose 3)

- A module can be reused in multiple projects
- A module cannot have dependencies to user-defined modules*
- A module can provide several hooks to provide module specific configuration
- A module can provide static assets (JS, CSS) that are automatically copied to the public folder by the module manager *
- A module is identified by its fully qualified class name (FQCN)*

2

Which events are not triggered by `Zend\ModuleManager\ModuleManager`? (Choose 2)

- `loadModules`
- `loadModule.pre`
- `loadModule`
- `loadModule.post`

Test Your Knowledge: Answers

1

Which statements are INCORRECT regarding Zend Framework 2 modules? (Choose 3)

- A module can be reused in multiple projects
- A module cannot have dependencies to user-defined modules
- A module can provide several hooks to provide module specific configuration
- A module can provide static assets (JS, CSS) that are automatically copied to the public folder by the module manager
- A module is identified by its fully qualified class name (FQCN)

2

Which events are not triggered by `Zend\ModuleManager\ModuleManager`? (Choose 2)

- `loadModules`
- `loadModule.pre`
- `loadModule`
- `loadModule.post`

MVC TOPIC OVERVIEW

- The **Model-View-Controller** software design pattern is used to separate business logic (the model part) from the presentation layer (the view part), and includes the code used to connect both parts together (the controller part)

- Zend Framework 2 is an MVC-based framework

- The term MVC typically refers to the components of the framework used to dispatch a traditional http request

Zend\Mvc

MVC Dispatch Flow

The MVC dispatch flow is the order in which components are dispatched during a typical http request from the Skeleton Application.

1. Autoloading is set up
2. Module manager loads modules
3. Router matches URI
4. Action method of controller class is called
5. View layer renders html

Steps 3-5 can be combined together and referred to as the *Dispatch Process*.

Note: The Module Manager dispatch process is covered in the Module Manager chapter.

Routing

Routing is the action of converting the url requested into a dispatchable `Action` method of a `Controller` class. In Zend Framework 2, routing is handled by a class that implements the **`RouteStackInterface`** interface. ZF2 ships with the **`SimpleRouteStack`** implementation which is used to route requests. This Router is created by the **`RouterFactory`** via the Service Manager.

Routes are registered using the `addRoutes()` method, passing in either a well-formed array, or an implementation of the **`RouteInterface`** interface. ZF2 ships with concrete implementations of the interface; for example, for matching **`HTTP`** requests.

If a route is matched, the router will return a **`RouteMatch`** that is used by the dispatch process to match and retrieve the controller class from the `ControllerManager` (see the chapter on Service Managers for more information).

Typically, routes are defined in the `getConfig()` method of the `Module` class using a well-formed array. Routes defined this way in the MVC are used to create and add the respective `RouteInterface` implementation to the Router as part of the dispatch process.

Controllers

`Controllers` are classes tasked with gluing together the data returned from the `Model` classes, and presenting them to the `View` layer. `Controllers` in Zend Framework 2 must implement the **`DispatchableInterface`** interface, and usually will extend the **`AbstractActionController'`** (which in turn extends the **`AbstractController`**).

An `Action` is simply a method of the controller that returns a class that implements the **`ModelInterface`**, usually a **`ViewModel`**. If nothing is returned, then an empty `ViewModel` is created; if an array is returned, then a `ViewModel` is created with that array as parameters.

New controller functionality can be added to controllers by creating a class that implements the controller **`PluginInterface`**, usually by extending the **`AbstractPlugin`** abstract class, which is then registered with the controller plugin manager.

View

The view layer is tasked with creating and returning the rendered HTML that will be sent to the browser as the body of the HTTP response. The view layer is essentially split into two parts: the *View Model*, which is a container for information that is needed to render the view, and the *View Renderer*, which is the class that performs the rendering process.

View Renderers take *view templates*, which are html files with added PHP scripts to selectively modify the output. The PHP employed in view templates should be lightweight, and used only to iterate or make conditional decisions on what to display.

Typically, the **`PhpRenderer`** is used to render HTML templates, but other renderers that implement the **`RendererInterface`** are also shipped with the Framework, such as the **`JsonRenderer`** for rendering serialized JSON, and the **`FeedRenderer`** for rendering RSS feeds.

The view template is located using the **`ViewResolverInterface`**. Within the MVC, the **`TemplatePathStack`** implementation is used to look through registered paths until a matching path is found. Paths are registered using the `addPaths()`, `addPath()` and `setPaths()` methods in a 'last-in, first-out' fashion.

View Helpers

View Helpers are available in the view template and extend the `ViewRenderer` to provide additional functionality. A number of useful view helpers are shipped with the Framework by various components; typically these are presented by the component in the View/Helper directory of that component. View helpers implement the **`HelperInterface`** and use the `__invoke()` magic method to return data.

View helpers are registered with the view helper plugin manager, which is discussed in more detail in the Service Managers chapter.

Escaping and Filtering

Zend Framework 2 ships with a number of `Escape*()` methods, designed to safely escape data so that it is securely displayed in the browser. Escaping refers to the process of adding backslashes before reserved characters so that the browser's parser treats the data as HTML rather than as anything dynamic. Because the context in which a string is being echoed is important with regard to *how* the string is escaped, ZF2 ships with a number of view helpers that should be used in the context they describe:

• `EscapeHtml`	for echoing into standard html
• `EscapeCss`	for echoing into css
• `EscapeHtmlAttr`	for when you want to echo into a html attribute
• `EscapeJs`	for echoing into a javascript block
• `EscapeUrl`	for echoing into an Url such as an anchor

Test Your Knowledge: Questions

1

Within the bootstrapping process, the requested action controller will be instantiated so that it is prepared for dispatching later in the application workflow. (T/F)

- True
- False

2

Which of the following interfaces are not implemented by an action controller within the `Zend\Mvc\Controller` component? (Choose 2)

- `Zend\Stdlib\DispatchableInterface`
- `Zend\EventManager\EventManagerAwareInterface`
- `Zend\EventManager\SharedEventManagerAwareInterface`
- `Zend\Mvc\InjectApplicationEventInterface`
- `Zend\Mvc\Controller\ControllerManagerAwareInterface`
- `Zend\ServiceManager\ServiceLocatorAwareInterface`

Test Your Knowledge: Answers

1

Within the bootstrapping process, the requested action controller will be instantiated so that it is prepared for dispatching later in the application workflow. (T/F)

- True
- False

2

Which of the following interfaces are not implemented by an action controller within the `Zend\Mvc\Controller` component? (Choose 2)

- `Zend\Stdlib\DispatchableInterface`
- `Zend\EventManager\EventManagerAwareInterface`
- `Zend\EventManager\SharedEventManagerAwareInterface`
- `Zend\Mvc\InjectApplicationEventInterface`
- `Zend\Mvc\Controller\ControllerManagerAwareInterface`
- `Zend\ServiceManager\ServiceLocatorAwareInterface`

PERFORMANCE TOPIC OVERVIEW

- In this chapter we will explore various elements of functionality in Zend Framework 2 that provide options to enhance the performance of your application:
  - Caching with Zend
  - Autoloading
  - Configuration optimization

- ZF2 provides a range of backend storage and frontend capture classes that supply a range of support options for caching patterns, ranging from simple configuration arrays to blocks of html code from your templates

Storage Adapters

Storage adapters are wrappers for real storage resources, such as memory and the filesystem, using the well-known *Adapter* pattern. They come with many methods to read, write, and modify stored items, and to gather information about stored items and the storage system. Many of these methods throw exceptions, which can be caught manually, by defining an exception callback to log exceptions, or finally by using the plugin **`Zend\Cache\Storage\Plugin\ExceptionHandler`**.

The storage adapters shipped with Zend Framework 2 are:

- Apc
- Dba
- Filesystem
- Memcached
- Memory
- Redis
- Session
- Wincache
- XCache
- ZendServerDisk
- ZendServerShm

All of the adapters, and any storage adapter you write to use with ZF2, must implement **`Zend\Cache\Storage\StorageInterface`**.

Creating Caching Adapters

Caching adapters can be created via **`Zend\Cache\StorageFactory`**, using its `factory()` method to create the adapter and attach all requested plugins. An alternative way is to instantiate one of the `Zend\Cache\Storage\Adapter\*` classes.

Configuration

Configuration is handled by **`Zend\Cache\Storage\Adapter\AdapterOptions`** or an adapter-specific options class, if it exists. You have many options for passing the options instance to the `class`: at instantiation; via the `setOptions()` method; via an associative array of options in either place (internally, these are then passed to an options class instance); or by passing either the options instance or associative array to the `Zend\Cache\StorageFactory::factory` method. See the Zend Framework 2 manual for a full list of configuration options available.

Each adapter may have separate capabilities, such as support for separate data types. They may also have advantages others do not have, such as a Filesystem storage that supports metadata like Created and Modified times. While the Redis adapter does not have such capabilities, it is much faster and easier to access across a cluster of servers.

Storage Capabilities

Storage capabilities describe how a storage adapter works and which features it supports. The method `getCapabilities()` can provide these capabilities, but only the storage adapter and its plugins can modify them. The following example queries the storage adapter capabilities and branches operations based on the results:

```
use Zend\Cache\StorageFactory;

$cache              = StorageFactory::adapterFactory('filesystem');
$supportedDatatypes = $cache->getCapabilities()->getSupportedDatatypes();

// now you can run specific code respecting a supported feature

if ($supportedDatatypes['object']){
    $cache->set($key,$object);
} else {
    $cache->set($key,serialize($object));
}
```

As capabilities are mutable (for example, by changing options), you can subscribe to the "change" event to receive notifications.

If you are writing your own plugin or adapter, you can also change capabilities. You can create your own marker to instantiate a new object of **`Zend\Cache\Storage\Capabilities.`** Full details of the capabilities available for the storage adapters are available in the Zend Framework 2 Manual.

Storage Plugins

Cache storage plugins are objects used to add missing functionality or to influence behavior of a storage adapter. The plugins listen to events triggered by adapters and can:

- change called method arguments (*.post - events)
- skip and directly return a result (using `stopPropagation`)
- change the result (with `setResult`)
- catch exceptions

Example: Listening for capability change events

```
use Zend\Cache\StorageFactory;

$cache=StorageFactory::adapterFactory('filesystem', array(
    'no_atime'=> false,
));

// Catching capability changes

$cache->getEventManager()->attach('capability', function ($event){
    echo count($event->getParams()).' capabilities changed';
});

// change option which changes capabilities

$cache->getOptions()->setNoATime(true);
```

Creating Storage Plugins

Storage plugins can either be created from `Zend\Cache\StorageFactory` with the `pluginFactory()` method, or by directly instantiating one of the `Zend\Cache\Storage\Plugin\*` classes. `Zend\Cache\StorageFactory` comes with the `factory()` method to create an adapter and all given plugins at the same time.

```
use Zend\Cache\StorageFactory;

// Via Storage Adapter factory:

$cache = StorageFactory::factory( array(
    'adapter' => 'filesystem',
    'plugins' => array('serializer'),
));

// Alternately, via plugin factory + addPlugin():

$cache  = StorageFactory::adapterFactory('filesystem');
$plugin = StorageFactory::pluginFactory('serializer');

$cache->addPlugin($plugin);

// Or completely manually:

$cache  = new Zend\Cache\Storage\Adapter\Filesystem();
$plugin = new Zend\Cache\Storage\Plugin\Serializer();

$cache->addPlugin($plugin);
```

For a list of available plugins and related methods, consult the ZF2 manual.

Zend\Cache

Cache patterns are configurable objects to solve known performance bottlenecks. Each should be used only in the specific situations they are designed to address. For example, you can use one of the CallbackCache, ObjectCache, or ClassCache patterns to cache method and function calls; to cache output generation, the OutputCache pattern can be used.

All cache patterns implement **Zend\Cache\Pattern\PatternInterface**.

Configuration is provided via the **Zend\Cache\Pattern\PatternOptions** class, which can be instantiated with an associative array of options passed to the constructor of any given PatternInterface implementation.

To configure a pattern object, you can set an instance of **Zend\Cache\Pattern\PatternOptions** with setOptions(), or provide your own options (either as an associative array or a PatternOptions instance) as the second argument to the factory. It is also possible to use a single instance of **Zend\Cache\Pattern\PatternOptions** and pass it to multiple pattern objects.

Creating Pattern Objects

Pattern objects can be created from the provided **Zend\Cache\PatternFactory** factory, or, by instantiating one of the Zend\Cache\Pattern\*Cache classes.

```
// Via the factory:

$callbackCache=Zend\Cache\PatternFactory::factory('callback', array(
    'storage'=>'apc',
));

// OR, the equivalent manual instantiation:

$callbackCache = new Zend\Cache\Pattern\CallbackCache();
$callbackCache->setOptions(
    new Zend\Cache\Pattern\PatternOptions( array('storage'=>'apc'))
);
```

For a list of available methods, consult the ZF2 manual.

Cache Patterns

Zend\Cache\Pattern\CallbackCache caches calls of specific functions and methods, given as a callback.

```
use Zend\Cache\PatternFactory;
use Zend\Cache\Pattern\PatternOptions;

// Via the factory:

$callbackCache=PatternFactory::factory('callback', array(
    'storage'      => 'apc',
    'cache_output' => true ,
));

// OR, the equivalent manual instantiation:

$callbackCache= **new** \Zend\Cache\Pattern\CallbackCache();
$callbackCache->setOptions( new PatternOptions( array(
    'storage'      => 'apc',
    'cache_output' => true,
)));
```

Zend\Cache\Pattern\ClassCache is an extension to the CallbackCache pattern. It has the same methods, but instead generates an internally-used callback in the base of the configured class name and the given method name.

```
use Zend\Cache\PatternFactory;

$classCache=PatternFactory::factory('class', array(
    'class'   => 'MyClass',
    'storage' => 'apc'
));
```

Zend\Cache\Pattern\OutputCache caches output between calls to start() and end().

```
use Zend\Cache\PatternFactory;

$outputCache=PatternFactory::factory('output', array(
    'storage'=>'apc'
));
```

The `CaptureCache` pattern is used to auto-generate static resources in the base of a HTTP request. The web server needs to be configured to run a PHP script generating the requested resource so further requests for the same resource can be shipped without calling PHP again. It comes with basic logic to manage generated resources.

Example: Usage as an Apache-404 handler.

```
# .htdocs

ErrorDocument 404/index.php

// index.php

use Zend\Cache\PatternFactory;

$capture=Zend\Cache\PatternFactory::factory('capture', array(
    'public_dir' => __DIR__,
));

// Start capturing all output excluding headers and write to public
directory

$capture->start();
// Don't forget to change HTTP response code
header('Status: 200', true ,200);
// Code below to
```

Configuration Optimizations

Template Maps

`Zend\View` and its view manager use a stack of paths similar to the `include_path` of PHP. This may be very convenient while developing your application - not needing to add a reference to every template file as you add them – but this does come with the added overhead of checking each path for the existence of a matching template every time a template is loaded.

The alternative to stack scanning is to use the template map of the view manager. This will map a template to a specific file on the file system, removing the need to scan a stack for the existence of a file. This is as quick as a simple array access.

You configure a simple template map in your module's config as follows:

```
return array(
    // your other module settings
    'view_manager' => array(
        'template_map' => array(
            'info/layout'               => __DIR__  .
'/../view/layout/my.phtml',
            'info/index/index'          => __DIR__ .
'/../view/info/index/index.phtml',
            'info/index/about'          => __DIR__ .
'/../view/info/index/about.phtml',
            'info/entertainment/albums' => __DIR__ .
'/../view/info/entertainment/albums.phtml',
        )
    )
);
```

Config Caching

With each request to your application, all of the config paths will be scanned, and all of the configurations found merged into one large configuration array. This can be a costly operation, particularly as you add more modules to your application.

To optimize this operation, cache the merged configurations into a single array, using a setting from the `ModuleManager`. You can do this by simply adding the following configuration to `config/application.config.php`:

```
return array(
    /* ... */
    'module_listener_options' => array(
    'config_cache_enabled'    => true,
        'module_map_cache_enabled' => true,
        'cache_dir'                => 'data/cache/',
    ),
    /* ... */
);
```

This will create a cache in `data/cache` in our applications root directory.

There is one important point to bear in mind when using a config cache. Factories provided in your config as closures will **not** be cacheable, as PHP cannot serialize them. Instead, add these configs via the `getXxxConfig` methods in your Module's `Module` class.

Autoloader Optimization

Zend Framework 2 supplies a classmap generator that can be used to create a lookup of class name to filesystem location, giving a direct path from which to load a given class. This avoids scanning multiple directories or using the include path.

The classmap produced by the classmap generator can be used with the `Zend\Loader\ClassMapAutoloader` component. For usage examples for this component, please see the Zend Framework 2 manual.

To run the classmap generator, use the PHP script supplied in the `bin` directory of the Zend Framework 2 distribution (or `vendor/bin` if you have installed with Composer).

```
php classmap_generator.php modules
```

The above example would create `modules/autoload_classmap.php`, which can be added to your classmap autoloader.

Test Your Knowledge: Questions

1

Within `Zend\ModuleManager\Listener\ListenerOptions`, you can activate configuration caching. What parts of the application will be cached when turning on all possible options? (Choose 2)

- The configuration from the `config/application.config.php` file
- The module map with all activated modules
- The merged configuration from all modules, plus configuration discovered on the configuration path `config/autoload/`
- All file assets (images, CSS files, JS files) from all activated modules

2

If you register a standard autoloader and a classmap autoloader, which behavior is provided by default to improve performance?

- Autoloaders will be registered in the order you defined
- The standard autoloader will always be prepended on the autoloader stack
- Classmap autoloader will always be prepended on the autoloader stack
- The PHP `include_path` is always used to find a matching class

Test Your Knowledge: Answers

1

Within `Zend\ModuleManager\Listener\ListenerOptions`, you can activate configuration caching. What parts of the application will be cached when turning on all possible options? (Choose 2)

- The configuration from the `config/application.config.php` file
- The module map with all activated modules
- The merged configuration from all modules, plus configuration discovered on the configuration path `config/autoload/`
- All file assets (images, CSS files, JS files) from all activated modules

2

If you register a standard autoloader and a classmap autoloader, which behavior is provided by default to improve performance?

- Autoloaders will be registered in the order you defined
- The standard autoloader will always be prepended on the autoloader stack
- Classmap autoloader will always be prepended on the autoloader stack
- The PHP `include_path` is always used to find a matching class

SECURITY TOPIC OVERVIEW

- One of the most important rule to follow when you develop web applications is: "Filter input, escape output"; most web security issues because this rule was not followed

- Cryptography is a great tool to protect sensitive information but you need to know how to use it – for example, which algorithm to use to encrypt the data; provide authentication and data integrity over encryption; how to store the encryption key

- If you want to develop secure web applications you cannot just use the right tool; you need to design your secure workflow, according to your needs and use case

Security

In order to protect a web application, you need to know your potential enemy and understand the types of attacks they can use against your application. Web security is dynamic - something that is valid today can be outdated tomorrow. Therefore, it is very important to stay up-to-date with the last potential vulnerabilities.

A good resource is the OWASP web site, **https://www.owasp.org**, a community project that collects best practices and information about web security. Every three years, they produce a top ten list of the most serious security flaws in web apps.

OWASP Top Ten List

1. Injection

Injection flaws, such as SQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.

2. Broken Authentication and Session Management

Application functions related to authentication and session management are often not implemented correctly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities.

3. Cross-Site Scripting (XSS)

XSS flaws occur whenever an application takes untrusted data and sends it to a web browser without proper validation or escaping. XSS allows attackers to execute scripts in the victim's browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.

4. Insecure Direct Object References

A direct object reference occurs when a developer exposes a reference to an internal implementation object, such as a file, directory, or database key. Without an access control check or other protection, attackers can manipulate these references to access unauthorized data.

5. Security Misconfiguration

Good security requires having a secure configuration defined and deployed for the application, frameworks, application server, web server, database server, and platform. Secure settings should be defined, implemented, and maintained, as defaults are often insecure. Additionally, software should be kept up to date.

6. Sensitive Data Exposure

Many web applications do not properly protect sensitive data, such as credit cards, tax IDs, and authentication credentials. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data deserves extra protection such as encryption at rest or in transit, as well as special precautions when exchanged with the browser.

7. Missing Function Level Access Control

Most web applications verify function level access rights before making that functionality visible in the UI. However, applications need to perform the same access control checks on the server when each function is accessed. If requests are not verified, attackers will be able to forge requests in order to access functionality without proper authorization.

8. Cross-Site Request Forgery (CSRF)

A CSRF attack forces a logged-on victim's browser to send a forged HTTP request, including the victim's session cookie and any other automatically included authentication information, to a vulnerable web application. This allows the attacker to force the victim's browser to generate requests the vulnerable application thinks are legitimate requests from the victim.

9. Using Components with Known Vulnerabilities

Components, such as libraries, frameworks, and other software modules, almost always run with full privileges. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications using components with known vulnerabilities may undermine application defenses and enable a range of possible attacks and impacts.

10. Unvalidated Redirects and Forwards

Web applications frequently redirect and forward users to other pages and websites, and use untrusted data to determine the destination pages. Without proper validation, attackers can redirect victims to phishing or malware sites, or use forwards to access unauthorized pages.

Session Security

PHP session variables store information about individual users during a session, and make this data available to all pages within the web application. These variables can also be used to change session settings.

A session in PHP is referenced using a special cookie variable named PHPSESSID, by default. There are many kinds of attacks on web-based sessions. For instance, Session Hijacking compromises the session token by stealing or predicting a valid session token to gain unauthorized access to the Web Server. Session Fixation is when an attacker takes advantage of a shared computer, capturing the session identifier before it is used when a user logs into a web application, and later using this to assume the identity of the victim.

Session Hijacking is particularly difficult to protect against since an IP address may change from the end user depending on their ISP, or a browser's user agent may change during the request, either by a web browser extension or an upgrade that retains session cookies. Session validators help to provide various protections against session hijacking.

Zend Framework 2 Components

ZF2 offers two basic validators for sessions: `Http User Agent` and `Remote Addr`. You can also provide a custom session validator.

Http User Agent

Zend provides a validator to check the session against the originally stored `$_SERVER['HTTP_USER_AGENT']` variable. Validation will fail in the event that this does not match and throws an exception in Zend after `session_start()` has been called.

Basic Usage:

```
use Zend\Session\Validator\HttpUserAgent;
use Zend\Session\SessionManager;

$manager = new SessionManager();

$manager->getValidatorChain()->attach(
    'session.validate',
    array(new HttpUserAgent(), 'isValid')
);
```

Remote Addr

ZF2 provides a validator to check the session against the originally stored `$SERVER['REMOTE_ADDR']` variable. Validation will fail whenever this does not match and will throw an exception after `session_start()` has been called.

Basic Usage:

```
use Zend\Session\Validator\RemoteAddr;
use Zend\Session\SessionManager;

$manager = new SessionManager();
$manager->getValidatorChain()->attach(
    'session.validate',
    array(new RemoteAddr(), 'isValid')
);
```

Cryptography

Cryptography is the study of hidden messages and the practice of how to protect information. It is a multidisciplinary science that involves mathematics, computer science, linguistics, software engineering and more. It protects information using special algorithms to transform a message in an unreadable format. Only the owner of the message, who knows the secret of the transformation, can read the original message.

Cryptography doesn't ensure security. Frequently, cryptography is wrongly associated with security - if we hear the word cryptography, we think that the system is secure by default. That is not true. Cryptography is only an instrument that can help to improve the security of a system. At the same time, if you don't know how to use this tool, you can do a lot of damage to the security of a system, instead of improving it – for example, using a weak algorithm to encrypt information, or a very simple password as the key for a cipher.

Encryption / Decryption

The message to protect is usually called the *plaintext* and the protected message is called the *ciphertext*. The process of transforming a message from plaintext to ciphertext is called Encryption. The opposite procedure, from ciphertext to plaintext, is called Decryption. The security of a message is related to the knowledge of a key, which contains secret information needed for the conversion. This means that the management of the key is one of the most important aspects for the security of the encrypted data.

Kerckhoffs’ Principle (Kerckhoffs’ Law)

"A cryptosystem should be secure even if everything about the system, except the key, is public knowledge." Kerckhoffs' principle (*from Wikipedia.org*)

Some important aspects around the security of a cryptographic system:

- Never use closed source algorithms... if you want to implement a cryptographic system that is reasonably secure, you should always use standard algorithms
- Don't spend time and energy to create a new cipher... if you are not an expert in cryptography, you will most likely fail
- One of the most used standards in cryptography today is the **Advanced Encryption Standard**... this algorithm won the NIST competition in 2001 after five years of challenging with fifteen different designs

Because cryptography is a very difficult process to implement, ZF2 contains a set of components that can help developers implement cryptography in the correct way. The available features are:

- encrypt-then-authenticate using symmetric ciphers (the authentication step is provided using HMAC)
- encrypt/decrypt using symmetric and public key algorithm (ex: RSA algorithm)
- generate digital sign using public key algorithm (ex: RSA algorithm)
- key exchange using the Diffie-Hellman method
- Key derivation function (e.g. using PBKDF2 algorithm);
- Secure password hash (e.g. using Bcrypt algorithm);
- generate Hash values;
- generate HMAC values;

ZF2 uses the `Mcrypt` extension of PHP for the symmetric algorithms (ex: AES) and the `OpenSSL` extension for the public key algorithms (ex: RSA).

For the secure password storage, ZF2 uses the `bcrypt` implementation of PHP available in the `crypt()` function since PHP 5.3.0.

Symmetric-Key and Public-Key Cryptography

There are two basic types of encryption algorithms: Symmetric-key and Public-key.

Symmetric Key Cryptography

Symmetric-key algorithms use a single key to encrypt and to decrypt data. Therefore, the secret key has to be shared among users, which can be a problem when employing insecure communication channels, such as the Internet.

Even though symmetric ciphers have this problem, they are the most widely used algorithms to encrypt data because they are very fast compared with the asymmetric algorithms. A commonly-used alternative process is a hybrid method, in which asymmetric algorithms are used to encrypt the session key used by symmetric algorithms to encrypt data.

Symmetric-key ciphers are implemented as either *block* ciphers or *stream* ciphers. A block cipher enciphers plaintext in blocks, while stream ciphers process input as individual characters.

In ZF2, Symmetric-key encryption is implemented using `Zend/Crypt/Symmetric` sub-components and `Mcrypt` extensions to support the following algorithms:

- AES (rijndael-128)
- Blowfish
- DES
- 3DES
- CAST-128/256
- rijndael-128/192/256
- Saferplus
- Serpent
- Twofish

For general use cases, the `Zend\Crypt\BlockCipher` class supports encryption plus authentication. The authentication process is very important in prevent Padding Oracle Attacks.

`Zend\Crypt\BlockCipher` implements its encrypt-then-authenticate mode using HMAC to provide authentication.

The symmetric cipher can be chosen with a specific adapter that implements the `Zend\Crypt\Symmetric\SymmetricInterface`. We support the standard algorithms of the Mcrypt extension. The adapter that implements the Mcrypt is `Zend\Crypt\Symmetric\Mcrypt`.

The example that follows shows how to use the `BlockCipher` class to encrypt-then-authenticate a string using the AES block cipher (with a key of 256 bit) and the HMAC algorithm (using the SHA-256 hash function).

```
use Zend\Crypt\BlockCipher;

$blockCipher = BlockCipher::factory('mcrypt', array('algo' => 'aes'));
$blockCipher->setKey('encryption key');
$result = $blockCipher->encrypt('this is a secret message');

echo "Encrypted text: $result \n";
```

The `BlockCipher` is initialized using a factory method with the name of the cipher adapter to use (`mcrypt`) and the parameters to pass to the adapter (the AES algorithm). In order to encrypt a string we need to specify an encryption key and we used the `setKey()` method for that scope. The encryption is provided by the `encrypt()` method.

The output of the encryption is a string, encoded in Base64 (default). It contains the HMAC value, the IV (Initialization Vector), and the encrypted text. The encryption mode used is the CBC (with a random IV by default) and SHA256 as the default hash algorithm of HMAC. The Mcrypt adapter encrypts using the PKCS#7 padding mechanism by default. You can specify a different padding method using a special adapter for that (`Zend\Crypt\Symmetric\Padding`). The encryption and authentication keys used by the `BlockCipher` are generated with the PBKDF2 algorithm, used as key derivation function from the user's key specified using the `setKey()` method.

You can change all the default settings passing the values to the factory parameters. For instance, if you want to use the Blowfish algorithm, with the CFB mode and the SHA512 hash function for HMAC you have to initialize the class as follow:

```
use Zend\Crypt\BlockCipher;

$blockCipher = BlockCipher::factory(
    'mcrypt',
    array(
        'algo' => 'blowfish',
        'mode' => 'cfb',
        'hash' => 'sha512'
    )
);
```

To decrypt a string we can use the `decrypt()` method. To successfully decrypt a string, configure the `BlockCipher` with the same parameters of the encryption.

Public Key Cryptography

Public-key algorithms use two keys: one to encrypt (public key) and one to decrypt (private key). The public key can be sent in plaintext, and if someone intercepts the key, s/he cannot use it to decrypt the message. It is nearly impossible to obtain the private key from the public key. Public-key algorithms are typically used in communication systems because they can be used by remote users without the need to share secret information (the key).

Zend Framework 2 offers two Public-key algorithms: **Diffie-Hellman** key exchange and **RSA**.

Diffie-Hellman

The Diffie-Hellman algorithm is a specific method of exchanging cryptographic keys. This key exchange method allows two parties with no prior knowledge of each other to jointly establish a shared secret key over an insecure communications channel. This key can then be used to encrypt subsequent communications using a symmetric key cipher.

The parameters of the Diffie-Hellman class are: a prime number (p), a generator (g) that is a primitive root mod p, and a private integer number. The security of the Diffie-Hellman exchange algorithm is related to the choice of these parameters. To know how to choose secure numbers, consult the RFC 3526 document.

RSA

The RSA algorithm can be used to encrypt/decrypt message and also to provide authenticity and integrity generating a digital signature of a message. RSA is based on the assumed difficulty of factoring large integers, the **factoring problem**. A user of RSA creates and then publishes the product of two large prime numbers, along with an auxiliary value, as their public key. The prime factors must be kept secret. Anyone can use the public key to encrypt a message, but with currently published methods, if the public key is large enough, only someone with knowledge of the prime factors can feasibly decode the message. Whether breaking RSA encryption is as hard as factoring is an open question known as the “RSA problem”.

Usage examples of the `Zend\Crypt\PublicKey\Rsa` class include:

- To generate a public key and a private key
- To encrypt/decrypt a string
- To generate a digital signature of a file

This example generates a public and private key of 2048 bit, storing the keys in two separate files, the `private_key.pem` for the private key and the `public_key.pub` for the public key.

```
use Zend\Crypt\PublicKey\RsaOptions;

$rsaOptions = new RsaOptions(array(
    'pass_phrase' => 'test'
));

$rsaOptions->generateKeys(array(
    'private_key_bits' => 2048,
));

file_put_contents('private_key.pem', $rsaOptions->getPrivateKey());
file_put_contents('public_key.pub', $rsaOptions->getPublicKey());
```

Encrypt and Decrypt a String with RSA

The code example below demonstrates how to encrypt and decrypt a string using the RSA algorithm. You can encrypt only small strings - the maximum size of encryption is given by the length of the public/private key. For instance, if we use a size of 2048 bit, you can encrypt a string with a maximum size of 1960 bit (245 characters). This limitation is related to the security measures of the OpenSSL implementation.

The normal application of a public key encryption algorithm is to store a key (or a hash) of the data you want to encrypt (or sign). A hash is typically 128-256 bits (the PHP sha1() function returns a 160 bit hash). An AES encryption key is 128 to 256 bits. Either of those will fit comfortably inside a single RSA encrypted message.

```
use Zend\Crypt\PublicKey\Rsa;

$rsa = Rsa::factory(array(
    'public_key'    => './public_key.pub',
    'private_key'   => './private_key.pem',
    'pass_phrase'   => 'test',
    'binary_output' => false
));

$text = 'This is the message to encrypt';

$encrypt = $rsa->encrypt($text);
printf("Encrypted message:\n%s\n", $encrypt);

$decrypt = $rsa->decrypt($encrypt);
if ($text !== $decrypt) {
    echo "ERROR\n";
} else {
    echo "Encryption and decryption performed successfully!\n";
}
```

Generate a digital signature of a file

Here is an example of how to generate a digital signature of a file:

```
use Zend\Crypt\PublicKey\Rsa;

$rsa = Rsa::factory(array(
    'private_key'   => 'path/to/private_key',
    'pass_phrase'   => 'passphrase of the private key',
    'binary_output' => false
));

$file = file_get_contents('path/file/to/sign');

$signature = $rsa->sign($file, $rsa->getOptions()->getPrivateKey());

$verify = $rsa->verify($file, $signature, $rsa->getOptions()-
>getPublicKey());

if ($verify) {
    echo "The signature is OK\n";
    file_put_contents($filename . '.sig', $signature);
    echo "Signature save in $filename.sig\n";
} else {
    echo "The signature is not valid!\n";
}
```

This example uses the Base64 format to encode the digital signature of the file (`binary_output` is false).

The signature, using the `sign()` function, is generated in two steps:

- Generating the hash of the content (using `SHA1` as default algorithm)
- Encrypting the hash with the private key of the owner

The signature is generated using `openssl_sign()` function of the OpenSSL extension.

Hash Functions

As discussed earlier, encryption algorithms protect sensitive information by transforming plaintext into ciphertext. To protect data against an accidental or intentional change, you can use a special family of algorithms: hash functions.

Hash functions take an arbitrary block of data and return a fixed-size bit string, the hash value (sometimes called the *message digest* or *digest*). These functions are used in different computer science fields; for instance, they are used with the hash table data structure to calculate the key of a value.

Password Hashing

In the `Zend\Crypt\Password` namespace you can find all the password formats supported by Zend Framework 2, currently:

- bcrypt;
- Apache (htpasswd)

Best practice recommends using the bcrypt algorithm for storing a user's password. It is considered secure against brute forcing attacks (see the details below).

Bcrypt

The **bcrypt** algorithm is a widely-used hashing algorithm. Classic hashing mechanisms like MD5 or SHA, with or without a *salt* value, are not considered secure anymore (**read this post to understand why**).

The security of bcrypt is related to the speed of the algorithm. Bcrypt is very slow - it can require even a second to generate an hash value. That means a brute force attack is impossible to execute, due to the amount of time required for every attempt.

Bcrypt uses a *cost* parameter that specifies the number of cycles to use in the algorithm. By increasing this number, the algorithm will spend more time generating the hash output. The *cost* parameter is represented by an integer value from 4 - 31. The default *cost* value of the `Zend\Crypt\Password\Bcrypt` is 10, which corresponds to almost 0.2 seconds using a CPU Intel i5 at 3.3 Ghz (the *cost* parameter is a relative value according to the speed of the CPU used).

To change the *cost* parameter of the bcrypt algorithm, use the `setCost()` method. Note that if you change the cost parameter, the resulting hash will be different, but this will not affect the verification process of the algorithm, therefore not breaking the password hashes you already have stored. Bcrypt reads the *cost* parameter from the hash value, during the password authentication. All of the parts needed to verify the hash are all together, separated with the $ character - first the algorithm, then the cost, the salt, and finally the hash.

The example below shows how to use the bcrypt algorithm to store a user's password:

```
use Zend\Crypt\Password\Bcrypt;

$bcrypt = new Bcrypt();
$securePass = $bcrypt->create('user password');
```

The return value of the `create()` method is the hash of the password. This value can then be stored in a repository like a database (the output is a string of 60 bytes).

To verify if a given password is valid against a bcrypt value you can use the `verify()` method. An example is shown below:

```
use Zend\Crypt\Password\Bcrypt;

$bcrypt = new Bcrypt();
$securePass = 'the stored bcrypt value';
$password = 'the password to check';

if ($bcrypt->verify($password, $securePass)) {
    echo "The password is correct! \n";
} else {
    echo "The password is NOT correct.\n";
}
```

The bcrypt algorithm uses a *salt* value to improve the randomness of the algorithm. By default, the `Zend\Crypt\Password\Bcrypt` component generates a random salt for each hash. If you want to specify a preselected salt, can use the `setSalt()` method.

ZF2 also provides a `getSalt()` method to retrieve the *salt* specified by the user. The *salt* and the *cost* parameter can be specified during the constructor of the class, as shown below:

```
use Zend\Crypt\Password\Bcrypt;

$bcrypt = new Bcrypt(array(
    'salt' => 'random value',
    'cost' => 13
));
```

Apache

ZF2 supports all the password formats used by **Apache** (`htpasswd`). These formats are:

- CRYPT… uses the traditional Unix `crypt(3)` function with a randomly-generated 32-bit salt (only 12 bits used) and the first 8 characters of the password
- SHA1… "`{SHA}`" + Base64-encoded SHA-1 digest of the password
- MD5… “`apr1`” + the result of an Apache-specific algorithm using an iterated (1,000 times) MD5 digest of various combinations of a random 32-bit salt and the password
- Digest… the MD5 hash of the string `user:realm:password` as a 32-character string of hexadecimal digits; `realm` is the Authorization Realm argument to the `AuthName` directive in `httpd.conf`

To specify the format of the Apache password, use the `setFormat()` method. An example with all the formats is presented below:

```
use Zend\Crypt\Password\Apache;

$apache = new Apache();

// CRYPT
$apache->setFormat('crypt');
printf ("CRYPT output: %s\n", $apache->create('password'));

// SHA1
$apache->setFormat('sha1');
printf ("SHA1 output: %s\n", $apache->create('password'));

// MD5
$apache->setFormat('md5');
printf ("MD5 output: %s\n", $apache->create('password'));

// DIGEST
$apache->setFormat('digest');
$apache->setUserName('enrico');
$apache->setAuthName('test');
printf ("Digest output: %s\n", $apache->create('password'));
You can also specify the format of the password during the constructor of
the class:
use Zend\Crypt\Password\Apache;

$apache = new Apache(array(
    'format' => 'md5'
));
```

Other possible parameters to pass in the constructor are username and authname, for the digest format.

Test Your Knowledge: Questions

1

What is Cross-Site Scripting (XSS)? (Choose Two)

- An attack based on CSS malicious code injection
- An attack based on injecting client-side scripts into web pages
- An attack performed if you do not filter the input data coming from the user
- An attack performed if you do not escape the output from your application

2

Why is the bcrypt algorithm considered secure to store user's password? (Choose Two)

- Because it's based on the Blowfish encryption algorithm
- Because it's an encryption algorithm rather than a simple hash function like MD5 or SHA1
- Because it's slow, very slow and this prevent brute force attacks
- Because it's fast, very fast and this prevent dictionary attacks

Test Your Knowledge: Answers

1

What is Cross-Site Scripting (XSS)? (Choose Two)

- An attack based on CSS malicious code injection
- An attack based on injecting client-side scripts into web pages
- An attack performed if you do not filter the input data coming from the user
- An attack performed if you do not escape the output from your application

2

Why is the bcrypt algorithm considered secure to store user's password? (Choose Two)

- Because it's based on the Blowfish encryption algorithm
- Because it's an encryption algorithm rather than a simple hash function like MD5 or SHA1
- Because it's slow, very slow and this prevent brute force attacks
- Because it's fast, very fast and this prevent dictionary attacks

DATABASE TOPIC OVERVIEW

- The model within the MVC design pattern is the part that deals with the application's core business purpose, or business rules (often called *business logic*). This often requires interaction with a database.

- `Zend\Db` and its related classes provide a simple SQL database interface for Zend Framework 2.

- **`Zend\Db\Adapter\Adapter`** is used for performing operations on an underlying RDBMS platform. It provides a bridge from vendor-specific PHP extensions to a common interface, so that the PHP application can be written once to multiple RDBMS platforms with little effort.

Zend\Db

Adapters

The Adapter object provides a common interface to all supported RDMBS PHP extensions and vendor databases, making it the most important sub-component.

The adapter creates an abstraction layer (the **Driver** portion) for the PHP extensions. It also creates a lightweight abstraction layer, the **Platform** portion, for the various idiosyncrasies that each vendor-specific platform might have in its SQL/RDBMS implementation.

Creating an Adapter: Instantiation

An adapter can be created by instantiating the **`Zend\Db\Adapter\Adapter`** class. The most common use case is to pass a configuration array to the adapter. The key-value pairs that should be included in this array can be found in the ZF2 online manual **here**.

```
$adapter = new \Zend\Db\Adapter\Adapter(array(
    'driver' => 'Pdo',
    'dsn' => 'mysql:dbname=mydb;host=127.0.01',
    'username' => 'cat',
    'password' => 'dog'
));
```

Creating an Adapter: Dependency Injection

For more flexibility, it is possible to manually create an instance of **`Zend\Db\Adapter\Adapter`**. Using the constructor arguments can give you more control in creating an adapter, for example when you wish to override the default classes used (see more information **here**).

Queries

You can use the `query()` method of `Zend\Db\Adapter\Adapter` to directly execute a SQL query. When using `query()`, the preferred way to process SQL statements is to supply a SQL statement with placeholders substituting for values (preparation), and then the parameters for those placeholders are supplied separately. `query()` is useful for one-off queries and the quick querying of a database through an adapter.

```
$adapter->query('SELECT * FROM `artist` WHERE `id` = ?', array(5));
```

If a statement has to be executed directly because preparation will not work (as when executing a DDL statement in some vendor platforms), you need to provide a hint to the method:

```
$adapter->query('ALTER TABLE ADD INDEX(`foo_index`) ON (`foo_column`)',
Adapter::QUERY\_MODE\_EXECUTE);
```

Driver

The **Driver** is where Zend\Db\Adapter\Adapter implements the connection level abstraction, making it possible to use all of Zend's interfaces via the PDO (and other PHP level drivers). To make this possible, each driver is composed of 3 objects:

- A connection: **`Zend\Db\Adapter\Driver\ConnectionInterface`**
- A statement: **`Zend\Db\Adapter\Driver\StatementInterface`**
- A result: **`Zend\Db\Adapter\Driver\ResultInterface`**

Platform

The Platform object provides an API that assists in crafting queries specific to the SQL implementation of a particular vendor. Nuances such as how identifiers or values are quoted, or what the identifier separator character is are handled by this object.
While you can instantiate your own Platform object, it is generally easier to get the proper Platform instance from the configured adapter. By default the platform type will match the underlying driver implementation:

```
$platform=$adapter->getPlatform();
// or
$platform=$adapter->platform; // magic property access
```

Platform objects should always implement the **`Zend\Db\Adapter\Platform\PlatformInterface`**.

StatementContainer

The **StatementContainer** object implements the **StatementContainerInterface** and is responsible for storing the driver specific version of a SQL statement. While you can create StatementContainer objects directly, the recommended way is via the createStatement() or query() methods of the Adapter object.

```
$statement = $adapter->createStatement($sql);
```

Note: Other classes have methods that can create StatementContainer objects which will be covered later.

The **Zend\Db\Adapter\ParameterContainer** object is a container for parameters that need to be passed into a StatementContainer object to supply all the parameterized parts of the SQL statement. This object implements the ArrayAccess interface. In addition to handling parameter names and values, the container can also assist in tracking parameter types for PHP to SQL type handling.

ResultSet

Zend\Db\ResultSet is used for abstracting results of queries into an object that can be iterated over to produce individual rows. ResultSet objects implement the Zend\Db\ResultSet\ResultSetInterface.

Zend Framework 2 ships with a pre-build ResultSet -- **Zend\Db\ResultSet\ResultSet.** This ResultSet extends the **Zend\Db\ResultSet\AbstractResultSet** abstract class and is useful when it is needed to present result data in a simple ArrayObject format.

```
$sql = "SELECT * FROM products WHERE color = 'red'";
$statement = $adapter->createStatement($sql);
$statement->prepare();
$result = $statement->execute();

// create the Resultset and iterate over it
if ($result instanceof ResultInterface && $result->isQueryResult()) {
    $resultSet = new ResultSet();
    $resultSet->initialize($result);
    foreach ($resultSet as $row) {
        echo $row->name . PHP_EOL;
    }
}
```

Hydrating Result Set

`Zend\Db\ResultSet\HydratingResultSet` is a flexible `ResultSet` object that allows the developer to choose an appropriate "hydration strategy" for getting row data into a target object.

While iterating over results, `HydratingResultSet` takes a prototype of a target object and clones it once for each row. The `HydratingResultSet` will then hydrate that clone with the row data.

(See the ZF2 manual section on `Zend\Stdlib\Hydrator` to get a better sense of the different strategies that can be employed to populate a target object.)

Zend\Db\Sql

Zend\Db\Sql is a SQL abstraction layer for building platform-specific SQL queries via an object-oriented API. A Zend\Db\Sql object either produces a Statement and Parameter container that represents the target query, or produces a full string that can be directly executed against the database platform.

Zend\Db\Sql objects require a Zend\Db\Adapter\Adapter object to run the query against.

There are four primary tasks associated with interacting with a database: select, insert, update and delete. A Zend\Db\Sql\Sql has a method for each of these primary tasks, which returns a corresponding object when invoked:

- $sql->select() returns a Zend\Db\Sql\Select object
- $sql->insert() returns a Zend\Db\Sql\Insert object
- $sql->update() returns a Zend\Db\Sql\Update object
- $sql->delete() returns a Zend\Db\Sql\Delete object

Once the required Sql object has been generated (and populated with the correct values), the object can then be prepared and executed.

```
$sql = new \Zend\Db\Sql\Sql($adapter);
$select = $sql->select()
    ->from('products')
    ->where(array('type' => 'Hat'));

$preparedSelect = $sql->prepareStatementForSqlObject($select);
$resultSet = $preparedSelect->execute();
```

The Zend\Db\Sql objects (Select, Insert, Update, Delete) implement two interfaces, **PreparableSqlInterface** and **SqlInterface**.

Zend\Db\TableGateway

The **Zend\Db\TableGateway** is a useful way to execute the most common and simple queries against a single table.

- select() for running SELECT statements
- update() for running UPDATE queries
- insert() for running INSERT queries
- delete() for running DELETE queries

TableGateway objects should implement the Zend\Db\TableGateway\TableGatewayInterface interface, and generally will be instances of Zend\Db\TableGateway\TableGateway, which is a concrete implementation of the AbstractTableGateway that implements the methods defined in the interface, and is ready to use out-of-the-box.

To create a TableGateway pass in the required constructor arguments.

```
$productTable = new TableGateway('products', $adapter);
```

$productTable is now ready to be queried against using the methods mentioned above.

```
// select
$resultSet = $productTable->select(array('color' => 'red'));
// insert
$productTable->insert(array(
    'name' => 'ElePHPant',
    'color' => 'yellow',
    'price' => '$29.99'
));
// update
$productTable->update(array(
    'price' => '$24.99'
), array(
    'color' => 'blue'
));
// delete
$productTable->delete(array(
    'color' => 'red'
));
```

TableGateway objects can also be created by passing the optional parameters $features and $resultSetPrototype.

- The $features parameter accepts an instance or array of objects that extend the **Feature\AbstractFeature** abstract class. This allows you to easily connect to a TableGateway with more complex connections, such as Master/Slave databases. The list of available features can be found under the **Zend\Db\TableGateway\Feature** namespace.
- The $resultSetPrototype parameter allows you to pass in a pre-configured ResultSet object (that implements the ResultSetInterface interface). This is especially useful for using the HydratingResultSet as mentioned above.

Zend\Db\RowGateway

Zend\Db\RowGateway is a component of Zend\Db that implements the Row Gateway pattern. The Row Gateway maps a single row of the database into an entity object. The object can then be modified and the save() method can be called to persist the modified row back to the table. The delete() method can be used to remove the row completely. RowGateway objects should implement the **RowGatewayInterface**, and typically will extend the **AbstractRowGateway** abstract class.

While RowGateway objects will generally be generated through the TableGateway interface by using the **RowGatewayFeature** feature (as described above).

```
$table= new TableGateway('product', $adapter, RowGatewayFeature('id'));
$resultSet = $table->select(array('color' => 'yellow'));

$rowGateway = $results->current();
$rowGateway->price = '$49.99';
$rowGateway->save();
```

It is also possible to generate RowGateway objects directly by running queries against the Adapter, creating a new RowGateway object manually, and using the **populate()** method.

```
$sql = "SELECT * FROM products WHERE color = 'red'";
$rowArray = $adapter->query($sql)->current()->getArrayCopy();

$rowGateway = new RowGateway('id', 'products', $adapter);
$rowGateway->populate($rowArray);

// we have our rowGateway, delete the row
$rowGateway->delete();
```

Zend\Db\Metadata

Zend\Db\Metadata makes it possible to retrieve metadata about the database, table, column, index or constraints in an object-orientated manner.
Zend\Db\Metadata\Metadata is a concrete implementation of the **Zend\Db\Metadata\MetadataInterface** interface that can be used to produce metadata descriptions.

```
$metadata = new Metadata($adapter);
// show table names
$tables = $metadata->getTableNames();
// show table description
$products = $metadata->getTable('products');
```

Data Definition Language

Zend\Db\Sql\Ddl, is used to create statement objects that produce DDL (Data Definition Language) SQL statements. When combined with a platform-specific Zend\Db\Sql\Sql object, these DDL objects are capable of producing statements with specialized data types, constraints, and indexes for a database/schema.

Creating Tables

Like Zend\Db\Sql objects, each statement type is represented by a class. For example, CREATE TABLE is modelled by a **CreateTable** object; similarly for ALTER TABLE (as **AlterTable**) and DROP TABLE (as **DropTable**). These classes all implement the **Zend\Db\Sql\Dbl\SqlInterface** (which in turn implements the **Zend\Db\Sql\SqlInterface**).

```
// either
$tableDdl = new CreateTable();
$tableDdl->setTable('products');

// or
$tableDdl = new CreateTable('products');

// or create a temporary table with optional 2nd parameter
$tableDdl = new CreateTable('products', true);
```

You can manually add new columns (or constraints) to the DDL by using the addColumn() (or addConstraint()) methods:

```
// add a new int column called type_id
$tableDdl->addColumn(new Column\Integer('type_id');


// add a new foreign key to the id column of the type table
$tableDdl->addConstraint(new Constraint\ForeignKey('fk-type_id', 'type_id',
                                                   'type', 'id');
```

Altering Tables

You can modify tables using the [AlterTable] object which is created in the same way as CreateTable.

You can also use the dropColumn() and dropConstraint() methods in a similar way as shown above.

```
$alterTableDdl->dropConstraint('fk-type_id');
$alterTableDdl->dropColumn('type_id');
```

Dropping Tables

To drop an entire table you can use the DropTable class.

```
$dropTableDdl = new DropTable('products');
```

Executing DDL Statements

Note: Creating a DDL object such as AlterTable does not run the statement created, this section will explain how to execute a created DDL object.

Once a DDL statement object has been created and configured, you can execute the statement using two objects: an Adapter instance, and a properly created Sql instance.

```
$adapter->query(
    $sql->getSqlStringForSqlObject($createTableDdl),
    Adapter::QUERY_MODE_EXECUTE
);
```

First, we pass the previously created CreateTable DDL object to the getSqlStringForSqlObject() method of the SQL object. This ensures that the statement is generated with any platform specific modifications.

We can then run the generated statement using the query() method of the adapter. The is no need to prepare the statement (this can sometimes result in un-executable queries on some platforms), so we use the QUERY_MODE_EXECUTE constant to execute without preparation.

Test Your Knowledge: Questions

1

Which statement is incorrect?

- I have to make sure that all parameters are quoted for each database request when using Zend\Db\Adapter
- All Zend\Db components quote all parameters for each database request automatically
- All Zend\Db\Sql components quote all parameters automatically
- An Zend\Db\TableGateway instances quote all parameters automatically

2

What will the following statement produce?

```
<php>
$sql = new Zend\Db\Sql\Select('tablename');

$sql->where(new Zend\Db\Sql\Predicate\In('tablename.foo',
array(1, 2)));

var_dump($sql->getSqlString());
</php>
```

- SELECT "tablename".* FROM "tablename" WHERE "tablename"."foo" IN ('1', '2')
- A warning, then: SELECT "tablename".* FROM "tablename" WHERE "tablename"."foo" IN ('1', '2')
- An exception due to missing parameter types
- An exception due to missing adapter/platform

Test Your Knowledge: Answers

1

Which statement is incorrect?

- I have to make sure that all parameters are quoted for each database request when using Zend\Db\Adapter
- All Zend\Db components quote all parameters for each database request automatically
- All Zend\Db\Sql components quote all parameters automatically
- An Zend\Db\TableGateway instances quote all parameters automatically

2

What will the following statement produce?

```
<php>
$sql = new Zend\Db\Sql\Select('tablename');

$sql->where(new Zend\Db\Sql\Predicate\In('tablename.foo',
array(1, 2)));

var_dump($sql->getSqlString());
</php>
```

- SELECT "tablename".* FROM "tablename" WHERE "tablename"."foo" IN ('1', '2')
- A warning, then: SELECT "tablename".* FROM "tablename" WHERE "tablename"."foo" IN ('1', '2')
- An exception due to missing parameter types
- An exception due to missing adapter/platform

UTILITY TOPIC OVERVIEW

- Zend Framework 2 provides a variety of utilitarian classes that can make your life easier as a developer, simplifying common tasks, enhancing security, saving boilerplate code, and making your codebase easier to test:
    - `Zend\Log`**:** A component for general purpose logging, supporting multiple log backends, formatting messages sent to the log, and filtering logged messages
    - `Zend\Mail`**:** Provides generalized functionality to compose and send text, HTML, and MIME-compliant multipart email messages
    - `Zend\Stdlib\Hydrator`**:** A component that provides mechanisms both for hydrating objects from arrays and for extracting array data sets from objects
    - `Zend\Session`**:** ZF2 expands on the built in session handling of PHP, and adds extra configuration, alternative storage implementations, and segregated session containers, with individual expiry
    - `Zend\Paginator`**:** A flexible component for paginating collections of data and presenting that data to users

Zend\Log

When logging data in your applications, you often need to decide the best way to handle the process, especially when you are writing an application to be hosted in an environment beyond your control, or when you wish to send different types of log messages to different backends. Zend Framework 2 provides a set of features that allow you to easily solve these challenges.

The Zend\Log namespace can be broken down into four key components:

- A *Logger*, which serves as a facade, composing the other components into a single interface to simplify use of the components
- *Writers*, to provide the backend, writing to a variety of mediums, including databases' email, syslog, or a stream
- *Filters*, to decide which messages passed to the logger to process
- *Formatters*, to format the data into a suitable format for writing to the log writer; each writer should have one formatter

The Logger

`Zend\Log\Logger` can be used to log PHP errors and intercept PHP Exceptions. Calling the static method `Zend\Log\Logger::registerErrorHandler($logger)` will add the `$logger` object before the current PHP error handler, and will pass the error along as well.

```
$logger = new Zend\Log\Logger();
$writer = new Zend\Log\Writer\Stream('php://output');
$logger->addWriter($writer);
Zend\Log\Logger::registerErrorHandler($logger);
```

Similarly, you can also configure a Logger to intercept `Exceptions` using the static method `Zend\Log\Logger::registerExceptionHandler($logger)`.

Creating a Log

To create a log, instantiate a Writer and then pass it to a Logger instance. A Logger must have at least one Writer. You can add any number of Writers using the Log's `addWriter()` method. You can also add a priority to each writer by specifying it as a number in the second argument of the `addWriter()` method.

The priority of a log writer indicates the order in which the writer will be triggered when a log message is written.

Logging Messages

To log a message, call the `log()` method of a Log instance and pass it the message with a corresponding priority.

```
$priority = Zend\Log\Logger::CRIT;
$message  = 'Warp core overloaded';
$logger->log($priority, $message);
```

Where `$priority` refers to the priority, following the **BSD Syslog protocol standard** (**http://tools.ietf.org/html/rfc3164**).

Each priority also approximates to the PHP `E_*` constants,

```
Zend\Log\Logger::EMERG;  // 0
Zend\Log\Logger::ALERT;  // 1
Zend\Log\Logger::CRIT;   // 2
Zend\Log\Logger::ERR;    // 3
Zend\Log\Logger::WARN;   // 4
Zend\Log\Logger::NOTICE; // 5
Zend\Log\Logger::INFO;   // 6
Zend\Log\Logger::DEBUG;  // 7
```

`$message` is the message to be logged, which will normally contain information such as where a problem occured, and why.

An array of extra options, or traversable objects, can be passed as a third parameter, which may be used by individual writers that implement support.

An alternative way is to call a method by the same name as the priority:

```
$logger->log(Zend\Log\Logger::INFO,'Informational message');
$logger->info('Informational message');
```

When the log() method is called (or one of its shortcuts), a log event is created, an associative array with data describing the event that is passed to the writers. The following keys are always created in this array: timestamp, message, priority, priorityName, and extra (which may be empty, or will contain the content of the third parameter passed to the log() method).

Writers

A Writer is an object which implements Zend\Log\Writer\AbstractWriter, and is used to record log data to a storage backend.

Writing to Files and Streams

Zend\Log\Writer\Stream sends log data to a **PHP stream**. To write log data to the PHP output buffer, use the URL php://output. Alternately, you can send log data directly to a stream like STDERR (php://stderr).

```
$writer= new Zend\Log\Writer\Stream('php://output');
$logger= new Zend\Log\Logger();
$logger->addWriter($writer);
$logger->info('Informational message');
```

We can use this to write to an application log file as follows:

```
$logfile = @fopen('data/logs/application.log', 'a', false);
if (!$logfile) {
    throw new Exception('Could not open logfile');
}

$writer = new Zend\Log\Writer\Stream($logfile);
$logger = new Zend\Log\Logger();
$logger->addWriter($writer);
$logger->info('Informational message');
```

Writing to Databases

`Zend\Log\Writer\Db` writes log information to a database table using `Zend\Db\Adapter\Adapter`. The `Zend\Log\Writer\Db` constructor receives a `Zend\Db\Adapter\Adapter`, a table name, an optional mapping of event data to database columns, and an optional string which contains the character separator for the log array.

Writing to a database, using sqlite as an illustrative example, this could be replaced with the db adapter used by your application.

```
$dbConfig = array (
  // Sqlite Configuration_
  'driver'=>'Pdo',
  'dsn'=>'sqlite:data/logs/application.sqlite.db',
);

$db     = new Zend\Db\Adapter\Adapter($dbConfig);
$writer = new Zend\Log\Writer\Db($db,'log');
$logger = new Zend\Log\Logger();
$logger->addWriter($writer);
$logger->info('Informational message');
```

Filters

A Filter processes messages passed to a log writer based upon rules specific to each type of filter. These rules may be based on priority, a regular expression match on the message, a random sample percentage of messages, or an implementation of `Zend\Log\Filter\FilterInterface`.

```
use Zend\Log\Logger;

$logger= new Logger();
$applicationLogWriter = new
Zend\Log\Writer\Stream('data/logs/messages.log');
$errorLogWriter    = new Zend\Log\Writer\Stream('data/logs/error.log');

$logger->addWriter($applicationLogWriter);
$logger->addWriter($additionalLogWriter);
```

(continued on next page...)

```
// add a filter only to errorLogWriter, this will filter all log messages
where priority is <= Zend\Log\Logger::CRIT
$filter= new Zend\Log\Filter\Priority(Logger::CRIT);
$criticalLogWriter->addFilter($filter);

// written only to $applicationLogWriter (Logger::INFO (6) is >
Logger::CRIT (2))
$logger->info('Informational message');

// written by both writers (Logger::EMERG (1) is <= Logger::CRIT (2))
$logger->emerg('Emergency message');
```

Formatters

A Formatter is an object responsible for taking event data, and returning a string with a formatted log line to be written by a writer.

Some Writers are not line-oriented and only use a specialized formatter – for example, the Database Writer, which inserts the event items directly into database columns. For Writers which cannot support a Formatter, an exception is thrown if you attempt to set a Formatter.

`Zend\Log\Formatter\Simple` is the default formatter, with a configuration equivalent to the following:

```
$format    = '%timestamp% %priorityName% (%priority%): %message%'.PHP_EOL;
$formatter = new Zend\Log\Formatter\Simple($format);
```

Zend\Mail

Zend\Mail provides generalized functionality to compose and send text, HTML, and MIME-compliant multipart email messages.

Zend\Mail\Message

A simple email consists of one or more recipients, a subject, a body and a sender (at minimum, requires at least one recipient and message body):

The Message class encapsulates a single email message, as described in RFCs **822 (http:/www.w3.org/Protocols/rfc822/)** and **2822 (http://www.ietf.org/rfc/rfc2822.txt)**.

It acts as a value object for setting mail headers and content. Multi-part email messages can be created using the Zendcomponent to create the message body, then assigning it.

```
use Zend\Mail;

$mail= new Mail\Message();
$mail->setBody('This is the text of the email.');
$mail->setFrom('sender@example.org', 'Sender\'s name');
$mail->addTo('recipient@example.com','Name of recipient');
$mail->setSubject('TestSubject');

$transport= new Mail\Transport\Sendmail();
$transport->send($mail);
```

Zend\Mail\Transport

Mail Transports perform mail delivery. There are typically two options: using PHP's native `mail()` functionality, which uses system resources to deliver mail, or using the SMTP protocol for delivering mail via a remote server. Zend Framework 2 also includes a "File" transport, which creates a mail file for each message sent; these can later be introspected as logs or consumed for the purpose of sending via an alternate transport mechanism later.

Mail can be sent using `Zend\Mail` with an implementation of `Zend\Mail\Transport\TransportInterface.` Zend Framework 2 ships with some useful transports out of the box:

- `Zend\Mail\Transport\Sendmail` to use the system Sendmail installation
- `Zend\Mail\Transport\Smtp` to send mail over a TCP connection to an SMTP server
- `Zend\Mail\Transport\File` to create a mail file on the local file system to be later inspected, or consumed by another application
- A custom transport, implementing `Zend\Mail\Transport\TransportInterface` useful for testing or development scenarios, to replace a transport in environments where sending mail does not need to be satisfied, without directly changing code in your application to accommodate the different environments

The `Zend\Mail\Transport\TransportInterface` defines exactly one method, `send()`. This method accepts a `Zend\Mail\Message` instance, which should then be introspected and serialized before sending.

Configuring the Default Sendmail Transport

Using a mail transport is typically involves instantiating it, optionally configuring it, and then passing a message to it

```
use Zend\Mail\Message;
use Zend\Mail\Transport\Sendmail as SendmailTransport;

$message= new Message();
$message->addTo('matthew@zend.com');
$message->addFrom('ralph.schindler@zend.com');
$message->setSubject('Greetings and Salutations!');
$message->setBody("Sorry, I'm going to be late today!");

$transport= new SendmailTransport();
$transport->send($message);
```

For more transport examples (SMTP, File, …), as well as config options and available methods, consult the ZF2 manual.

Zend\Stdlib\Hydrator

Hydration is the process of transposing an array of data to the properties of an object.

The Hydrator component is designed to provide mechanisms for both hydrating objects from arrays and for extracting array data sets from them.

The component consists of an interface, and several implementations for common use cases. Below is a sample Hydrator interface:

```
namespace Zend\Stdlib\Hydrator;

interface HydratorInterface
{
    /**
     * Extract values from an object
     *
     * @param object $object
     * @return array
     */
    public function extract($object);

    /**
     * Hydrate $object with the provided $data.
     *
     * @param array $data
     * @param object $object
     * @return void_
     */
    public function hydrate(array $data, $object);
}
```

Zend Framework 2 ships with the following implementations:

- ArraySerializable: will hydrate an object implementing exchangeArray or populate, and extract from an object implementing getArrayCopy
- ClassMethods: will hydrate and extract an object using setters and getters
- ObjectProperty: will hydrate and extract from public properties of an object
- Reflection: will hydrate and extract from properties of an object, using reflection

Usage:

Instantiate the Hydrator, then pass information to it:

```
use Zend\Stdlib\Hydrator;

$hydrator = new Hydrator\ArraySerializable();
$object   = new ArrayObject( array() );

$hydrator->hydrate($someData, $object);

// or, if the object has data we want as an array:
$data = $hydrator->extract($object);
```

Zend\Stdlib\Hydrator\Filter

Hydrator filters allow you to filter properties of an object that are accessible by a hydrator, preventing access to certian properties. This is useful especially when you want to `extract()` objects to the userland and strip some internals [ex: `getServiceManager()`], or when you are hydrating from user input, to prevent security flaws similar to Ruby On Rails mass assignment vulnerability.

Hydrator filters come with a Composite Implementation and filters for common use cases. The filters are implemented on the `AbstractHydrator`, so you can directly start using them if you extend it - even on custom hydrators.

```
namespace Zend\Stdlib\Hydrator\Filter;

interface FilterInterface

{
    /**
     * Should return true, if the given filter
     * does not match
     *
     * @param string $property The name of the property
     * @return bool
     */

    public function filter($property);

}
```

Hydrator filters that ship with Zend Framework 2 are:

- `Zend\Stdlib\Hydrator\Filter\FilterComposite`, which allows combining multiple filters with `AND` or `OR` conditions
- `Zend\Stdlib\Hydrator\Filter\GetFilter` filters methods that do not have a `'get'` prefix
- `Zend\Stdlib\Hydrator\Filter\IsFilter` filters methods that do not have an 'is' prefix
- `Zend\Stdlib\Hydrator\Filter\HasFilter` filters methods that do not have a `'has'` prefix
- `Zend\Stdlib\Hydrator\Filter\MethodMatchFilter` filters methods that match, or do not match a given name
- `Zend\Stdlib\Hydrator\Filter\NumberOfParameterFilter` filters methods that do not have a set number of parameters, specified in its constructor

Adding Filters to a Hydrator

You can add filters to any hydrator that implements `Zend\Stdlib\Hydrator\FilterEnabledInterface` (which is implemented by the `AbstractHydrator` class in version 2.3 or newer), by using either the `Zend\Stdlib\Hydrator\Filter\FilterInterface` or a callable:

```
$hydrator->addFilter("len", function ($property){

    if (strlen($property)!==3){
        return false;
    }

    return true;
});
```

By default, every filter is added with a conditional "or"; to see how to add with "and" with its related code, consult the **Adding Filters** section of the ZF2 manual.

Removing Filters

The ClassMethods hydrator is the only hydrator with pre-registered filters ("is", "has", "get", "parameter") that can be unregistered. The example below shows how to avoid extracting methods that start with "is".

```
use Zend\Stdlib\Hydrator\ClassMethods;

class Example {

    /**
     * @var bool
     */
    protected $booleanValue = true;

    /**
     * @var string
     */
    protected $scalarProperty = 'hello world';

    /**
     * Get the booleanValue property value
     *
     * @return bool
     */
    public function isProperty() {
        return $this->booleanValue;
    }

    /**
     * Get the scalarValue property value
     *
     * @return string
     */
    public function getScalarProperty()
    {
        return $this->scalarProperty;
    }
}

$hydrator = new ClassMethods( false );
$hydrator->removeFilter('is');

$object = new Example();

$properties = $hydrator->extract($object);

/*
 * Outputs:
 * array(1) {
 *  'scalarProperty' =>
 *   string(11) "hello world"
 * }
 */
var_dump($properties);
```

Provider Interface

The provider interface allows an object used with a hydrator to supply the hydrator with its own filters. It is generally recommended not to use this interface, as it exposes the class to its own serialization functionality.

```
namespace Zend\Stdlib\Hydrator\Filter;

interface FilterProviderInterface

{
    /**
     * Provides a filter for hydration
     *
     * @return FilterInterface
     */
    public function getFilter();

}
```

Note: all pre-registered filters from the `ClassMethods` hydrator are ignored when this interface is used.

Strategy Interface

You can add a `Zend\Stdlib\Hydrator\Strategy\StrategyInterface` to any hydrator to manipulate how it implements `extract()` and `hydrate()` for specific key /value pairs:

```
namespace Zend\Stdlib\Hydrator\Strategy;

interface StrategyInterface
{
    /**
     * Converts given value so that it can be extracted by the hydrator.
     *
     * @param mixed $value The original value.
     * @return mixed Returns the value that should be extracted.
     */
    public function extract($value);

    /**
     * Converts the given value so that it can be hydrated by the hydrator.
     *
     * @param mixed $value The original value.
     * @return mixed Returns the value that should be hydrated.
     */
    public function hydrate($value);
}
```

This interface is similar to `Zend\Stdlib\Hydrator\HydratorInterface` – this is because the strategies provide a proxy implementation for `hydrate()` and `extract()`.

The `Zend\Stdlib\Hydrator\Strategy\StrategyEnabledInterface` provides a set of methods for adding strategies within a hydrator: `addStrategy(), getStrategy(), hasStrategy(), and removeStrategy()`.

Every hydrator shipped with the framework provides this functionality. The `AbstractHydrator` has it fully functional implemented. If you want to use this functionality in your own hydrators, you should extend the `AbstractHydrator`.

Zend Framework 2 ships with the following strategy implementations:

- `ClosureStrategy…` uses a supplied closure
- `DefaultStrategy…` simply returns the value
- `SerializableStrategy…` uses a supplied serializer

```
use Zend\Stdlib\Hydrator\ClassMethods;
use Zend\Stdlib\Hydrator\Strategy\StrategyInterface;

class Example
{

    /**
     * @var int
     */
    protected $timestamp = 1388534400;

    /**
     * Get the timestamp property value
     *
     * @return bool
     */
    public function getTimestamp()
    {
        return $this->timestamp;
    }
}
```

(continued on next page...)

```
class ConvertTimestampStrategy implements StrategyInterface
{
    public function extract($value)
    {
        $date = new DateTime('@' . $value);
        return $date->format(DateTime::RSS);
    }

    public function hydrate($value)
    {
        $date = new DateTime($value);
        return $date->getTimestamp();
    }
}

$hydrator = new ClassMethods( false );
$hydrator->addStrategy('timestamp', new ConvertTimestampStrategy());


$object = new Example();

$properties = $hydrator->extract($object);

/*
 * Outputs:
 * array(1) {
 *  'timestamp' =>
 *  string(31) "Wed, 01 Jan 2014 00:00:00 +0000"
 * }
 */

var_dump($properties);
```

Aggregate Hydrator

`Zend\Stdlib\Hydrator\Aggregate\AggregateHydrator` is an implementation of `Zend\Stdlib\Hydrator\HydratorInterface` that composes multiple hydrators via event listeners. You typically use an aggregate hydrator to hydrate or extract data from complex objects that implement multiple interfaces, and therefore need multiple hydrators to handle that in subsequent steps. Use of this hydrator requires the `Zend\EventManager`, which is used internally by triggering events for each of the `hydrate()` and `extract()` methods. A simple use case for an aggregate hydrator might also be to extract both properties and methods from an object via multiple methods:

```
use Zend\Stdlib\Hydrator\Aggregate\AggregateHydrator;
use Zend\Stdlib\Hydrator\ClassMethods;
use Zend\Stdlib\Hydrator\ObjectProperty;

class Example
{
    public $foo = 'foo';

    protected $bar = 'bar';

    public function setBar($value)
    {
        $this->bar = (string) $value;
        return $this;
    }

    public function getBar()
    {
        return $this->bar;
    }
}

$hydrator = new AggregateHydrator();
$hydrator->add($methods    = new ClassMethods(),
AggregateHydrator::DEFAULT_PRIORITY);
$hydrator->add($properties = new ObjectProperty(),
AggregateHydrator::DEFAULT_PRIORITY-10);

$example = new Example();

/*
 * Outputs:
 * array(2) {
 *  'bar' =>
 *  string(3) "bar"
 *  'foo' =>
 *  string(3) "foo"
 * }
 */
var_dump($hydrator->extract($example));
```

Zend\Session

Session Manager

`Zend\Session\SessionManager` is the class responsible for all aspects of session management: starting a session, checking if a session exists, writing the session data, regenerating the session identifier, setting the session duration, and destroying a session. Session containers consume the `SessionManager` in order to be able to namespace portions of session data, and provide individual expiry for each container. The session manager can validate sessions to ensure that the session data is correct and that the session has not been hijacked.

Initializing the Session Manager

Generally, you want to ensure that the session manager has been initialized. The configuration is set inside of the application's module bootstrap. It is also a good practice to supply validators to prevent session hijacking.

Configuration of the session manager in a standard ZF2 application can be done in either the local or global application configuration, using the top-level key "`session`". Configuration of session behavior falls under the second-level key "`config`", while configuration of session storage falls under the second-level key "`storage`". Example:

```
return array(
    'session' => array(
        'config' => array(
            'class' => 'Zend\Session\Config\SessionConfig',
            'options' => array(
                'name' => 'myapp',
            ),
        ),
        'storage' => 'Zend\Session\Storage\SessionArrayStorage',
        'validators' => array(
            'Zend\Session\Validator\RemoteAddr',
            'Zend\Session\Validator\HttpUserAgent',
        ),
    ),
);
```

For an example of how to use this configuration to create the session manager, consult the ZF2 manual.

`Zend\Session\Config\StandardConfig` provides a basic interface for implementing sessions *when not* leveraging PHP's `ext/session`. This is used more for special cases, such as when session management is handled by another system, or when unit testing.

Basic Usage Example:

```
use Zend\Session\Config\StandardConfig;
use Zend\Session\SessionManager;

$config= new StandardConfig();
$config->setOptions( array(
    'remember_me_seconds'=>1800,
    'name'=>'zf2',
));

$manager = new SessionManager($config);
```

`Zend\Session\Config\SessionConfig` provides a basic interface for implementing sessions *when* leveraging PHP's `ext/session`. It is based on the `StandardConfig` object, but also proxies to the underlying session management functions and `INI` settings of PHP.

Basic Usage Example:

```
use Zend\Session\Config\SessionConfig;
use Zend\Session\SessionManager;

$config = new SessionConfig();

$config->setOptions( array(
    'phpSaveHandler'=>'redis',
    'savePath'=>'tcp://127.0.0.1:6379?weight=1&timeout=1',
));

$manager = new SessionManager($config);
```

A full list of configuration options for `StandardConfig` and `SessionConfig` is available in the Zend Framework 2 manual.

Session Container

Zend\Session\Container instances provide the primary API for manipulating session data in ZF2. Containers are used to segregate all session data; however, you can use the default namespace if you want only one namespace for all session data. Each instance of Zend\Session\Container corresponds to an entry of Zend\Session\Storage, where the namespace is used as the key. Zend\Session\Container itself is an instance of an ArrayObject.

Basic Usage:

```
use Zend\Session\Container;

$container = new Container('namespace');
$container->item = 'foo';
```

```
You can also statically set the default session manager (ex: when using
multiple session managers):
use Zend\Session\Container;
use Zend\Session\SessionManager;

$manager= new SessionManager();

Container::setDefaultManager($manager);
```

Session Save Handlers

ZF2 comes with a standard set of Save Handler classes. These handler classes are decoupled from PHP's save handler functions and areonlyimplemented as a PHP save handler when used along with Zend\Session\SessionManager.

Zend\Session\SaveHandler\Cache allows you to provide an instance of Zend\Cache, to use as a session save handler. This is common when using products such as memcached:

```
use Zend\Cache\StorageFactory;
use Zend\Session\SaveHandler\Cache;
use Zend\Session\SessionManager;

$cache = StorageFactory::factory( array(
    'adapter' => array(
        'name' => 'memcached',
        'options' => array(
            'server' => '127.0.0.1',
        ),
    ),
));

$saveHandler = new Cache($cache);
$manager     = new SessionManager();

$manager->setSaveHandler($saveHandler);
```

Session Storage

Storage handlers are the intermediary between the start of a session and when it writes and closes. Zend\Session\Storage\ArrayStorage is the default session storage.

Zend\Session\Storage\ArrayStorage provides a facility to store all information in an ArrayObject. This storage method is likely incompatible with 3rd party libraries and all properties are inaccessible through the $_SESSION property. Additionally, ArrayStorage does not automatically repopulate the storage container with each new request and would have to manually be re-populated. The primary use case for ArrayStorage is for unit testing and/or functional testing.

Basic Usage:

```
use Zend\Session\Storage\ArrayStorage;
use Zend\Session\SessionManager;

$populateStorage = array('foo' => 'bar');
$storage = new ArrayStorage($populateStorage);
$manager = new SessionManager();

$manager->setStorage($storage);
```

Zend\Session\Storage\SessionStorage replaces $_SESSION by providing a facility to store all information in an ArrayObject. This means that it may not be compatible with 3rd party libraries, although information stored in the $_SESSION superglobal should be available in other scopes.

Basic Usage:

```
use Zend\Session\Storage\SessionStorage;
use Zend\Session\SessionManager;

$manager = new SessionManager();
$manager->setStorage( new SessionStorage());
```

`Zend\Session\Storage\SessionArrayStorage` provides a facility to store all information directly in the `$_SESSION` superglobal. This storage class provides the greatest compatibility with 3rd party libraries and allows for directly storing information into `$_SESSION`.

Basic Usage:

```
use Zend\Session\Storage\SessionArrayStorage;
use Zend\Session\SessionManager;

$manager = new SessionManager();
$manager->setStorage( new SessionArrayStorage());
```

Session Validators

Session validators are used in the defense against session hijacking. Protecting against hijacking is a complex process that needs to take into consideration many complications, such as allowing for an IP address that may change from the end user depending on their ISP, or a change in the browser's user agent during a request due to a web browser extension OR an upgrade that retains session cookies.

`Zend\Session\Validator\HttpUserAgent` provides a validator to check the session against the original, stored variable `$SERVER['HTTP_USER_AGENT']`. If the two do not match, an exception is thrown in `Zend\Session\SessionManager` after `session_start()` has been called.

Basic Usage:

```
use Zend\Session\Validator\HttpUserAgent;
use Zend\Session\SessionManager;

$manager = new SessionManager();

$manager->getValidatorChain()->attach(
    'session.validate',
    array( new HttpUserAgent(), 'isValid' )
);
```

`Zend\Session\Validator\RemoteAddr` provides a validator to check the session against the original, stored variable `$_SERVER['REMOTE_ADDR']`. If the two do not match, an exception is thrown in `Zend\Session\SessionManager` after `session_start()` has been called.

Basic Usage:

```
use Zend\Session\Validator\RemoteAddr;
use Zend\Session\SessionManager;

$manager= new SessionManager();
$manager->getValidatorChain()->attach(
    'session.validate',
    array( new RemoteAddr(), 'isValid')
);
```

Zend\Paginator

Zend\Paginator is a flexible component for paginating collections of data to present to users. The component is designed to:

- Paginate arbitrary data, not just relational databases
- Fetch only the results that need to be displayed
- Avoid forcing users to adhere to only one way of displaying data or rendering pagination controls
- Loosely couple Zend\Paginator to other ZF2 components (ex: Zend\View, Zend\Db, …) so it can be used independently

Paginating Data Collections

Zend\Paginator requires a generic way of accessing data to operate, which is provided through data source adapters. ZF2 includes some of these adapters by default:

- ArrayAdapter Paginates a PHP array
- DbSelect Takes a Zend\Db\Sql\Select instance, plus either a Zend\Db\Adapter\Adapter or Zend\Db\Sql\Sql instance, in order to paginate rows from a database
- Iterator Accepts an instance of an object that implements both Iterator and Countable
- Null Allows you to provide a count of items, and then provides a zero-filled array of items for each page
- Callback Allows you to provide a callable that is called on pagination
- DbTableGateway In a similar fashion to DbSelect, allows you to paginate rows from a database, in this instance defined by a Zend\Db\TableGateway instance

Creating an Instance of Zend\Paginator

To create an instance of `Zend\Paginator`, supply an adapter to the constructor.

```
use Zend\Paginator\Paginator;
use Zend\Paginator\Adapter\ArrayAdapter;

$paginator = new Paginator( new ArrayAdapter($array));
```

In the case of the `Null` adapter, supply an item count to its constructor in place of a data collection.

```
$paginator->setCurrentPageNumber($page);
```

The next step is to tell the paginator what page number the user requested in the controller action:

- The simplest (and recommended) way to keep track of this value is through the query string ($_GET Parameters). You can set the page number requested in your controller from a query string parameter with the simple code below.

```
$this->getRequest()->getQuery('page', 1)
```

- Some situations however require the use of a route parameter, The following is a sample route you might use in an Array configuration file:

```
return array(
    'routes' => array(
        'paginator'=> array(
            'type'=>'segment',
            'options'=> array(
                'route'=>'/list/[page/:page]',
                'defaults'=> array(
                    'page'=>1,
                ),
            ),
        ),
    ),
);
```

With the above route, set the current page number in the controller action:

```
$paginator->setCurrentPageNumber($this->params()->fromRoute('page'));
```

Rendering Pages with View Scripts

With Zend\Paginator, the view script is used to render the page items and display the pagination control. Zend\Paginator implements the SPL interface IteratorAggregate, to loop over your items and display them.

```
<html>
<body>
    <h1>Example</h1>
    <?php if (count($this->paginator)):?>
    <ul>
    <?php foreach ($this->paginator **as** $item): ?>
        <li><?php echo $item; ?></li>
    <?php endforeach ;?>
    </ul>
    <?php endif ;?>
    <?php echo $this->paginationControl($this->paginator, 'Sliding',
     'my_pagination_control', array('route' => 'application/paginator'));
?>
</body>
</html>
```

The PaginationControl view helper accepts up to four parameters:

- the paginator instance to render with
- a scrolling style
- a view script name
- and an array of additional parameters

The second and third parameters are very important. While the view script name is used to determine how the pagination control should look, the scrolling style is used to control how it should behave. By setting the default view script name, default scrolling style, and view instance, you can eliminate the calls to PaginationControl completely:

```
Zend\Paginator\Paginator::setDefaultScrollingStyle('Sliding');
Zend\View\Helper\PaginationControl::setDefaultViewPartial(
    'my_pagination_control'
);
```

When all of these values are set, you can render the pagination control inside your view script with a simple echo statement

```
<?php **echo** $this->paginator;?>
```

To see sample pagination controls, consult the ZF2 manual.

Scrolling Styles

There are four default scrolling styles packaged with ZF2:

- `All`… Returns every page; useful for dropdown menu pagination controls with relatively few pages, to make all pages available at once
- `Elastic`… A Google-like scrolling style that expands and contracts as a user scrolls through the pages
- `Jumping`… As users scroll through, the page number advances to the end of a given range, then starts again at the beginning of the new range
- `Sliding`… A Yahoo!-like scrolling style that positions the current page in the Center of the page range, or as close as possible; (this is the Default style)

Configuration

`Zend\Paginator` has several configuration methods that can be called:

- `setCurrentPageNumberSets`… the current page number (default = 1)
- `setItemCountPerPage`… Sets the maximum number of items to display on a page (default = 10)
- `setPageRange`… Sets the number of items to display in the pagination control (default = 10)

  Note: Most of the time this number will be adhered to exactly, but scrolling styles do have the option of only using it as a guideline or starting value (see: Elastic Scrolling Style)
- `setView`… Sets the view instance, for rendering

Test Your Knowledge: Questions

1

How would you compose your own writer to add to Zend\Log\Logger, using the "addWriter" method? (Choose 2)

- Write a class that implements Zend\Log\Writer\WriterInterface
- Write a class that extends the Zend\Log\Writer\AbstractWriterClass abstract class
- Write a class that has the public methods "log", "write", and "shutdown"
- Write a class that extends the Zend\Log\Writer\AbstractWriter abstract class

2

Which methods are a valid way to add “To” recipients to a Zend\Mail\Message object (assuming $message is a valid instance)? (Choose 2)

- `$message->setTo('recipient@example.com<Name>');`
- `$message->setTo('recipient@example.com', 'Name');`
- `$message->addTo(new Zend\Mail\Address('recipient@example.com', 'Name'));`
- `$message->addTo(new Zend\Mail\To(‘recipient@example.com’, ‘Name’));`

Test Your Knowledge: Answers

1

How would you compose your own writer to add to Zend\Log\Logger, using the "addWriter" method? (Choose 2)

- Write a class that implements Zend\Log\Writer\WriterInterface
- Write a class that extends the Zend\Log\Writer\AbstractWriterClass abstract class
- Write a class that has the public methods "log", "write", and "shutdown"
- Write a class that extends the Zend\Log\Writer\AbstractWriter abstract class

2

Which methods are a valid way to add “To” recipients to a Zend\Mail\Message object (assuming $message is a valid instance)? (Choose 2)

- `$message->setTo('recipient@example.com<Name>');`
- `$message->setTo('recipient@example.com', 'Name');`
- `$message->addTo(new Zend\Mail\Address('recipient@example.com', 'Name'));`
- `$message->addTo(new Zend\Mail\To(‘recipient@example.com’, ‘Name’));`

AUTHORIZATION ACCESS TOPIC OVERVIEW

- It is important that you are able to distinguish between these two functions:
  - Authentication is the process of verifying a user's identity against some set of pre-determined criteria – *"are they who they claim to be?"* ...
  - Authorization is the process of assigning rights to the user based on their identity – *"are they allowed to perform this action?"* ...

- Zend Framework 2 provides components for both functions:
  - `Zend\Authentication` *(for authentication)*
  - `Zend\Permissions\Acl` *(for authorization)*
  - `Zend\Permissions\RBAC` *(for authorization)*

- The `Zend\Authentication` component provides an API for authentication and includes concrete authentication adapters for common use case scenarios

- The `Zend\Permissions\Acl` component provides a lightweight and flexible access control list (ACL) implementation for privileges management. In general, an application utilizes ACL(s) to control access to certain protected objects by other requesting objects

- The `Zend\Permissions\Rbac` component provides lightweight, role-based access control. RBAC differs from access control lists (ACL) by putting the emphasis on roles and their permissions rather than objects (resources)

Zend\Authentication

Adapters

`Zend\Authentication` adapters are used to validate against a particular type of authentication service, such as LDAP, RDBMS, or file-based storage.

All adapters share certain features: they accept authentication credentials, perform queries against an authentication service, and return results. They can vary widely in other aspects, such as behaviors and available options.

- `Zend\Authentication` adapters can be used in **two ways**:
    - o Directly: using the adapter's `authenticate()` method
    - o Indirectly: using `Zend\Authentication\AuthenticationService::authenticate()`

- `Zend\Authentication` adapters return an instance of `Zend\Authentication\Result` with the `authenticate()` method, to represent the results of an authentication attempt

- Four methods provide a basic set of **user-facing operations**:

`isValid()`	returns TRUE upon successful authentication
`getCode()`	returns a `Zend\Authentication\Result` constant identifier to determine success or failure
`getIdentity()`	returns the identity of a successful authentication attempt
`getMessages()`	returns an array of messages regarding a failed attempt

- Common authentication technology adapters include:

LDAP	`Zend\Authentication\Adapter\Ldap`
Database table	`Zend\Authentication\Adapter\DbTable`
HTTP	`Zend\Authentication\Adapter\Http`

Storage

An important aspect of the authentication process is the ability to retain the identity, to have it persist across requests in accordance with the PHP session configuration. By default:

- `Zend\Authentication` provides **persistent storage** of the identity from a successful authentication attempt using the PHP session
- `Zend\Authentication\AuthenticationService::authenticate()` is used to **store the result** object
- `Zend\Authentication\AuthenticationService` uses a **storage class** named `Zend\Authentication\Storage\Session`
- `Zend\Authentication\Storage\Session` uses a **session namespace**, Zend_Auth
- Storage can be **customized** by creating an object that implements `Zend\Authentication\Storage\StorageInterface` and passing it via the `Zend\Authentication\AuthenticationService::setStorage()` method
- ZF2 provides **chain storage** for using multiple storage mechanisms in a queue; the chained storage interfaces can be given a priority (which defaults to the order they are added in), and once the earliest storage adapter gets a match, the rest of the storage adapters are populated; for example, you may want to check the Session adapter for credentials first, before checking the Database adapter

Zend\Permissions\Acl

By using an ACL, an application controls how request objects (**roles**) are granted access to protected objects (**resources**). Rules can be assigned according to a set of criteria – see Assigning Rules via Assertions later in this document.

Resources

A resource is an object to which access is controlled.

- `Zend\Permissions\Acl\Resource\ResourceInterface` is used to **create** a resource, implemented by a class consisting of the single method `getResourceId()`
- `Zend\Permissions\Acl\Resource\GenericResource` is provided as a basic resource **implementation** for developers to extend as needed
- `Zend\Permissions\Acl\Acl` provides a **tree structure**, to which multiple resources can be added and organized
- `Zend\Permissions\Acl\Acl` allows you to assign **privileges** (ex: read, update) on resources for specific roles

Roles

A role is an object that requests access to a resource.

- Roles **implement** `Zend\Permissions\Acl\Role\RoleInterface`, which consists of the single method `getRoleId()`
- `Zend\Permissions\Acl\Role\GenericRole` is provided as a **basic role** implementation for developers to extend as needed
- `Zend\Permissions\Acl\Acl` allows for **multiple role inheritance**, thereby supporting the inheritance of rules

General ACL Process

Create ACL
Register Roles
Define Rules

Creating the ACL

Creating an ACL utilizing `Zend\Permissions\Acl\Acl` can be done by just instantiating the class:

```
$acl = new \Zend\Permissions\Acl\Acl();
```

By default, Zend denies all access to resource privileges unless an "allow" rule is specified.

Registering Roles

Commonly, `Zend\Permissions\Acl\Role\GenericRole` is used to add roles, but any object that implements `Zend\Permissions\Acl\Role\RoleInterface` will work. Roles can be added using the `addRole()` method, passing either an instance of `Zend\Permissions\Acl\Role\GenericRole`, or a string (which will create a new `GenericRole` with the name of the passed string).

```
$acl->addRole('guest');
```

Defining Rules

`Zend\Permissions\Acl\Acl` provides an implementation in which rules are assigned, from general to specific. The most specific rule will be honored. Rules are added using the `allow()` and `deny()` methods.

ZF2's inheritance feature helps to reduce the number of rules needed.

```
// allow role 'guest' to privilege 'view'
$acl->allow('guest', **null** ,'view');
// deny roles 'guest' & 'hidden' to privileges 'edit' & 'delete'
$acl->deny(array('guest','hidden'), null, array('edit','delete'));
```

Removing Access Controls

Use the methods `removeAllow()` and `removeDeny()`, or set a `NULL` privilege.

```
$acl->removeAllow('guest', **null** ,'view');
$acl->removeDeny('hidden', **null** ,'edit');
```

Zend\Permissions\Rbac

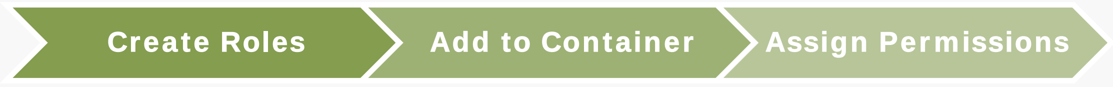
RBAC differs from access control lists (ACL) by putting the emphasis on **roles** and their **permissions** rather than objects (resources).

RBAC Model

An RBAC-based access control model is composed of the following elements and relationships:

- Role: object that requests access to a permission
- Permission: assigned to a role
- Identity: assigned one or more roles
    - Identities <-> Roles: many to many relationship
    - Roles <-> Permissions: many to many relationship
    - Roles may have a parent role

General RBAC Process



Creating A Role

- Creating a role is done by **extending** the abstract class `Zend\Permission\Rbac\AbstractRole` **or** use the **default class** provided in `Zend\Permission\Rbac\Role`
- You can instantiate a role and add it to the RBAC container or add a role directly using the RBAC container `addRole()` method

Permissions

- Permissions can be directly applied to a role or applied after retrieval from the RBAC container
- Parent roles inherit the permissions of their children

Dynamic Assertions

Dynamic assertions allow extra assertions to be added that need to be passed at runtime for access to be granted. For example, you may have several users that have *edit* permission on a blog post, but authors should only be able to edit their *own* blog posts. Dynamic assertions should implement the `Zend\Permissions\Rbac\AssertionInterface` as follows:

```
class AssertAuthorOwnsPost implements AssertionInterface
{

    protected $blogPost;
    protected $user;

    function  _construct($blogPost, $user)
    {
        $this->blogPost=$blogPost;
        $this->user=$user;
    }

    public function assert(Rbac $rbac)
    {
        if (!$this->blogPost || !$this->user) {
            return false;
        }
        return $this->blogPost->getAuthorId() === $this->user->getUserId();
    }
}
```

Passing the instantiated assertion class to the `isGranted()` method will then check against the dynamic assertion:

```
$assertion= new AssertAuthorOwnsPost($blogPost, $user);
$rbac->isGranted($user->getRole(),'edit.post', $assertion);
```

Classes

- `Zend\Permissions\Rbac\AbstractIterator`
- `Zend\Permissions\Rbac\AbstractRole`
- `Zend\Permissions\Rbac\AssertionInterface`
- `Zend\Permissions\Rbac\Rbac`
- `Zend\Permissions\Rbac\Role`

Test Your Knowledge: Questions

1

Why is using HTTP Basic authentication insecure?

- Because the password is sent in clear text
- Because it is not possible to use Basic authentication with SSL
- Because it forces you to use MD5 encryption for your passwords

2

What is the main purpose for using the `Zend\Auth` component?

- Authorization and authentication
- Only authentication
- Only authorization
- There is no **`Zend\Auth`** component within ZF2

Test Your Knowledge: Answers

1

Why is using HTTP Basic authentication insecure?

- Because the password is sent in clear text
- Because it is not possible to use Basic authentication with SSL
- Because it forces you to use MD5 encryption for your passwords

2

What is the main purpose for using the `Zend\Auth` component?

- Authorization and authentication
- Only authentication
- Only authorization
- There is no `Zend\Auth` component within ZF2

FORM TOPIC OVERVIEW

- `Zend\Form` is the glue between your domain models, controllers and views

- It is intended to be a lightweight and flexible layer of objects that allow you to represent the HTML of form elements programmatically, compose filtering and validation rules, and bind data to (and from) the form

- The important elements that make up a Zend\Form component are:

`Form`	The object representation of the root form object
`Fieldset`	Groups of form elements
`Element`	Represents a single form element (ex: textbox, submit button)
`InputFilter`	A group of filtering and validation rules to apply to the elements
`ViewHelper`	A series of helper classes tasked with rendering the HTML of forms, fieldsets and elements

Zend\Form

Elements

A Zend\Form\Element is the most low-level component of Zend\Form and is used to logically represent the base HTML elements of a form. ZF2 ships with Element classes to generate all of the expected form elements such as Text, Select, Submit. It also contains various "special" Element classes that generate more specialized components used for securing forms, such as Csrf and Captcha, which are used for securing forms (they will be covered in-depth later).

All Element objects extend the base Zend\Form\Element class, which in turn implements the interfaces: InitializableInterface, ElementInterface and ElementAttributeRemovalInterface.

```
// create the element
$title = new \Zend\Form\Element\Text();
// set its unique name
$title->setName('title');

// or


$title = new \Zend\Form\Element\Text('title');
// give it a value (the text to be displayed in the box when rendered)
$title->setValue('Mostly Harmless');
// set the label to be rendered in the <label> tag
$title->setLabel('Book Title');
// give it the HTML attribute class of 'my-input'
$title->setAttribute('class', 'my-input');
```

Fieldsets

Fieldset objects are a convenient way of grouping reusable form elements that will often represent a discrete set of domain objects. A Fieldset object extends the base Element class, and implements the FieldsetInterface.

Fieldsets can be nested, so one Fieldset can contain another, but will typically contain Element objects. Element objects can be added to the Fieldset using the add() method

```
$fieldSet = new \Zend\Form\Fieldset('book-details');
$fieldSet->add($title);
```

Form

The **Form** element acts as the top-level container for all the Element and Fieldset objects, along with the information needed to validate and render the form. Zend\Form\Form extends the Fieldset class, and implements the **FormInterface** interface.

To create a form instantiate a Zend\Form\Form and use the add() method to assign Element and Fieldset objects.

```
$form = new \Zend\Form\Form('book');
$form->add($fieldset);
```

Because the Form ultimately extends the Element class, you can use those methods to configure how the HTML for the entire form should be rendered.

```
// give this form the class of form-form
$this->setAttribute('class', 'form-form');
```

Alternative Form Definitions

Factory

Zend\Form ships with a number of methods to define form objects which can make the task of building forms simpler. The **Zend\Form\Factory** takes a configuration array and produces a Zend\Form\Form object with the elements added according to that config via the createForm() method.

```
$factory = new \Zend\Form\Factory();

$form = $factory->createForm(array(
    'elements' => array(
        array(
            'name' => 'part-code',
            'type' => 'Text',
            'options' => array(
                'label' => 'Part Code',
            ),
        ),
        array(
            'name' => 'submit',
            'type' => 'Submit',
        ),
    ),
));
```

Extending Zend\Form

Probably the most common method of building forms in real projects is by extending the base Zend\Form\Form class and implementing the elements and fieldsets in the __construct() method of your custom class.

```
namespace Product\Form;

use Zend\Form\Form;

class ProductForm extends Form
{
    public function __construct()
    {
        // call parent contructor naming the form "product"
        parent::__construct('product');

        // add the elements
        $this->add(array(
            'name' => 'part-code',
            'type' => 'Text',
            'options' => array(
                'label' => 'Part Code',
            ),
        ));

        $this->add(array(
            'name' => 'submit',
            'type' => 'Submit',
        ));

    }
}
```

Annotations

You can use comment annotations to define form elements (for example in your entity object), and then use the **Zend\Form\Annotation\AnnotationBuilder** to generate the form.

Using annotations can be helpful to keep the form building logic in the same place as the entity definition, but building the form on every request from annotations can be costly, and so the generated forms should be cached where appropriate.

Note: The Annotation builder uses the annotation parser from the Doctrine project, and so has a composer dependency on the Doctrinelibrary.

```
namespace Application\Entity;

use Zend\Form\Annotation;

/**
 * @Annotation\Name("product")
 * @Annotation\Hydrator("Zend\Stdlib\Hydrator\ClassMethods")
 */
class Product
{
  /**
     * @Annotation\Type("PartNo")
     * @Annotation\Options({"label":"Part Number"})
     */
    protected $partNo;

    public function setPartNo($partNo)
    {
        $this->partNo = $partNo;
    }

    public function getPartNo()
    {
        return $this->partNo;
    }
}
$builder = new AnnotationBuilder();
$form = $builder->createForm('Application\Entity\Product');
```

FormElementManager

The `Zend\Form\FormElementManager` is an instance of the `AbstractPluginManager` and is a `ServiceManager` that is soly tasked with managing form `Element` objects.

Using the `FormElementManager` greatly simplifies the process of adding custom form elements to forms, as they only need to be added to the `FormElementManager` in the usual way.

```
class Module implements FormElementProviderInterface
{
    public function getFormElementConfig()
    {
        return array(
            'invokables' => array(
                'PartCode' => 'Parts\Form\Element\PartCode'
            ),
        );
    }
}
```

The custom element can then be added to a form in the same way as the shipped elements. This is because when an element is referenced in the type key, you are actually referencing its `FormElementManager` key name rather than its fully qualified namespace name.

```
$form->add(array(
    'name' => 'part-code',
    'type' => 'PartCode', // type is the key name from element manager
    'options' => array(
        'label' => 'Part Code',
    ),
));
```

Filtering and Validation

In order to filter and validate input, you can attach a `Zend\InputFilter\InputFilter` using the `setInputFilter()` method.

An `InputFilter` is a group of filtering and validation rules that can be used to normalize and validate input from any source. When you attach an `InputFilter` to a form, that InputFilter will automatically be run when the form's data is checked using the `isValid()` method. Input filter elements need to be named with the exact same name as the form element in order that the data in the form element is filtered and validated by the defined rules.

Adding input filter elements is done using the `add()` method. A pre-configured `InputFilter` element (that extends the `InputFilterInterface` interface), or an array that will be used to build the `InputFilter`.

```
$inputFilter = new InputFilter();
$inputFilter->add(array(
    'name' => 'part-code',
    'required' => true, // required (cannot be null)
    'filters' => array(
        array('name' => 'StringTrim'), // trim whitespace
    ),
    'validators' => array(
        array('name' => 'Alpha'), // must be alpha only
    ),
));
```

InputFilterManager

The `FilterPluginManager` and the `'ValidatorPluginManager'` are custom `ServiceManager` implementations tasked with locating filters and validators respectively. The filter and validator names in the above examples are the names of keys in the service managers. Adding your own filters and validators is done by adding keys into the plugin manager.

Validating Fieldsets

In order to validate nested fieldsets, you need to nest the InputFilters respectively.

```
$fieldSet = new \Zend\Form\Fieldset('part-details');
$fieldSet->add(array(
    'name' => 'part-code',
    'type' => 'PartCode', // type is the key name from element manager
    'options' => array(
        'label' => 'Part Code',
    ),
));
$this->add($fieldSet);

$fieldSetInputFilter = new InputFilter('part-details');
$fieldSetInputFilter->add(array(
    'name' => 'part-code',
    'required' => true, // required (cannot be null)
    'filters' => array(
        array('name' => 'StringTrim'), // trim whitespace
    ),
    'validators' => array(
        array('name' => 'Alpha'), // must be alpha only
    ),
));

$inputFilter = new InputFilter();
$inputFilter->add($fieldSetInputFilter);

$this->setInputFilter($inputFilter);
```

Processing Form Submissions

In the controller, you can process a form submission by checking if the request is of the correct type, setting the data in the form to the submitted data with the `setData()` method, and then validating with the `isValid()` method.

```
public function indexAction()
{
    $form = new PaymentForm();

    // only process if this is a POST request
    if($this->getRequest()->isPost()) {
        $form->setData($this->getRequest()->getPost());
        if($form->isValid()) {
            $validData = $form->getData();
            // do some data processing here
        }
    }
}
```

Object Binding and Hydration

Zend\Form can work with objects rather than arrays, and can use a Zend\StdLib\Hydrator to return populated entity objects from the getData() and getObject() methods. The setHydrator() method is used to tell the form which hydrator to use when hydrating and extracting data from the object, and the bind() method is used to bind the object class to be used.

```
$form = new ProductForm();
$product = new Entity\Product();

$form->setHydrator(new ClassMethods());
/// if Product is populated binding will set the values of the form
elements
$form->bind($product);

if($this->getRequest()->isPost()) {
    $form->setData($this->getRequest()->getPost());
    if($form->isValid()) {
        // $validData is a populated instance of Entity\Product
        $validData = $form->getData();
        // do some data processing here
    }
}
```

Rendering Forms in the View

Typically, forms are rendered in the view layer by using form view helpers. There are a number of form view helpers **bundled with the framework** that can be used to quickly and easily output the form. This can be useful for prototyping.

There are a number of helpers that can be used to render individual elements, labels and errors, and also helpers that can be used to render whole rows and whole forms.

```
<div class="login-form">
    <?php echo $this->form($form); ?>
</div>
```

Generally, you will want to render forms with exact markup expected by the project. In this case, you should write your own form view helpers. Writing form view helpers is exactly the same process as writing any other view helpers (see the MVC chapter).

Test Your Knowledge: Questions

1

What happens when an object is bound to a form using the bind method?

- The values of the object are validated using the input filter specifications
- The values of the object are extracted using the attached hydrator
- The form is automatically prepared for rendering
- The values of the object are automatically modified if the form already contained validated data using the attached hydrator

2

Which statement is incorrect?

- The `FormElementManager` can be used to add custom fieldsets
- The `FormElementManager` can be used to add custom form elements
- The `FormElementManager` can be used to add custom form collections
- The `FormElementManager` can be used to add custom form hydrators !!
- The `FormElementManager` can be used to add custom forms

Test Your Knowledge: Answers

1

What happens when an object is bound to a form using the bind method?

- The values of the object are validated using the input filter specifications
- The values of the object are extracted using the attached hydrator
- The form is automatically prepared for rendering
- The values of the object are automatically modified if the form already contained validated data using the attached hydrator

2

Which statement is incorrect?

- The `FormElementManager` can be used to add custom fieldsets
- The `FormElementManager` can be used to add custom form elements
- The `FormElementManager` can be used to add custom form collections
- The `FormElementManager` can be used to add custom form hydrators
- The `FormElementManager` can be used to add custom forms

FILTERING TOPIC OVERVIEW

- `Zend\InputFilter`: a component that filters and validates generic sets of input data

- `Zend\Filter`: provides a set of commonly needed data filters, along with a simple filter chaining mechanism by which multiple filters can be applied to a single datum in a user-defined order

- `Zend\Validator`: provides a set of commonly needed validators. It also provides a simple validator chaining mechanism by which multiple validators may be applied to a single datum in a user-defined order.

Zend\Filter

Zend\Filter is a component that provides a set of commonly needed data filters. It also provides a simple filter chaining mechanism by which multiple filters may be applied to a single datum in a user-defined order.

The process of Filtering can apply to two different processes – one that removes data and produces a subset of the input, or one that transforms (sometimes referred to as "normalizes") the input. The latter definition – transformation of input - applies when using the Zend\Filter component

Zend\Filter\FilterInterface requires a single method, filter(), to be implemented by a filter class.

```
$htmlEntities= new Zend\Filter\HtmlEntities();
echo $htmlEntities->filter('&');
echo $htmlEntities->filter('"');
```

Basic Usage

```
$strtolower = new Zend\Filter\StringToLower;
echo $strtolower->filter('I LOVE ZF2!'); // i love zf2!
```

If a filter inherits from **Zend\Filter\AbstractFilter** (as with all out-of-the-box filters) you can also use them as callable PHP objects:

```
$strtolower = new Zend\Filter\StringToLower;
echo $strtolower('I LOVE ZF2!'); // i love zf2!
```

Using the StaticFilter

```
echo StaticFilter::execute('&','HtmlEntities');
```

If it is inconvenient to load a given filter class and create an instance of the filter, you can use StaticFilter with its method, execute(), as an alternative invocation style. The execute() method automatically loads the class, creates an instance, and applies the filter() method to the data input:

The first argument is a data input value to pass to the filter() method; the second argument is a string that corresponds to the basename of the filter class, relative to the Zend\Filter namespace. You can also pass an array of constructor arguments.

Multiple Filtering

```
$original = "my_original_content";

// Attach a filter
$filter = new Zend\Filter\Word\UnderscoreToCamelCase();
$filtered = $filter->filter($original);

// Use its opposite
$filter2 = new Zend\Filter\Word\CamelCaseToUnderscore();
$filtered = $filter2->filter($filtered)
```

When using two filters in sequence, keep in mind that it is often not possible to get the original output by using the opposite filter. For example:

At first glance, it may appear that the original output will be returned after the second filter is applied, That is not the case, after applying the first filter, `my_original_content` will be changed to `MyOriginalContent` , but after applying the second filter the result is `My_Original_Content`.

Standard Filter Classes

ZF2 comes with a standard set of filters, for example:

`Alnum([boolean $allowWhiteSpace [,string$locale]]):`

```
// Default settings, deny whitespace
$filter= new \Zend\I18n\Filter\Alnum();
echo $filter->filter("This is (my) content: 123");
// Returns "Thisismycontent123"

// First param in constructor is $allowWhiteSpace
$filter = new \Zend\I18n\Filter\Alnum(true);
echo $filter->filter("This is (my) content: 123");
// Returns "This is my content 123"
```

Filter Chains

Often, multiple filters should be applied to some value in a particular order; **Zend\Filter\FilterChain** provides this functionality. Filters are run either in the order in which they are added, or based on a priority value passed when registering the filter. In the following example, any non-alphabetic characters are first removed from the username and then any uppercase characters are converted to lowercase.

```
// Create a filter chain and add filters to the chain
$filterChain = new Zend\Filter\FilterChain();
$filterChain->attach(new Zend\I18n\Filter\Alpha())
    ->attach( new Zend\Filter\StringToLower());

// Filter the username
$username = $filterChain->filter($_POST['username']);
```

Any object that implements **Zend\Filter\FilterInterface** may be used in a filter chain.

Setting Filter Chain Order

For each filter added to the FilterChain, you can set a priority to define the chain order. The default value is 1000. In this example, uppercase characters are converted to lowercase before any non-alphabetic characters are removed:

```
// Create a filter chain and add filters to the chain
$filterChain = new Zend\Filter\FilterChain();
$filterChain->attach(new Zend\I18n\Filter\Alpha())
    ->attach( new Zend\Filter\StringToLower(), 500);
// 500 is lower than default 1000 priority
```

Using the Plugin Manager

An instance of the FilterPluginManager is attached to every FilterChain object. Every filter used in a FilterChain must be known by this FilterPluginManager. To add a filter, use attachByName(), where the first parameter is the filter's name within the FilterPluginManager, the second parameter takes any options for creating the filter instance; the third parameter is the priority.

Note: For more information about FilterPluginManager see the Service Manager chapter.

Zend\Filter\Inflector

Zend\Filter\Inflector is a general purpose tool for rules-based inflection of strings to a given target. It implements **Zend\Filter\FilterInterface**; you perform inflection by calling `filter()` on the object instance.

```
// Transform `MixedCase` and `camelCaseText` to another format
$inflector= new Zend\Filter\Inflector('pages/:page.:suffix');
$inflector->setRules(array(
    ':page' => array('Word\CamelCaseToDash', 'StringToLower'),
    'suffix' => 'html',
));

$string = 'camelCasedWords';
$filtered = $inflector->filter(array('page'=> $string));

// pages/camel-cased-words.html
$string = 'this_is_not_camel_cased';
$filtered = $inflector->filter(array('page' => $string));
// pages/this_is_not_camel_cased.html
```

An inflector requires a target and one or more rules. A target is basically a string that defines placeholders for variables you wish to substitute. These are specified by prefixing with a `:`.

When calling `filter()`, you then pass in an array of key and value pairs corresponding to the variables in the target. Each variable in the target can have zero or more rules associated with them. Rules may be either static or refer to a Zend\Filter class. Static rules will replace with the text provided. Otherwise, a class matching the rule provided will be used to inflect the text.

The inflector target is a string with some placeholders for variables. Placeholders take the form of an identifier, a colon (:) by default, followed by a variable name (`:script`, `:path`). The `filter()` method looks for the identifier followed by the variable name being replaced. You can change the identifier using the `setTargetReplacementIdentifier()` method, or passing it as the third argument to the constructor:

```
// Via constructor
$inflector = new Zend\Filter\Inflector('#foo/#bar.#sfx', null, '#');

// Via accessor
$inflector->setTargetReplacementIdentifier('#');
```

Writing Filters

`Zend\Filter` supplies a set of commonly needed filters, but developers will often need to write custom filters for their particular use cases. Custom filters can be any class implementing `Zend\Filter\FilterInterface`, or any PHP callable (including classes implementing `__invoke()`).

`Zend\Filter\FilterInterface` defines a single method, `filter()`, that can be implemented by user classes.

```
namespace Application\Filter;

use Zend\Filter\FilterInterface;

class MyFilter implements FilterInterface
{
    public function filter($value)
    {
        // perform some transformation upon $value to arrive on
$valueFiltered
        return $valueFiltered;
    }
}
```

Then to attach an instance of the filter defined above to a filter chain:

```
$filterChain = new Zend\Filter\FilterChain();
$filterChain->attach(new Application\Filter\MyFilter());
```

Zend\InputFilter

Zend\InputFilter is a component that filters and validates generic sets of input data. To pass input data to the InputFilter, use the setData() method. The data must be specified using an associative array. Below is an example of validating data (an email and a password) coming from a form using the POST method.

```
use Zend\InputFilter\InputFilter;
use Zend\InputFilter\Input;
use Zend\Validator;

$email = new Input('email');
$email->getValidatorChain()
    ->attach( new Validator\EmailAddress());

$password = new Input('password');
$password->getValidatorChain()
    ->attach( new Validator\StringLength(8));

$inputFilter = new InputFilter();
$inputFilter->add($email)
->add($password)
->setData($_POST);


if ($inputFilter->isValid()) {
    echo "The form is valid";
} else {
    echo "The form is not valid \n ";
    foreach ($inputFilter->getInvalidInput() as $error) {
        print_r($error->getMessages());
    }
}
```

Add Validators

You can add one or more validators to each input using the `attach()` method for each validator. It is also possible to specify a *validation group*, a subset of the data to be validated; this may be done using the `setValidationGroup()` method. You can specify the list of the input names as an array or as individual parameters.

```
// As individual parameters
$inputFilter->setValidationGroup('email', 'password');

// or as an array of names
$inputFilter->setValidationGroup(array ('email','password'));
```

Validate and Filter Data

You can validate and/or filter data using the `InputFilter`. To filter data, use the `getFilterChain()` method of individual `Input` instances, and attach filters to the returned filter chain. The `getValue()` method returns the filtered value of the `'foo'` input, while `getRawValue()` returns the original input value.

```
use Zend\InputFilter\Input;
use Zend\InputFilter\InputFilter;

$input= new Input('foo');
$input->getFilterChain()
    ->attachByName('stringtrim')
    ->attachByName('alpha');

$inputFilter= new InputFilter();
$inputFilter->add($input)
    ->setData(array(
        'foo'=>' Bar3 ',
));

echo "Before: \n ";
echo $inputFilter->getRawValue('foo') . " \n "; // the output is ' Bar3 '

echo "After: \n ";
echo $inputFilter->getValue('foo') . " \n "; // the output is 'Bar'
```

`Zend\InputFilter\Factory` allows for the initialization of `InputFilter` based on a configuration array (or Traversable object).

```
use Zend\InputFilter\Factory;

$factory= new Factory();
$inputFilter=$factory->createInputFilter(array(
    'password'=> array(
    'name'=>'password',
    'required'=> true,
    'validators'=> array(
        array(
            'name' => 'not_empty',
        ),
        array(
            'name' => 'string_length',
            'options'=> array(
                'min'=>8
            ),
        ),
    ),
));

$inputFilter->setData($_POST);
echo $inputFilter->isValid() ? "Valid form" : "Invalid form";
```

The factory may be used to create not only Input instances, but also nested `InputFilters`, allowing you to create validation and filtering rules for hierarchical data sets.

Finally, the default InputFilter implementation is backed by a Factory. This means that when calling add(), you can provide a specification that the Factory would understand, and it will create the appropriate object. You may create either Input or InputFilter objects in this way.

```
use Zend\InputFilter\InputFilter;

$filter= new InputFilter();

// Adding a single input
$filter->add(array(
    'name' => 'username',
    'required' => true,
    'validators' => array(
        array(
        'name' => 'not_empty',
    ),
    array(
        'name' => 'string_length',
        'options' => array(
            'min' => 5
        ),
    ),
));

// Adding another input filter that also contains a single input. Merging
both.
$filter->add(array(
    'type' => 'Zend\InputFilter\InputFilter',
    'password' => array(
        'name' =>'password',
        'required' => true,
        'validators' => array(
            array(
                'name'=> 'not_empty',
            ),
            array(
                'name' => 'string_length',
                'options' => array(
                    'min' => 8
                ),
            ),
        ),
    ),
));
```

File Upload Input

The **Zend\InputFilter\FileInput** class is a special Input type for handling file uploads (typically using the $_FILES superglobal). While FileInput uses the same interface as Input, it differs in a few ways:

- It expects the raw value to be an entry as found in the $_FILES superglobal.
- The validators are run *before* the filters (which is the opposite behaviour of Input). This is so that analyses *uploaded* file validations can be run prior to any filters that may rename/move/modify the file.
- Instead of adding a NotEmpty validator, it will (by default) automatically add a Zend\Validator\File\UploadFile validator

Note: if you are using anelement in your form, you will need to use the FileInput *instead of* Input or you will encounter issues.

Basic Usage: Use of FileInput is essentially the same as Input

```
use Zend\Http\PhpEnvironment\Request;
use Zend\Filter;
use Zend\InputFilter\InputFilter;
use Zend\InputFilter\Input;
use Zend\InputFilter\FileInput;
use Zend\Validator;

// Description text input
$description = new Input('description'); // Standard Input type
$description->getFilterChain() // Filters run 1st w/ Input
            ->attach(new Filter\StringTrim());

$description->getValidatorChain() // Validators run 2nd w/ Input
             ->attach( new Validator\StringLength(array(
                       'max' => 140
)));

// File upload input
$file = new FileInput('file'); // Special File Input type
$file->getValidatorChain() // Validators run 1st w/ FileInput
     ->attach(new Validator\File\UploadFile());

$file->getFilterChain() // Filters run 2nd w/ FileInput
     ->attach(new Filter\File\RenameUpload(array(
        'target' => './data/tmpuploads/file',
        'randomize' => true ,
)));
```

```
// Merge $\_POST and $\_FILES data together
$request = new Request();
$postData = array_merge_recursive($request->getPost(),
$request->getFiles());

$inputFilter = new InputFilter();
$inputFilter->add($description)
    ->add($file)
    ->setData($postData);

if ($inputFilter->isValid()) {
    // FileInput validators are run, but not the filters
    echo "The form is valid \n ";
    $data = $inputFilter->getValues();
    // This is when the FileInput filters are run.
} else {
    echo "The form is not valid \n ";
    foreach ($inputFilter->getInvalidInput() as $error) {
        print_r($error->getMessages());
    }
}
```

Zend\Validator

Zend\Validator provides a set of commonly needed validators. It also provides a simple validator chaining mechanism by which multiple validators may be applied to a single datum in a user-defined order.

Zend\Validator\ValidatorInterface defines two methods:

- isValid() performs validation on the provided value, returning TRUE if the value passes against the validation criteria.
- getMessages() when isValid() returns FALSE, returns an array of messages explaining reasons for validation failure.

Basic Usage:

```
$validator = new Zend\Validator\EmailAddress();

if ($validator->isValid($email)){
    // email appears to be valid
} else {
    // email is invalid; print the reasons
    foreach ($validator->getMessages() as $messageId => $message) {
        printf("Validation failure '%s': %s\n"), $messageId, $message);
    }
}
```

For information on Customizing Messages or Translating Messages, consult the ZF2 manual.

Standard Validation Classes

ZF2 comes with a standard set of validation classes. Validation classes return a Boolean value for whether or not a value validates successfully. They also provide information about *why* a value failed validation. The availability of the reasons for validation failures may be valuable to an application for various purposes, such as providing statistics for usability analysis, or for providing failure messages in forms.

For example, `Zend\I18n\Validator\Alnum` allows you to validate whether a given value contains only alphanumerical characters.

```
$validator = new Zend\I18n\Validator\Alnum();

if ($validator->isValid('Abcd12')) {
    // value contains only allowed chars
} else {
    // false
}
```

For a complete list of the Standard Validation and File Validation Classes packaged with ZF2, consult the manual.

Validator Chains

Often, multiple validations should be applied to some value in a particular order. Validators are run in the order they were added to `Zend\Validator\ValidatorChain`. Any object that implements `Zend\Validator\ValidatorInterface` may be used in a validator chain.

Example: Username must be between 6 and 12 alphanumeric characters

```
// Create a validator chain and add validators to it
$validatorChain = new Zend\Validator\ValidatorChain();
$validatorChain->attach(new Zend\Validator\StringLength(
    array(
        'min' => 6,
        'max' =>12,
        )
    )
)->attach( new Zend\I18n\Validator\Alnum());

// Validate the username
if ($validatorChain->isValid($username)) {
    // username passed validation
} else {
    // username failed validation; print reasons_
    foreach ($validatorChain->getMessages() as $message) {
        echo "_$message_ \n ";
    }
}
```

In some cases it makes sense to have a validator break the chain if its validation process fails. `Zend\Validator\ValidatorChain` supports these cases via the `attach()` method's second parameter.

By setting `$breakChainOnFailure` to `TRUE`, the added validator will break the chain execution upon failure, which avoids running any other validations that are unnecessary or inappropriate for the situation.

If the above example were written as follows, then the alphanumeric validation would not occur if the string length validation fails:

```
$validatorChain->attach(new Zend\Validator\StringLength(
    array ('min' => 6,
    'max' => 12
)), true)
->attach(new Zend\I18n\Validator\Alnum());
```

Writing Validators

As mentioned earlier, `Zend\Validator\ValidatorInterface` defines two methods, `isValid()` and `getMessages()`. These methods can be implemented by user classes to create custom validation objects. An object that implements `Zend\Validator\ValidatorInterface` may be added to a validator chain with `Zend\Validator\ValidatorChain::attach()`.

Basic validation failure message functionality is implemented in `Zend\Validator\AbstractValidator`. To include this functionality when creating a validation class, extend `Zend\Validator\AbstractValidator`. In the extending class, implement the `isValid()` method logic and define the message variables and message templates that correspond to the types of validation failures that can occur. If a value fails your validation tests, then `isValid()`should return `FALSE`. If the value passes your validation tests, the `isValid()` should return `TRUE`.

Basic Usage: Creating a Simple Validation Class

```
class MyValid\Float extends Zend\Validator\AbstractValidator
{
    const FLOAT ='float';
    protected $messageTemplates = array (
        self::FLOAT => "'%value%' is not a floating point value"
    );

    public function isValid($value)
    {
        $this->setValue($value);
        if (!is_float($value)) {
            $this->error(self::FLOAT);
            return false;
        }

        return true;
    }
}
```

In the example on the previous page, the validation rules are that the input value must be a floating point value. The class defines a template for its single validation failure message, which includes the built-in magic parameter, `%value%`.

The call to `setValue()` prepares the object to insert the tested value into the failure message automatically, if validation fails. The call to `error()` tracks the reason for validation failure. Since this class only defines one failure message, it is not necessary to provide `error()` with the name of the failure message template.

To see how to write a validation class with Dependent / Independent conditions, consult the ZF2 manual.

Validation Messages

Each validator based on Zend\Validator\ValidatorInterface provides one or multiple messages in the case of a failed validation. You can use this information to set your own messages, or to convert existing messages to something different. These validation messages are keyed by constants which can be found at top of each validator class. Example: Zend\Validator\GreaterThan

```
protected $messageTemplates= array (
    self::NOT\GREATER => "'%value%' is not greater than '%min%'",
);
```

Here the constant self::NOT_GREATER refers to the failure and is used as the key, and the value itself is used as value of the message array. You can retrieve all message templates from a validator by using the getMessageTemplates() method. It returns you the above array, which contains all messages a validator could return in the case of a failed validation:

```
$validator = new Zend\Validator\GreaterThan();
$messages = $validator->getMessageTemplates();
$validator = new Zend\Validator\GreaterThan();

$validator->setMessage(
    'Please enter a lower value',
    Zend\Validator\GreaterThan::NOT_GREATER
);
```

Using the setMessage() method, you can set another message to be returned in case of the specified failure:

Using Pre-Translated Validation Messages

ZF2 ships with more than 45 different validators and more than 200 failure messages, including pre-translated validation messages. You can find them within the path/resources/languages in your ZF2 installation. To translate messages to a particular language, attach a translator instance to Zend\Validator\AbstractValidator using these resource files.

Example: Translating Validation Messages to German

```
$translator= new Zend\Mvc\I18n\Translator();
$translator->addTranslationFile(
    'phpArray',
    'resources/languages/en.php',
    'default',
    'en\_US'
);
Zend\Validator\AbstractValidator::setDefaultTranslator($translator);
```

Limiting the Size of a Validation Message

`Zend\Validator\AbstractValidator` can automatically limit the maximum returned size of a validation message. To get the actual set size, use `Zend\Validator\AbstractValidator::getMessageLength()`. If it is -1, then the returned message will not be truncated (default).

To limit the returned message size, use `Zend\Validator\AbstractValidator::setMessageLength()` to set the desired integer size. When the returned message exceeds the set size, the message will be truncated and the string '...' will be added instead of the rest of the message.

Ex: `Zend\Validator\AbstractValidator::setMessageLength(100);`

Test Your Knowledge: Questions

1

What must you do when implementing your own validation class? (Choose 1)

- **Register the service manager**
- **Implement the interface Zend/Validator/ValidatorInterface**
- **Have a getMessages() method**
- **Implement the error() method**

2

What is the expected result of the following code?

```
<php>
use Zend\Filter\FilterChain;
use Zend\Filter\StringToUpper;
use Zend\Filter\Word;
$string = 'php_is_great';
$filterChain = new FilterChain();
$filterChain->attach(new StringToUpper())
            ->attach(new Word\UnderscoreToCamelCase())
            ->attach(new Word\CamelCaseToDash());
echo $filterChain->filter($string);
</php>
```

- **The string ‘PHPISGREAT’ is emitted**
- **The string ‘P-H-P-I-S-G-R-E-A-T’ is emitted**
- **The string ‘php-is-great’ is emitted**

Test Your Knowledge: Answers

1

What must you do when implementing your own validation class? (Choose 1)

- Register the service manager
- **Implement the interface Zend/Validator/ValidatorInterface**
- Have a getMessages() method
- Implement the error() method

2

What is the expected result of the following code?

```
<php>
use Zend\Filter\FilterChain;
use Zend\Filter\StringToUpper;
use Zend\Filter\Word;
$string = 'php_is_great';
$filterChain = new FilterChain();
$filterChain->attach(new StringToUpper())
            ->attach(new Word\UnderscoreToCamelCase())
            ->attach(new Word\CamelCaseToDash());
echo $filterChain->filter($string);
</php>
```

- **The string ‘PHPISGREAT’ is emitted**
- The string ‘P-H-P-I-S-G-R-E-A-T’ is emitted
- The string ‘php-is-great’ is emitted

INTERNATIONALIZATION TOPIC OVERVIEW

- **Zend\I18n** is the Zend Framework 2 component responsible for internationalization - handling translations, currencies, date formats and other things that are likely to change depending on location

- Zend\I18n has a hard dependency on the **Intl (http://php.net/intl)** core PHP extension

- Note: i18n is a widely used abbreviation for internationalization, and the two words are interchangeable

Zend\I18n

Translation

Zend\I18n\Translator is a component tasked with handling the display of the correct language string depending on the locale the user is using. Typically, the translator is initialized and translations are loaded for any supported locale.

Translations can either be loaded from a file (by any class that implements the **Zend\I18n\Translator\Loader\FileLoaderInterface** interface), or from a remote source (by any class that implements the **RemoteLoaderInterface** interface).

Adding Translations

There are two options when adding translations:

- add every translation file individually (best for formats storing multiple locales in the same file)
- add translations via a pattern (best for formats containing one locale/file)

Adding a Single File to the Translator:

```
use Zend\I18n\Translator\Translator;
$translator= new Translator();
$translator->addTranslationFile($type, $filename, $textDomain, $locale);
```

Where:

- $type is the name of the format loader
- $filename points to the file containing the translations
- $textDomain specifies a category name for the translations
- $locale specifies which language the translated strings are from (required only for formats containing translations for a single locale)

Storing a Single Locale per File

When storing one locale per file, specify those files via a pattern. This allows you to add new translations to the file system without touching your code. Patterns are added with the `addTranslationFilePattern()` method. Note that you do not need to specify a locale, and that the file location is presented as a `sprintf` pattern.

```
// locale/de/messages.php
//
// return array(
//     'Welcome' => 'Willkommen'
// );

use Zend\I18n\Translator\Translator;

$translator = new Translator();

$type       = 'phparray';
$pattern    = 'i18n/%s/messages.php';
$textDomain = 'mystrings';

$translator->setLocale('de');
$translator->addTranslationFilePattern($type, __DIR__, $pattern,
$textDomain);

echo $translator->translate('Welcome', 'mystrings'); // Willkommen
```

Setting a Locale

The translator will, by default, get the locale to use from the PHP `intl` extension's `Locale` class. If you want to explicitly set an alternative locale, pass it to the `setLocale()` method.

When there is no translation for a specific message ID in a locale, the message ID itself is returned. If you wish to avoid this, you may alternatively set a fallback locale by passing the `setFallbackLocale()` method, which is then used to retrieve a fallback translation.

Translating Messages

Simply call the `translate()` method of the translator to translate a message.

```
$translator->translate($message, $textDomain, $locale);
```

Where

- `$message` is the ID of message to translate (if missing or empty, original message ID returned)
- `$textDomain` is parameter is that specified when adding translations
- `$locale` is usually not used in this context, as the locale is taken from the locale set in the translator by default.

Translating Plural Messages

Call the `translatePlural()` method to translate plural messages. Instead of a single message, it takes a singular and a plural value and an additional integer number on which the returned plural form is based.

```
translator->translatePlural(singular, plural, number, textDomain, locale);
```

Caching Translations

Caching translations in production guarantees an optimized loading procedure, and saves time and resources spent on loading and parsing individual formats.

To enable caching, pass a `Zend\Cache\Storage\Adapter` to the `setCache()*` method of the `Translator` instance. To disable the cache, pass a null value to it.

I18n View Helpers

ZF2 comes with a comprehensive set of internationalization view helper classes, for tasks such as formatting a date or currency, or displaying translated content.

Example: CurrencyFormat Helper

```
currencyFormat(
    float $number [,
    string $currencyCode = null [,
    bool $showDecimals = null [,
    string $locale = null [,
    string $pattern = null
]]]])


// Within a view script

echo $this->currencyFormat(1234.56,'USD', **null** ,'en_US'); // $1,234.56

echo $this->currencyFormat(1234.56,'EUR', **null** ,'de_DE'); // 1.234,56
€

echo $this->currencyFormat(1234.56, **null** , true ); // $1,234.56

echo $this->currencyFormat(1234.56, **null** , false ); // $1,235

echo $helper(12345678.90,'EUR', **true** ,'de_DE','#0.# kg'); //
12345678,90 kg

echo $helper(12345678.90,'EUR', **false** ,'de_DE','#0.# kg'); // This
returns: "12345679 kg"
```

Where:

- $number is the numeric currency value
- $currencyCode is the 3-letter ISO 4217 code
- `$showDecimals` (Optional) Boolean false as third argument to show no decimals
- `$locale` (Optional) Locale in which currency should be formatted
- $pattern (Optional) pattern string used by formatter

For a full guide to the `i18n` view helpers in Zend Framework 2, please refer to the manual.

I18n Filters

Tasks that are trivial to handle on a local level often become increasingly complex to code when targeted for international audiences. Filters within the Internationalization component of Zend Framework 2 help you to handle these challenges - for example, dealing with something as simple as a character range of "a-z". Should this range be the twenty six classic characters from the English alphabet? Or should it include the five diacritics and two orthographic ligatures of the French alphabet?

Example: The Alnum Filter

The `Alnum` filter can be used to return only alphabetic characters and digits in the unicode "letter" and "number" categories, respectively. All other characters are suppressed.

```
Alnum([Boolean allowWhiteSpaces[,string locale ]])
// Default settings, deny whitespace_

$filter = new  \Zend\I18n\Filter\Alnum();

echo $filter->filter("This is (my) contént: 123");
// Thisismycontént123

// First param in constructor is $allowWhiteSpace
$filter = new \Zend\I18n\Filter\Alnum(true);

echo $filter->filter("This is (my) contént: 123");
// This is my contént 123


// Second param in constructor is locale, (ja, ko, and zh will use a-z
// and not all Unicode letter category letters.)

$filter = new \Zend\I18n\Filter\Alnum(true , 'ja');

echo $filter->filter("This is (my) contént: 123");
// This is my content 123
```

Where:

- `$allowWhiteSpace` permits whitespace when set to true
- `$locale` string used to identify characters to filter (ex: locale name, en_US)

For a full guide to the i18n filters in Zend Framework 2, please refer to the manual.

I18n Validators

Validation often requires specific considerations based on user locale; for example, "1.001" can mean "one point zero zero one" in some regions, while it would mean "one thousand and one" in others. Zend Framework 2 ships with many built-in validators to solve these common problems.

Example: Float Validation

`Zend\I18n\Validator\Float` validates whether a given value contains a floating-point value; it also validates localized input.

Example: Simple Float Validation

```
$validator = new Zend\I18n\Validator\Float();
$validator->setLocale('en');

var_dump($validator->isValid(1234.5)); // bool(true)

var_dump($validator->isValid('10a01')); // bool(false)

var_dump($validator->isValid('1,234.5')); // bool(true)
```

For a full guide to the i18n validators in Zend Framework 2, please refer to the manual.

Test Your Knowledge: Questions

1

What is the correct way to set the current locale to American English (en-US)?

- `Locale::setDefault('en-US')`
- `locale('en-US')`
- `Zend\I18n\Locale::setDefault('en-US')`
- `Zend\I18n\Locale\Locale::setDefault('en-US')`

2

Which of the following CANNOT be used to add translation text strings to a Zend\I18n\Translator instance?

- Defining local translation files
- Defining local translation file patterns
- Defining remote translation files
- Adding an array with translation texts directly

Test Your Knowledge: Answers

1

What is the correct way to set the current locale to American English (en-US)?

- `Locale::setDefault('en-US')`
- `locale('en-US')`
- `Zend\I18n\Locale::setDefault('en-US')`
- `Zend\I18n\Locale\Locale::setDefault('en-US')`

2

Which of the following CANNOT be used to add translation text strings to a `Zend\I18n\Translator` instance?

- Defining local translation files
- Defining local translation file patterns
- Defining remote translation files
- Adding an array with translation texts directly

WEB SERVICES TOPIC OVERVIEW

- Web Service Clients: various services exist under the `ZendService` namespace to make connecting to specific APIs easier; for example, `ZendService\Amazon` is designed to help with authenticating and retrieving data from Amazon web service

- Zend Framework 2 ships with a number of components for building a RESTful web service

- The `Zend\Soap` component is designed to help with the creation of SOAP servers; it can be used with or without WSDL

Web Services

HttpClient

Zend\Http\Client is a component used to perform an HTTP request to external services. Requests are created, then sent using the dispatch() method, which returns an instance of **Zend\Http\Response**. The URI to request can either be passed in as the first constructor method, or by using the setUri() method.

```
$client = new Client('http://api.mywebshop.com/products');
$result = $client->dispatch();

// or


$client = new Client();
$client->setUri('http://api.mywebshop.com/products');
$result = $client->dispatch();
```

Configuration

Configuration options can either be passed as an array as the second parameter of the constructor, or by using the setOptions() method.

```
$client = new Client('http://api.mywebshop.com/products', array(
    'maxredirects' => 1,
    'timeout' => 10,
    'useragent' => 'My Cool Application',
));

// or

$client = new Client();
$client->setUri('http://api.mywebshop.com/products');
$client->setOptions(array(
    'maxredirects' => 1,
    'timeout' => 10,
    'useragent' => 'My Cool Application',
));
```

HTTP Methods

By default, a Client will perform an HTTP GET request to the selected URI. The parameters for a GET request, to be passed in as a query string, can be set using the `setParameterGet()` method, passing a key/value array of parameter pairs.

```
$client = new Client('http://api.mywebshop.com/products');
$client->setParameterGet(array(
    'filter' => 'red'
));
// will create uri http://api.mywebshop.com/products?filter=red
```

You can perform an HTTP POST method (or PUT, DELETE, ...), by passing the correct value to the `setMethod()` property. The **`Zend\Http\Request`** class has constants that correspond to valid methods. To add POST parameters, use the `setParametersPost()` method.

```
$client = new Client('http://api.mywebshop.com/products');
$client->setMethod(Request::METHOD_POST);
$client->setParameterPost(array(
    'filter' => 'red'
));
```

A convenience static class **`Zend\Http\ClientStatic`** exists to make performing GET and POST requests slightly quicker to write.

REST Services

Typically, RESTful services will output only in the format that was requested - either through an HTTP header or in a query string parameter - and will route to different actions depending on the HTTP method.

AbstractRestfulController

Because RESTful web services should only have actions that correspond to the relevant HTTP verbs, the `AbstractRestfulController` simply maps the HTTP method to the action.

- GET request without an id parameter calls the `getList()` action, and should return a relevant list of resources
- GET request with an id parameter calls the `get()` action with the parameter `$id`, and should return a single resource with the given id
- PUT request containing POST(ed) id and data parameters calls the `update()` function, passing `$id` and `$data;` this is for editing the given resource
- POST request containing only the POST(ed) data parameter calls the `create()` function, passing the `$data` parameter; this is for creating a new resource
- DELETE request should contain an id parameter, and calls the `delete()` function, passing this `$id`; you use this to delete the given resource

Note: Routes that reference `AbstractRestfulController` should contain controller and id parameters, where appropriate.

View JsonStrategy

The `JsonStrategy` is a view strategy that is tasked with converting array based output into a JSON string automatically. The strategy can be enabled in the view settings for a module (if one module enables it, it's available to all).

```
'view_manager' => array(
  'strategies' => array(
        'ViewJsonStrategy',
    ),
),
```

Once enabled, any module that returns a `JsonModel` (typically instead of a `ViewModel`) will have its parameters automatically converted to valid JSON. The `JsonStrategy` will set the correct HTTP headers for returning JSON.

AcceptableViewModelSelector Controller Plugin

The `AcceptableViewModelSelector` is a controller plugin that helps to select which view model should be returned, based on the HTTP Accept header. A key/value pair of View Model to allowable Accept values is defined, and the correct view model is instantiated and returned.

```
public function getList()
{
    $viewModel = $this->acceptableViewModelSelector(array(
        'Zend\View\Model\JsonModel' => array(
            'application/json',
        ),
        'Zend\View\Model\FeedModel' => array(
            'application/rss+xml',
        ),
    ));
}
```

SOAP Services

The **Zend\Soap** component is designed to support the creation of SOAP servers. It can be used with or without WSDL.

The **Zend\Soap\Server** component can be instantiated with different constructor parameters, depending on whether it is using a WSDL or non-WSDL mode. In WSDL mode, the first argument should be the URI of the WSDL file; for non-WSDL mode, it would simply be null.

The handling class is bound to the SOAP server using the `setClass()` method, or using the `setObject()` method if an object has already been initialized.

```
$server = new Server(null, array());
$server->setClass('My\Soap\Server\Product');
// or
$server->setObject(new My\Soap\Server\Product());
```

You can then use the `addFunction()` (or `loadFunctions()`) method to add functions to the SOAP server.

```
$server->addFunction('listProducts');
$server->addFunction('showProduct');
```

Finally, use the handle() method to tell the server to start handling requests.

```
$server->handle();
```

Test Your Knowledge: Questions

1

Which of the following calls will correctly set the headers of the `Zend\Http\Client` object stored in `$client`? (Choose 2)

- `$client->getHeaders()->addHeader('key', 'value');`
- `$client->setHeaders(array('key' => 'value));`
- `$client->addHeader('key', value');`
- `$client->setHeaders(Zend\Http\Headers::fromString('key: value');`

2

Which of the following methods are NOT defined in the Zend\Mvc\Controller\AbstractRestfulController? (Choose 2)

- `create()`
- `createList()`
- `delete()`
- `deleteList()`
- `head()`
- `getList()`
- `updateList()`

Test Your Knowledge: Answers

1

Which of the following calls will correctly set the headers of the `Zend\Http\Client` object stored in `$client`? (Choose 2)

- `$client->getHeaders()->addHeader('key', 'value');`
- `$client->setHeaders(array('key' => 'value));`
- `$client->addHeader('key', value');`
- `$client->setHeaders(Zend\Http\Headers::fromString('key: value');`

2

Which of the following methods are NOT defined in the Zend\Mvc\Controller\AbstractRestfulController? (Choose 2)

- `create()`
- `createList()`
- `delete()`
- `deleteList()`
- `head()`
- `getList()`
- `updateList()`