ZEND FRAMEWORK CERTIFICATION

STUDY GUIDE

The PHP Company

Zend Technologies, Inc.
2006-2008

ZEND FRAMEWORK CERTIFICATION :

The Zend Framework 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 (ZF), 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 ZF 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 14 areas of expertise created by the ZF Certification Education Advisory Board are:

Authentication and Authorization	Internationalization
Coding Standards	Mail
Databases	Model-View-Controller (MVC)
Diagnosis and Maintenance	Performance
Filtering and Validation	Search
Forms	Security
Infrastructure	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. 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 14 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 14 topic areas tested by the ZF Certification exam will have three associated sections within the Study Guide.

First is the Topic "Snapshot" view, which provides a visual framework of the sub-topics considered required knowledge for each major topic, and highlighted knowledge areas within each. They serve as guides, or hints… do you understand how these elements relate to the topic? Could you answer basic to advanced questions on these?

Next follows a more in-depth discussion of these various target areas ("Focus" section). In order to avoid repetition, and make the discussion more logical, the content is presented in an integrated fashion. So, the content for areas that are listed multiple times under the sub-topics (Ex: Front Controllers) will appear only once.

Finally, at the end of each topic will be a couple of representative questions, similar to those you will find on the examination. You might choose to take these questions after looking at the Snapshot, if you feel confident that you know the exam topic well, or you can take these questions as a wrap-up to your studying the Focus area 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 Certification course.

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

CERTIFICATION TOPIC : AUTHENTICATION & AUTHORIZATION

Zend_Auth

Definitions ●··● *Use / Purpose* ●··● *Identity*

Identity Persistence ●··● *Results Object*

Zend_Acl

Definitions ●··● *Use / Purpose* ●··● *Resource / Role*

Rules

Zend Framework: Authentication & Authorization

For the exam, recall that…

First, it is important that you are able to distinguish between the 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 – *"they are entitled to do the following actions because of who they are"* …

Zend Framework provides components for both functions – `Zend_Auth` and `Zend_Acl`.

ZEND_AUTH

`Zend_Auth` provides an API for authentication and includes adapters for the most commonly used scenarios.

Here are some things to keep in mind:

- `Zend_Auth` implements the Singleton pattern through its static `getInstance()` method.
    - Singleton pattern means only one instance of the class is available at any one time
    - The new operator and the clone keyword will not work with this class… use `Zend_Auth::getInstance()` instead
- The adapters authenticate against a particular service, like LDAP, RDBMS, etc.; while their behavior and options will vary, they share some common actions:
    - accept authenticating credentials
    - perform queries against the service
    - return results

Identity Persistence
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. This is accomplished in ZF with the method `Zend_Auth::authenticate()`. The default storage class is `Zend_Auth_Storage_Session` (which uses `Zend_Session`). A custom class can be used instead by creating an object that implements `Zend_Auth_Storage_Interface` and passing it to `Zend_Auth::setStorage()`.

Persistence can be customized by using an adapter class directly, foregoing the use of `Zend_Auth` entirely.

ZEND_AUTH

Authentication Results

Zend_Auth adapters return an instance of Zend_Auth_Result with authenticate() to represent the results of an authentication attempt. The Zend_Auth_Result object exposes the identity that was used to attempt authentication.

Adapters populate and return a Zend_Auth_Result object upon an authentication attempt, so that the following four methods can provide a set of user-facing operations common to the results of Zend_Auth adapters:

- `isValid()` returns TRUE if and only if the result represents a successful authentication attempt
- `getCode()` returns a Zend_Auth_Result constant identifier for confirming success or determining the type of authentication failure
- `getIdentity()` returns the identity of the authentication attempt
- `getMessages()` returns an array of messages regarding a failed authentication attempt

Zend_Auth adapters allow for the use of authentication technologies, such as

LDAP …	Zend_Auth_Adapter_Ldap
Database table …	Zend_Auth_Adapter_DbTable
HTTP …	Zend_Auth_Adapter_Http
OpenID …	Zend_Auth_Adapter_OpenId

ZEND_ACL

Zend_Acl provides a lightweight and flexible Access Control List (ACL) feature set along with privileges management, generally via the use of request objects and protected objects. Zend_Acl can be easily integrated with ZF MVC components through use of an Action Helper or Front Controller Plugin. Keep in mind that this component is *only* involved with authorizing access, and does not in any way verify identity (that process is authentication, in ZF accomplished with Zend_Auth).

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. Combining the processes of authentication and authorization is commonly called "access control".

Access control rules are specified with Zend_Acl::allow() and Zend_Acl::deny(). Note that calling Zend_Acl::isAllowed() against a Role or Resource that was not previously added to the ACL results in an exception.

Some definitions you should know:

- **Resource:** an object with controlled access
- **Role:** an object that requests access to a Resource

Creating An ACL

Creating an ACL utilizing Zend_Acl is easy, as shown in the sample code below:

```
<?php
require_once 'Zend/Acl.php';

$acl = new Zend_Acl();
```

Creating Resources

Creating a resource is a simple process - a class needs only implement the Zend_Acl_Resource_Interface, consisting of the single method getResourceId(), in order for Zend_Acl to consider the object a Resource. Zend_Acl_Resource is provided as a basic implementation for extensibility.

ZEND_ACL

Creating Roles

In a process similar to Resource creation, a class only has to implement the `Zend_Acl_Role_Interface`, consisting of the single method `getRoleId()`, for `Zend_Acl` to consider the object a Role. `Zend_Acl_Role` is provided as a basic implementation for extensibility.

Inheritance - Resources

`Zend_Acl` provides a tree structure onto which multiple resources can be added. Queries on a specific resource will automatically search the Resource's hierarchy for rules assigned to its parent Resources, allowing for simple inheritance of rules.

Accordingly, if a default rule is to be applied to all resources within a branch, it is easiest to assign the rule to the parent. Assigning exceptions to those resources *not* to be included in the parent rule can be easily imposed via `Zend_Acl`.

Note that a Resource can inherit from only one parent Resource, although that parent can trace back to its own parent, and so on.

Inheritance - Roles

Unlike Resources, in `Zend_Acl` a Role can inherit from one *or more* Roles to support the inheritance of rules. While this ability can be quite useful at times, it also adds complexity to inheritance. In the case of multiple inheritance, if a conflict arises among the rules, the order in which the Roles appear determine the final inheritance - the first rule found via query is imposed.

Assigning Rules via Assertions

There are times when a rule should not be absolute – where access to a Resource by a Role should depend on a number of criteria. `Zend_Acl` has built-in support for implementing rules based upon conditions that need to be met, with the use of `Zend_Acl_Assert_Interface` and the method `assert()`. Once the assertion class is created, an instance of the assertion class must be supplied when assigning conditional rules. A rule thus created will be imposed only when the assertion method returns TRUE.

TEST YOUR KNOWLEDGE : QUESTIONS

1 Zend_Acl supports _____ inheritance among Resource objects.

a. cyclic

b. multiple

c. no

d. single

2 Zend_Auth throws an exception upon an unsuccessful authentication attempt due to invalid credentials (e.g., the username does not exist).

a. True

b. False

TEST YOUR KNOWLEDGE : ANSWERS

★ = CORRECT

1

```
Zend_Acl supports _____ inheritance among Resource objects.

     a. cyclic

     b. multiple

     c. no

  ★  d. single
```

2

```
Zend_Auth throws an exception upon an unsuccessful
authentication attempt due to invalid credentials (e.g., the
username does not exist).

     a. True
  ★  b. False
```

CERTIFICATION TOPIC : CODING CONVENTIONS

Coding Conventions

Use / Purpose *Naming Conventions* *Closing Tags*

Zend Framework: Coding Standards

For the exam, here's what you should know already …

You should know general coding standards for using PHP (Ex: tags, code demarcation, syntax for strings, & arrays, …) as well as accepted naming conventions.

You should know the rules, guidelines, and code standards established for the Zend Framework.

ZF PHP CODING STANDARDS

Good coding standards are important in any development project, particularly when multiple developers are working on the same project. Having coding standards helps to ensure that the code is of high quality, has fewer bugs, and is easily maintained.

PHP File Formatting

- Never use the closing tag "?>" for files that contain only PHP code – this prevents trailing whitespace from being included in the output
- Indentation should be 4 spaces, not using tabs
- Maximum line length is 120 characters, with the goal limit of 80 characters for clarity
- Lines should be terminated with a linefeed (LF, ordinal 10, hexadecimal 0x0A), not a carriage return or a carriage return/linefeed combination

Naming Conventions

Class Names

- Map directly to the directories where they are stored
- Contain only alphanumeric characters; numbers are discouraged; underscores are permitted only in place of the path separator (Example: `Zend/Db/Table.php` maps to `Zend_Db_Table`)
- Multiple-word names: in general, each word contains a capitalized first letter, with the remaining letters lowercase (Ex: `Zend_Pdf`); there are exceptions, as when a word is an acronym or somehow non-standard (Ex: `Zend_XmlRpc`)
- Classes authored by ZF or its Partners must start with "`Zend_`" and must be stored under the `Zend/` directory hierarchy; conversely, any code *not* authored by Zend or its Partners must never start with "`Zend_`"

Interfaces

- Interface classes must follow the same conventions as other classes (outlined above), *and* end with the word "Interface" (Ex: `Zend_Log_Adapter_Interface`)

Filenames

- For all files, only alphanumeric characters, underscores, and the hyphen character ("-") are permitted; spaces are prohibited
- Any file that contains any PHP code should end with the extension "`.php`", with the exception of View Scripts

ZF PHP CODING STANDARDS

Naming Conventions (continued)

Function Names

- Can only contain alphanumeric characters; underscores are not permitted; numbers are discouraged
- Must always start with a lowercase letter
- Multi-word names: the first letter of the first word is lowercase, the first letter of each subsequent word must be capitalized, known as "`camelCase`"
- Should be explanatory and use enough language as is practical to enhance the understandability of the code
- Accessors for objects should be prefixed with "`get`" or "`set`" (OOP)
- (Note: use of functions in the global scope are discouraged; wrap these functions in a static class instead)

Method Names

- Must always start with a lowercase letter
- Should contain the Pattern name when practical
- For methods declared with a "private" or "protected" construct, the first character of the variable name must be a single underscore (the only time an underscore is permitted in a method name)

Variable Names

- Can only contain alphanumeric characters; underscores are not permitted; numbers are discouraged
- Must always start with a lowercase letter and follow the "camelCase" formatting rules
- For class member variables declared with a "private" or "protected" construct, the first character of the variable name must be a single underscore (the only time an underscore is permitted in a variable name)
- Should be explanatory and use enough language as is practical to enhance the understandability of the code

ZF PHP CODING STANDARDS

Naming Conventions (continued)

Constants

- Can contain alphanumeric characters, underscores, and numbers
- Must always use capitalized letters
- Multi-word names: must separate each word with an underscore (Ex: EMBED_SUPPRESS)
- Must define as class members using the "`const`" construct
- Note: defining constants in the global scope is discouraged

Coding Style

PHP Code Demarcation

- PHP code must be delimited by the full-form, standard PHP tags: `<?php` *and* `?>`
- Short tags are never allowed

Strings - Literals

- When a string is literal (containing no variable substitutions), the apostrophe or "single quote" must be used to demarcate the string (Ex: `$a = 'Example String';`)
- When a literal string itself contains apostrophes, the string can be demarcated with quotation marks " "; this is especially encouraged for SQL statements

Strings - Concatenation

- Strings may be concatenated using the "." operator; a space must always be added before and after the "." operator to improve readability (Ex: `'Zend'.' '.'Tech'`)
- Can break the statement into multiple lines to improve readability; each successive line should be padded with whitespace so that the `".";` operator is aligned under the `"="` operator (Example:

```
$sql  = "SELECT 'id', 'name' FROM 'people' "
      . "WHERE 'name' = 'Susan' "
      . "ORDER BY 'name' ASC ";
```

ZF PHP CODING STANDARDS

Coding Style: (continued)

Arrays – Numerically Indexed

- Negative numbers are not allowed as indices
- Can start with a non-negative number, but discouraged; better to use a base index of 0
- When declared with the `array` construct, a trailing space must be added after each comma delimiter for readability (Example: `$anyArray = array(1, 2, 'Zend');`)
- For multi-line indexed arrays using the `array` construct, each successive line must be padded with spaces so that the beginning of each line aligns as shown:

```
$sampleArray = array(1, 2, 3, 'Zend', 'Studio',
                     $a, $b, $c,
                     56.44, $d, 500)
```

Arrays - Associative

- When declared with the `array` construct, the statement should be broken into multiple lines; each successive line must be padded with whitespace so that both the keys and the values are aligned, as shown:

```
$sampleArray =   array('firstKey'  => 'firstValue',
                       'secondKey' => 'secondValue');
```

Classes - Declaration

- Classes are named following naming convention rules
- The brace is always written on the line underneath the class name ("one true brace" form)
- Code within a class must be indented four spaces (no tabs)
- Restricted to only one class per PHP file
- Placing additional code into a class file is discouraged; if utilized, two blank lines must separate the class from additional PHP code in the file (Example: Class Declaration)

```
/**
 * Documentation Block Here
 */
class SampleClass
{
    // entire content of class must be indented four spaces
}
```

ZF PHP CODING STANDARDS

Coding Style (continued)

Class Member Variables

- Member variables must be named following variable naming conventions (*see Naming section*)
- Variables declared in a class must be listed at the top of the class, prior to declaring any methods
- The `var` construct is not permitted; member variables always declare their visibility by using one of the `private`, `protected`, or `public` constructs
- Accessing member variables directly by making them public is discouraged in favor of accessor methods (`set/get`).

Functions and Methods - Declaration

- Functions must be named following naming convention rules (*see Naming section*)
- Methods inside classes must always declare their visibility by using one of the `private`, `protected`, or `public` constructs
- The brace is always written on the line underneath the function name ("one true brace" form) ; no space between the function name and the opening argument parenthesis
- Functions in the global scope are strongly discouraged
- Pass-by-reference is allowed in the function declaration only; call-time pass-by-reference is prohibited
- Example: Function Declaration in a class

```
/**
 * Documentation Block Here
 */
class Foo
{
    /**
     * Documentation Block Here
     */
    public function bar()
    {
        // entire content of function
        // must be indented four spaces
    }
}
```

ZF PHP CODING STANDARDS

Coding Style (continued)

Functions and Methods - Usage:

- Functions arguments are separated by a single trailing space after the comma delimiter
- Call-time pass-by-reference is prohibited (*see Declarations section*)
- For functions whose arguments permit arrays, the function call may include the "array" construct and can be split into multiple lines to improve readability;; standards for writing arrays still apply; Ex:

```
threeArguments(array(1, 2, 3), 2, 3);

threeArguments(array(1, 2, 3, 'Zend', 'Studio',
                     $a, $b, $c,
                     56.44, $d, 500), 2, 3);
```

Control Statements – if / else / else if:

- Control statements based on the `if` and `else if` constructs must have a single space before the opening parenthesis of the conditional, and a single space after the closing parenthesis
- Within the conditional statements between the parentheses, operators must be separated by spaces for readability inner parentheses are encouraged to improve logical grouping of larger conditionals
- The opening brace is written on the same line as the conditional statement; closing brace is always written on its own line; any content within braces must be indented four spaces

```
if ($a != 2) {
    $a = 2;
}
```

- For "`if`" statements that include "`else if`" or "`else`", the formatting conventions are as shown in the following examples:

```
if ($a != 2) {
    $a = 2;
} else (
    $a = 7;
}
```

```
if ($a != 2) {
    $a = 2;
} else if ($a == 3) {
    $a = 4;
} else {
    $a = 7;
}
```

ZF PHP CODING STANDARDS

Coding Style (continued)

Switch

- Control statements written with the "`switch`" construct must have a single space before the opening parenthesis of the conditional statement, and a single space after the closing parenthesis
- All content within the "`switch`" statement must be indented four spaces; content under each "`case`" statement must be indented an additional four spaces
- All `switch` statements must have a default case

Inline Documentation – Documentation Format

- All documentation blocks ("`docblocks`") must be compatible with the phpDocumentor format
- All source code files written for Zend Framework or that operate with the framework must contain a "file-level" docblock at the top of each file and a "class-level" docblock immediately above each class

Inline Documentation - Files

- Every file that contains PHP code must have a header block at the top of the file that contains these phpDocumentor tags at a minimum:

```
/**
 * Short description for file
 *
 * Long description for file (if any)...
 *
 * LICENSE: Some license information
 *
 * @copyright  2005 Zend Technologies
 * @license    http://www.zend.com/license/3_0.txt   PHP License 3.0
 * @version    $Id:$
 * @link       http://dev.zend.com/package/PackageName
 * @since      File available since Release 1.2.0
*/
```

Inline Documentation - Classes

- Similar to Files, every class must have a docblock that contains these phpDocumentor tags at a minimum - `Descriptions, @copyright, @license, @version, @link, @since, @deprecated`

ZF PHP CODING STANDARDS

Coding Style (continued)

Inline Documentation - Functions

- Every function, including object methods, must have a docblock that minimally contains: a function description; all the arguments; all possible return values
- Not necessary to use the "@access" tag because the access level is already known from the "`public`", "`private`", or "`protected`" construct used to declare the function
- If a function or method might throw an exception, it is best to use "`@throws`"

  Example: @throws exceptionclass [description]

TEST YOUR KNOWLEDGE : QUESTIONS

1

```
Is the following class name valid?: My_Foo_123
   a. Yes
   b. No
```

2

```
The filename "Zend/Db/Table.php" must map to the class name
_________.?
```

TEST YOUR KNOWLEDGE : ANSWERS

★ = CORRECT

1

Is the following class name valid?: My_Foo_123

★ a. Yes

b. No

2

The filename "Zend/Db/Table.php" must map to the class name __________.?

★ Zend_Db_Table

CERTIFICATION TOPIC : DATABASES

Zend_Db

Definitions · *Use / Purpose* · *Adapters*

Constructors · *Lazy Connections*

Zend_Db_Statement

Definitions · *Use / Purpose* · *Named Parameters*

Zend_Db_Select

Definitions · *Use / Purpose* · *Fluent Interface*

Quoting · *Complex Queries* · *Zend_Db : : factory()*

Zend_Db_Table

Definitions ●··● *Use / Purpose* ●··● *Primary Keys*

Application Logic ●··● *Naming* ●··● *Methods*

Zend Framework - Databases

For the exam, here's what you should know already …

You should be able to create and work with `Zend_Db` component extensions, such as `_Adapter`, `_Statement`, `_Table`, and `_Library`.

You should know how to connect to a database, in particular using constructor arguments and lazy connections.

You should know how to specify database configurations within a Configuration file.

You should be able to fetch data in rows, columns, and individually, and be able to utilize different fetch modes. This includes the ability to fetch data from content returned by executing a statement. You should know how to construct complex queries.

You should be able to manipulate data within a database (insert, modify, delete).

You should know what a Fluent Interface is, including how and when to use it.

Methods:
You should know how to insert application logic into the appropriate method.

You should know how and when to use the `quoteIdentifier` method.

You should understand how to utilize the `Zend_Db::factory()` options.

ZEND_DB

Zend_Db and its related classes provide a simple SQL database interface for Zend Framework.

`Zend_Db_Adapter_Abstract` is the base class for connecting PHP applications to a RDBMS. They create a bridge from vendor-specific PHP extensions to a common interface, so that the PHP applications can be written once but deployed multiple times according to the RDBMS with little change or effort.

Connecting to a Database

Using an Adapter and Constructor

You create an instance of the Adapter using its constructor, which takes one argument in an array of parameters used to declare the connection. Note: `Zend_Db_Adapter` uses a PHP extension that must be enabled in the PHP environment.

```
<?php

require_once 'Zend/Db/Adapter/Pdo/Mysql.php';

$db = new Zend_Db_Adapter_Pdo_Mysql(array(
    'host'     => '127.0.0.1',
    'username' => 'webuser',
    'password' => 'xxxxxxxx',
    'dbname'   => 'test'
));
```

Using Zend_Db: :factory

The Factory method of connection is an alternative to the use of the Constructor. Instead, the Adapter instance is created using the static method `Zend_Db::factory()`, which dynamically loads the Adapter class file on demand utilizing `Zend_Loader::loadClass()`.

```
<?php

require_once 'Zend/Db.php';

// Automatically load class Zend_Db_Adapter_Pdo_Mysql and create
// an instance of it.
$db = Zend_Db::factory('Pdo_Mysql', array(
    'host'     => '127.0.0.1',
    'username' => 'webuser',
    'password' => 'xxxxxxxx',
    'dbname'   => 'test'
));
```

ZEND_DB

If you create your own class that extends `Zend_Db_Adapter_Abstract`, but you do not name your class with the "`Zend_Db_Adapter`" package prefix, you can use the `factory()` method to load your Adapter, providing you specify the leading portion of the adapter class with the `'adapterNamespace'` key in the parameters array.

Using Zend_Config with the Zend_DB Factory

You can specify either argument of the `factory()` method as an object of type `Zend_Config`. If the first argument is a Config object, it is expected to contain a property named `'adapter'`, which contains the string providing the adapter class name base. Optionally, the object may contain a property named `'params'`, with sub-properties corresponding to adapter parameter names. This is used *only* if the second argument of the `factory()` method is absent. Example:

```
<?php

require_once 'Zend/Config.php';
require_once 'Zend/Db.php';

$config = new Zend_Config(
    array(
        'database' => array(
            'adapter' => 'Mysqli',
            'params' => array(
                'dbname' => 'test',
                'username' => 'webuser',
                'password' => 'secret',
            )
        )
    )
);

$db = Zend_Db::factory($config->database);
```

If an optional second argument of the `factory()` method exists, it can be an associative array containing entries corresponding to adapter parameters. If the first argument is of the type `Zend_Config`, it is assumed to contain all parameters, and the second argument is ignored.

ZEND_DB

Passing Options to the Factory
Presented below is a code example of how to pass an option to the Factory – in this case, it is the option for automatically quoting identifiers. You can find other examples in the Reference Guide.

```
<?php
$options = array(
    Zend_Db::AUTO_QUOTE_IDENTIFIERS => false
);

$params = array(
    'host'              => '127.0.0.1',
    'username'          => 'webuser',
    'password'          => 'xxxxxxxx',
    'dbname'            => 'test',
    'options'           => $options
);

$db = Zend_Db::factory('Pdo_Mysql', $params);
```

Managing Lazy Connections
Creating an instance of an Adapter class results in the Adapter saving the connection parameters, but the actual connection is on demand, triggered the first time a query is executed. This ensures that creating an Adapter object is quick and not resource intensive.

To force the Adapter to connect to the RDBMS, use the `getConnection()` method, which returns an object for the connection represented by the required PHP database extension. For example, using any of the Adapter classes for PDO drivers will result in `getConnection()` returning the PDO object, after initiating it as a live connection to the specific database. Forcing a connection can be a useful tool in capturing exceptions thrown by RDBMS connection failures (Ex: invalid account credentials).

It can be useful to force the connection if you want to catch any exceptions it throws as a result of invalid account credentials, or other failure to connect to the RDBMS server. These exceptions are not thrown until the connection is made, so it can help simplify your application code if you handle the exceptions in one place, instead of at the time of the first query against the database.

ZEND_DB

Fetching Data

(Note: examples in this section use the ZF Sample Bug-Tracking Database, shown in the Programmers Guide)

You can run a SQL SELECT query and retrieve its results in one step using the `fetchAll()` method. The first argument to this method is a string containing a SELECT statement. Alternatively, the first argument can be an object of class `Zend_Db_Select`. The Adapter automatically converts this object to a string representation of the SELECT statement.

The second argument to `fetchAll()` is an array of values to substitute for parameter placeholders in the SQL statement.

```
<?php
$sql = 'SELECT * FROM bugs WHERE bug_id = ?';
$result = $db->fetchAll($sql, 2);
```

Changing the Fetch Mode

By default, `fetchAll()` returns an array of rows, each of which is an associative array. The keys of this array are the columns/column aliases named in the select query. You can specify a different style of fetching results using the `setFetchMode()` method. The modes supported are identified by constants:

- `Zend_Db::FETCH_ASSOC` **Default** Fetch mode for Adapter classes
  returns data in an associative array; the keys are the column names, as strings

- `Zend_Db::FETCH_BOTH`
  returns data in an array of arrays; the keys are strings used in the `FETCH_ASSOC` *mode, and integers as used in the* `FETCH_NUM` *mode*

- `Zend_Db::FETCH_COLUMN`
  returns data in an array of values, where each value is returned by one column of the result set; by default, the first column is used and indexed by 0

- `Zend_Db::FETCH_OBJ`
  returns data in an array of objects; the default class is the PHP built-in class `stdClass`*; columns of the result set are available as public properties of the object*

ZEND_DB

Changing Fetch Mode – Example: (more examples in Reference Guide)

```
<?php
$db->setFetchMode(Zend_Db::FETCH_OBJ);

$result = $db->fetchAll('SELECT * FROM bugs WHERE bug_id = ?', 2);

// $result is an array of objects
echo $result[0]->bug_description;
```

Inserting Data

New rows can be added to a table in the database using the `insert()` method. The first argument is a string naming the table, the second argument an associative array mapping column names to data values. Example:

```
<?php
$data = array(
    'created_on'      => '2007-03-22',
    'bug_description' => 'Something wrong',
    'bug_status'      => 'NEW'
);

$db->insert('bugs', $data);
```

Quoting Values and Identifiers

SQL queries often need to include the values of PHP variables in SQL expressions, which can be risky if a value in the PHP string contains certain symbols, such as the quote symbol. This would not only result in invalid SQL, but consequently would impose a security risk (*see the Security section*).

The `Zend_Db_Adapter` class provides convenient functions to help reduce vulnerabilities to SQL Injection attacks in PHP code. The solution is to escape special characters such as quotes in PHP values before they are interpolated into SQL strings. This protects against both accidental and deliberate manipulation of SQL strings by PHP variables that contain special characters.

ZEND_DB

Quoting Values and Identifiers (continued)

- Using `quote()`

The `quote()` method accepts a single argument, a scalar string value returned with special characters escaped according to the RDBMS being used, and surrounded by string value delimiters. The SQL string value delimiter for MySQL is the single-quote (`'`).

```
<?php
$name = $db->quote("O'Reilly");
echo $name;
// 'O\'Reilly'

$sql = "SELECT * FROM bugs WHERE reported_by = $name";

echo $sql;
// SELECT * FROM bugs WHERE reported_by = 'O\'Reilly'
```

- Using `quoteInto()`

You can use the `quoteInto()` method to interpolate a PHP variable into a SQL expression or statement. It takes two arguments: the first is a string containing a placeholder symbol (?), and the second is a value or PHP variable that should be substituted for that placeholder.

The placeholder symbol is the same symbol used by many RDBMS brands for positional parameters, but the `quoteInto()` method only emulates query parameters. The method simply interpolates the value into the string, escapes special characters, and applies quotes around it. True query parameters maintain the separation between the SQL string and the parameters as the statement is parsed in the RDBMS server.

```
<?php
$sql = $db->quoteInto("SELECT * FROM bugs WHERE reported_by = ?",
                      "O'Reilly");

echo $sql;

// SELECT * FROM bugs WHERE reported_by = 'O\'Reilly'
```

ZEND_DB

Quoting Values and Identifiers (continued)

- Using `quoteIdentifier()`

As with values, if PHP variables are used to name tables, columns, or other identifiers in SQL statements, the related strings may need to be quoted. By default, SQL identifiers have syntax rules like PHP and most other programming languages. For example, identifiers should not contain spaces, certain punctuation or special characters, or international characters. Also certain words are reserved for SQL syntax, and should not be used as identifiers.

However, SQL has a feature called *delimited identifiers*, which allows broader choices for the spelling of identifiers. If you enclose a SQL identifier in the proper type of quotes, you can use identifiers with spellings that would be invalid without the quotes. Delimited identifiers can contain spaces, punctuation, or international characters. You can also use SQL reserved words if you enclose them in identifier delimiters.

The `quoteIdentifier()` method works like `quote()`, but applies the identifier delimiter characters to the string according to the type of Adapter used. For example, standard SQL uses double-quotes (`"`) for identifier delimiters, and most RDBMS brands use that symbol. MySQL uses back-quotes (`` ` ``) by default. The `quoteIdentifier()` method also escapes special characters within the string argument. SQL-delimited identifiers are case-sensitive, unlike unquoted identifiers. Therefore, the use of delimited identifiers require that the identifier is spelled exactly as it is stored in the schema, including the case of the letters.

In most cases where SQL is generated within `Zend_Db` classes, the default is that all identifiers are delimited automatically; this default can be changed by using `Zend_Db::AUTO_QUOTE_IDENTIFIERS` when instantiating the Adapter.

```
<?php
// we might have a table name that is an SQL reserved word
$tableName = $db->quoteIdentifier("order");

$sql = "SELECT * FROM $tableName";

echo $sql
// SELECT * FROM "order"
```

ZEND_DB

ZEND_DB_STATEMENT

Like the methods `fetchAll()`and `insert()`, a statement object can be used to gain more options for running queries and fetching result sets. `Zend_Db_Statement` is based on the `PDOStatement` object in the PHP Data Objects extension.

Creating a Statement

Typically, a statement object is returned by the `query()` method of the database Adapter class. This method is a general way to prepare any SQL statement. The first argument is a string containing an SQL statement. The optional second argument is an array of values to bind to parameter placeholders in the SQL string. The statement object corresponds to a SQL statement that has been prepared and executed once with the bind-values specified. If the statement was a SELECT query or other type of statement that returns a result set, it is now ready to fetch results. A statement can be created with its constructor, but this is less typical. Note: there is no factory method to create this object.

```
<?php
$stmt = $db->query('SELECT * FROM bugs WHERE reported_by = ? AND
        bug_status = ?', array('goofy', 'FIXED'));
```

Executing a Statement

If a statement is created using its constructor, or if a statement is to be executed multiple times, then the statement object needs to be executed. Use the `execute()` method of the statement object. The single argument is an array of values to bind to parameter placeholders in the statement.

If positional parameters, or those that are marked with a question mark symbol (?), are used, pass the bind values in a *plain* array. If named parameters, or those that are indicated by a string identifier preceded by a colon character (`:`), pass the bind values in an *associative* array. The keys of this array should match the parameter names. Execution (positional) example:

```
<?php
$sql = 'SELECT * FROM bugs WHERE reported_by = ? AND
        bug_status = ?';

$stmt = new Zend_Db_Statement_Mysqli($db, $sql);
$stmt->execute(array('goofy', 'FIXED'));
```

ZEND_DB

Fetching Content

You can call methods on the statement object to retrieve rows from SQL statements that produce a result set – for example, SELECT, SHOW, DESCRIBE, EXPLAIN.

INSERT, UPDATE, and DELETE are examples of statements that do not produce a result set – these SQL statements can be executed using Zend_Db_Statement, but cannot have results fetched by method calls. Code Example: fetch() in a loop

```
<?php
$stmt = $db->query('SELECT * FROM bugs');
while ($row = $stmt->fetch()) {
    echo $row['bug_description'];
}
```

ZEND_DB_SELECT

The Zend_Db_Select object represents a SQL SELECT query statement. The class has methods for adding individual parts to the query. Some parts of the query can be specified using PHP methods and data structures, and the class will form the correct SQL syntax. After the query is built, it can be executed as if written as a string. Important features of this component include:

- Object-oriented methods for specifying SQL queries in a piece-by-piece manner
- Database-independent abstraction of some parts of the SQL query
- Automatic quoting of metadata identifiers in most cases, to support identifiers containing SQL reserved words and special characters
- Quoting identifiers and values, to help reduce risk of SQL injection attacks

Creating a Select Object

To create an instance of a Zend_Db_Select object, use the select() method of a Zend_Db_Adapter_Abstract object. Ex: Adapter select() method

```
<?php
$db = Zend_Db::factory( ...options... );
$select = $db->select();
```

ZEND_DB

Building Select Queries

When building a query, clauses can be added to the `Zend_Db_Select` object individually, using a separate method for each.

```
<?php
// Create the Zend_Db_Select object
$select = $db->select();

// Add a WHERE clause
$select->where( ...specify search criteria... )

// Add an ORDER BY clause
$select->order( ...specify sorting criteria... );
```

Using a Fluent Interface

A fluent interface means that each method returns a reference to the object on which it was called, so that another method can be immediately called. Ex:

```
<?php
$select = $db->select()
    ->from( ...specify table and columns... )
    ->where( ...specify search criteria... )
    ->order( ...specify sorting criteria... );
```

Creating Complex Queries

Zend Framework provides numerous ways to add, modify, and remove data utilizing its components and extensibility. Presented below are just some of the ways to handle data – additional information and code examples can be found in the Programmers Guide.

- Add `FROM` Clause: Specify the table name as a simple string; `Zend_Db_Select` applies identifier quotes around the table name so that special characters can be used.

```
<?php
// Build this query:
//   SELECT *
//   FROM "products"

$select = $db->select()
    ->from( 'products' );
```

ZEND_DB

- Specify Schema: Some RDBMS brands support a schema specifier for a table. Specify the table name as `schemaName.tableName`, where `Zend_Db_Select` quotes each part individually; alternatively specify the schema name separately. A schema name specified in the table name takes precedence over a schema provided separately in the event that both are presented.

```
<?php
// Build this query:
//   SELECT *
//   FROM "myschema"."products"

$select = $db->select()
    ->from( 'myschema.products' );

// or

$select = $db->select()
    ->from('products', '*', 'myschema');
```

- Add Column: The columns to be selected from a table can be specified in the second argument of `from()`. The default is the wildcard *, meaning all columns.

```
<?php
// Build this query:
//   SELECT p."product_id", p."product_name"
//   FROM "products" AS p
$select = $db->select()
    ->from(array('p' => 'products'),
        array('product_id', 'product_name'));

// Build the same query, specifying correlation names:
//   SELECT p."product_id", p."product_name"
//   FROM "products" AS p
$select = $db->select()
    ->from(array('p' => 'products'),
        array('p.product_id', 'p.product_name'));

// Build this query with an alias for one column:
//   SELECT p."product_id" AS prodno, p."product_name"
//   FROM "products" AS p
$select = $db->select()
    ->from(array('p' => 'products'),
        array('prodno' => 'product_id', 'product_name'));
```

ZEND_DB

- Add Table:

Many useful queries involve using a `JOIN` to combine rows from multiple tables; tables can also be added to a `Zend_Db_Select` query using the `join()` method. Using this method is similar to the `from()` method except a condition can also be specified in most cases. The second argument to `join()` is a string that is the join condition, an expression that declares the criteria by which rows in one table match rows in the the other table.

```
<?php
// Build this query:
//   SELECT p."product_id", p."product_name", l.*
//   FROM "products" AS p JOIN "line_items" AS l
//   ON p.product_id = l.product_id

$select = $db->select()
    ->from(array('p' => 'products'),
        array('product_id', 'product_name'))
    ->join(array('l' => 'line_items'),
        'p.product_id = l.product_id');
```

- Add WHERE clause:

The `where()` method can specify criteria for restricting rows of the result set. The first argument of this method is a SQL expression, which is used in a SQL WHERE clause in the query.

```
<?php
// Build this query:
//   SELECT product_id, product_name, price
//   FROM "products"
//   WHERE price > 100.00

$select = $db->select()
    ->from(
        'products',
        array('product_id', 'product_name', 'price'))
    ->where('price > 100.00');
```

ZEND_DB

- Add Order By Clause: Use Zend_Db_Select with the order() method to specify a column or an array of columns by which to sort. Each element of the array is a string naming a column, with the optional ASC or DESC keyword following it, separated by a space. As with from() and group(), column names are quoted as identifiers, unless they contain parentheses or are an object of type Zend_Db_Expr.

```
<?php
// Build this query:
//   SELECT p."product_id", COUNT(*) AS line_items_per_product
//   FROM "products" AS p JOIN "line_items" AS l
//     ON p.product_id = l.product_id
//   GROUP BY p.product_id
//   ORDER BY "line_items_per_product" DESC, "product_id"

$select = $db->select()
    ->from(array('p' => 'products'),
        array('product_id'))
    ->join(array('l' => 'line_items'),
        'p.product_id = l.product_id',
        array('line_items_per_product' => 'COUNT(*)'))
    ->group('p.product_id')
    ->order(array('line_items_per_product DESC', 'product_id'));
```

- Add LIMIT clause: Use the limit() method in Zend_Db_Select to specify the count of rows and the number of rows to skip. The first argument to this method is the desired count of rows. The second argument is the number of rows to skip. Note that the LIMIT syntax is not supported by all RDBMS brands.

```
<?php
// Build this query:
//   SELECT p."product_id", p."product_name"
//   FROM "products" AS p
//   LIMIT 10, 20

$select = $db->select()
   ->from(array('p' => 'products'), array('product_id',
         'product_name'))
   ->limit(10, 20);
```

ZEND_DB

ZEND_DB_TABLE

The `Zend_Db_Table` class is an object-oriented interface to database tables. It provides methods for many common operations on tables. The base class is extensible, allowing the addition of custom logic. The `Zend_Db_Table` solution is an implementation of the Table Data Gateway pattern and includes a class that implements the Row Data Gateway pattern.

Defining a Table Class

To access a table in the database, define a class that extends `Zend_Db_Table_Abstract`.

Defining a Table Name and Schema

Declare the database table for which the class is defined, using the protected variable `$_name`. This is a string, with the name of the table spelled exactly as it appears in the database. If no table name is supplied, it defaults to the name of the class (which also must exactly match the spelling of the table name in the database).

Example: Declare Table Class with Explicit Table Name

```php
<?php
class Bugs extends Zend_Db_Table_Abstract
{
    protected $_name = 'bugs';
}
```

Example: Declare Table Class with Implicit Table Name

```php
<?php
class Bugs extends Zend_Db_Table_Abstract
{
    //table name matches class name, by default
}
```

ZEND_DB

Overriding Table Setup Methods

When an instance of a Table class is created, the constructor calls a set of protected methods that initialize metadata for the table. These methods can be extended to define metadata explicitly. Note: remember to call the method of the same name in the parent class at the end of the method. Code Example:

```
<?php
class Bugs extends Zend_Db_Table_Abstract
{
    protected function _setupTableName()
    {
        $this->_name = 'bugs';
        parent::_setupTableName();
    }
}
```

The setup methods that can be overridden are:

- `_setupDatabaseAdapter()`
  checks that an adapter has been provided; gets a default adapter from the registry if needed. By overriding this method, you can set a database adapter from some other source.

- `_setupTableName()`
  defaults the table name to the name of the class. By overriding this method, you can set the table name before this default behavior runs.

- `_setupMetadata()`
  sets the schema if the table name contains the pattern "`schema.table`"; calls `describeTable()` to get metadata information; defaults `$_cols` array to the columns reported by `describeTable()`. Override this method to specify the columns.

- `_setupPrimaryKey()`
  defaults the primary key columns to those reported by `describeTable();` checks that the primary key columns are included in the `$_cols` array. Override this method to specify the primary key columns.

TEST YOUR KNOWLEDGE : QUESTIONS

1 Zend_Db contains a factory() method by which you may instantiate a database adapter object

a. True

b. False

2 Zend_Db_Select supports querying columns containing expressions (e.g., LOWER(someColumn))

a. True

b. False

TEST YOUR KNOWLEDGE : ANSWERS

★ = CORRECT

1

```
Zend_Db contains a factory() method by which you may
instantiate a database adapter object
```

★ a. True
b. False

2

```
Zend_DB_Select supports querying columns containing
expressions (e.g., LOWER(someColumn))
```

★ a. True
b. False

CERTIFICATION TOPIC : DIAGNOSIS & MAINTENANCE

Zend_Log

Use / Purpose ●··● *Objects & Events* ●··● *Writer*

Formatters & Filters

Zend_Debug

Use / Purpose ●··● *Dump Method*

Zend Framework: Diagnosis & Maintenance

For the exam, here's what you should know already …

You should be able to explain when and how you would use `Zend_Log`, including the creation and destruction of Log objects.

You should understand how backends, filters, and formatters work within the Framework.

In addition, you should understand the difference between using `Zend_Log` and `Zend_Debug`, and when it is appropriate to use each.

ZEND_LOG

The Zend_Log component supports multiple log backends, formatting messages sent to the log, and filtering messages from being logged. These functions are divided into the following objects:

- **Log** (instance of Zend_Log)
  The object most used by an application; unlimited in number; Log object must contain at least one Writer, and can optionally contain one or more Filters

- **Writer (**inherits from Zend_Log_Writer_Abstract**)**
  Responsible for saving data to storage.

- **Filter (**implements Zend_Log_Filter_Interface**)**
  Prevents log data from being saved; a filter may be applied to an individual Writer, or to a Log where it is applied before all Writers; Filters can be chained

- **Formatter (**implements Zend_Log_Formatter_Interface)
  Formats the log data before it is written by a Writer; each Writer has exactly one Formatter

Create a Log

To create a log, instantiate a Writer (using Zend_Log_Writer_Stream) and then pass it on to a Log instance. More Writers can be added with addWriter(); one is required. Ex:

```
<?php
$logger = new Zend_Log();
$writer = new Zend_Log_Writer_Stream('php://output');
```

Logging Messages

Call the log() method of a Log instance and pass it the message with a corresponding priority; the first parameter is a string message; the second, an integer priority. Ex:

```
<?php
$logger->log('Informational message', Zend_Log::INFO);
```

ZEND_LOG

Destroy a Log

Set the variable containing it to `null` to destroy it; this will automatically call the `shutdown()` instance method of each attached Writer before the Log object is destroyed:

```
<?php
$logger = null;
```

Priorities – Built-In

• `EMERG = 0;`	Emergency: system is unusable
• `ALERT = 1;`	Alert: action must be taken immediately
• `CRIT = 2;`	Critical: critical conditions
• `ERR = 3;`	Error: error conditions
• `WARN = 4;`	Warning: warning conditions
• `NOTICE = 5;`	Notice: normal but significant condition
• `INFO = 6;`	Informational: informational messages
• `DEBUG = 7;`	Debug: debug messages

Priorities – Custom

User-defined priorities can be added at runtime using the Log's `addPriority()` method.
Example: Create Priority called FOO and log the priority

```
<?php
$logger->addPriority('FOO', 8);
```

```
<?php
$logger->log('Foo message', 8);
```

ZEND_LOG

Writers

A Writer is an object that inherits from `Zend_Log_Writer_Abstract`, and is responsible for recording log data to a storage backend.

Writing to 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`. To write to file, use a Filesystem URL. By default, the stream opens in the append mode – for another mode, provide an optional second parameter to the constructor.

Alternatively, you can send log data directly to a stream like STDERR `(php://stderr)`. Ex:

```
<?php
$writer = new Zend_Log_Writer_Stream('php://output');
$logger = new Zend_Log($writer);
```

Writing to Databases:

`Zend_Log_Writer_Db` writes log information to a database table using `Zend_Db`. The constructor of `Zend_Log_Writer_Db` receives a `Zend_Db_Adapter` instance, a table name, and a mapping of database columns to event data items. Ex:

```
<?php
$params = array ('host'     => '127.0.0.1',
                 'username' => 'malory',
                 'password' => '******',
                 'dbname'   => 'camelot');
$db = Zend_Db::factory('PDO_MYSQL', $params);

$columnMapping = array('lvl' => 'priority', 'msg' => 'message');
$writer = new Zend_Log_Writer_Db($db, 'log_table_name',
                                 $columnMapping);
$logger = new Zend_Log($writer);
$logger->info('Informational message');

$logger->info('Informational message');
```

ZEND_LOG

Writers (CONTINUED)

Testing with the Mock

The Zend_Log_Writer_Mock is a simple writer that records the raw data it receives in an array exposed as a public property. Ex:

```
<?php
$mock = new Zend_Log_Writer_Mock;
$logger = new Zend_Log($mock);

$logger->info('Informational message');

var_dump($mock->events[0]);

// Array
// (
//     [timestamp] => 2007-04-06T07:16:37-07:00
//     [message] => Informational message
//     [priority] => 6
//     [priorityName] => INFO
// )
```

The logged events can be cleared by simply setting `$mock->events = array()`.

ZEND_LOG

Formatters

A Formatter is an object that is responsible for taking an `event` array describing a log event and outputting a string with a formatted log line.

Some Writers are not line-oriented and cannot use a Formatter (Ex: the Database Writer, which inserts the event items directly into database columns). For Writers that cannot support a Formatter, an exception is thrown if you attempt to set a Formatter.

Simple Formatting

`Zend_Log_Formatter_Simple` is the default formatter. It is configured automatically when you specify no formatter. The default configuration is equivalent to the following:

```
<?php
$format = '%timestamp% %priorityName% (%priority%):
            %message%' . PHP_EOL;
$formatter = new Zend_Log_Formatter_Simple($format);
```

A formatter is set on an individual Writer object using the Writer's `setFormatter()` method:

Formatting to XML

`Zend_Log_Formatter_Xml` formats log data into XML strings. By default, it automatically logs all items in the event data array. Ex:

```
<?php
$format = '%timestamp% %priorityName% (%priority%):
            %message%' . PHP_EOL;
$formatter = new Zend_Log_Formatter_Simple($format);
```

It is possible to customize the root element as well as specify a mapping of XML elements to the items in the event data array. The constructor of `Zend_Log_Formatter_Xml` accepts a string with the name of the root element as the first parameter, and an associative array with the element mapping as the second parameter:

ZEND_LOG

Filters

A filter object blocks a message from being written to the log.

Filtering for All Writers

To filter before all writers, you can add any number of Filters to a Log object using the `addFilter()` method:

```
<?php
$logger = new Zend_Log();

$writer = new Zend_Log_Writer_Stream('php://output');
$logger->addWriter($writer);

$filter = new Zend_Log_Filter_Priority(Zend_Log::CRIT);
$logger->addFilter($filter);

// blocked
$logger->info('Informational message');

// logged
$logger->emerg('Emergency message');
```

When you add one or more Filters to the Log object, the message must pass through all of the Filters before any Writers receives it.

Filtering for a Writer Instance

To filter only on a specific Writer instance, use the `addFilter()` method of that Writer:

```
<?php
$logger = new Zend_Log();

$writer1 = new Zend_Log_Writer_Stream('/path/to/first/logfile');
$logger->addWriter($writer1);

$writer2 = new Zend_Log_Writer_Stream('/path/to/second/logfile');
$logger->addWriter($writer2);

// add a filter only to writer2
$filter = new Zend_Log_Filter_Priority(Zend_Log::CRIT);
$writer2->addFilter($filter);

// logged to writer1, blocked from writer2
$logger->info('Informational message');

// logged by both writers
$logger->emerg('Emergency message');
```

ZEND_DEBUG

Dumping Variables

The static method `Zend_Debug::dump()` prints or returns information about an expression. This simple technique of debugging is common, because it is easy to use in an *ad hoc* fashion, and requires no initialization, special tools, or debugging environment.

The `$var` argument specifies the expression or variable about which the `Zend_Debug::dump()` method outputs information.

The `$label` argument is a string to be prepended to the output of `Zend_Debug::dump()`. Best Practice: use labels if you are dumping information about multiple variables on a given screen.

The boolean `$echo` argument specifies whether the output of `Zend_Debug::dump()` is echoed or not. If `true`, the output is echoed. Regardless of the value of the `$echo` argument, the return value of this method contains the output.

```
<?php
Zend_Debug::dump($var, $label=null, $echo=true);
```

It may be helpful to understand that internally, `Zend_Debug::dump()`wraps the PHP function `var_dump()`. If the output stream is detected as a web presentation, the output of `var_dump()` is escaped using `htmlspecialchars()` and wrapped with (X)HTML `<pre>` tags.

Using `Zend_Debug::dump()` is best for ad hoc debugging during software development add code to dump a variable and then remove the code very quickly.

Also consider the `Zend_Log` component when writing more permanent debugging code. Ex: use the DEBUG log level and the Stream log writer, to output the string returned by `Zend_Debug::dump()`.

TEST YOUR KNOWLEDGE : QUESTIONS

1

Which ONE of the following will NOT display the value of $var?

```
a. echo Zend_Debug::dump($var, 'var', false);

b. ob_start();
   Zend_debug::dump($var, 'var', false);
   ob_end_flush();

c. Zend_Debug::dump($var, 'var', true);

d. Zend_Debug::dump($var, 'var');
```

2

Which formatters are provided with Zend_Log? (choose two)?

```
a. Zend_Log_Formatter_Db

b. Zend_Log_Formatter_Simple

c. Zend_Log_Formatter_Text

d. Zend_Log_Formatter_Xml
```

TEST YOUR KNOWLEDGE : ANSWERS

★ = CORRECT

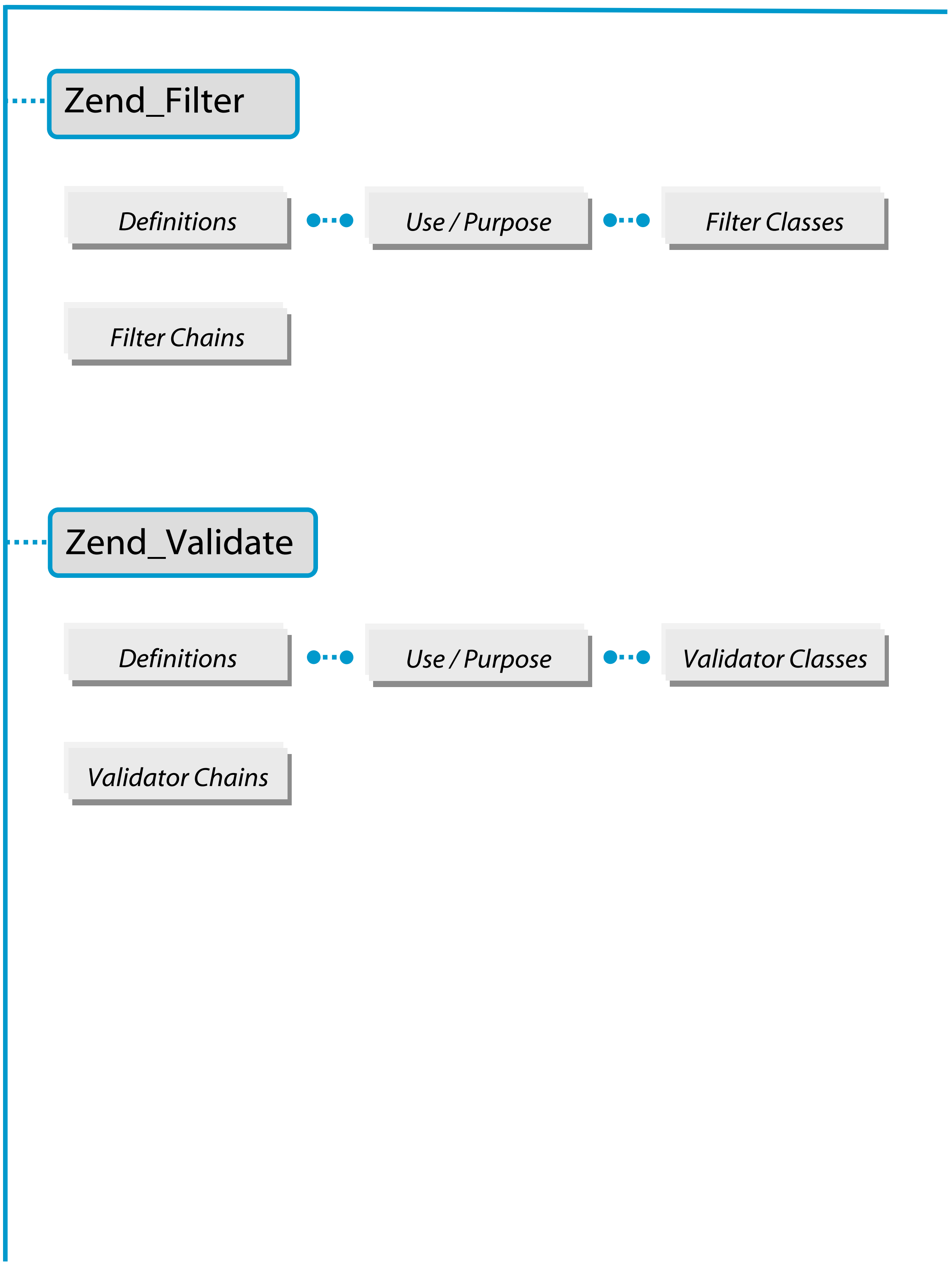
1

```
Which ONE of the following will NOT display the value of
$var?
   a. echo Zend_Debug::dump($var, 'var', false);

★  b. ob_start();
      Zend_debug::dump($var, 'var', false);
      ob_end_flush();

   c. Zend_Debug::dump($var, 'var', true);

   d. Zend_Debug::dump($var, 'var');
```

2

```
Which formatters are provided with Zend_Log? (choose two)?


   a. Zend_Log_Formatter_Db

★  b. Zend_Log_Formatter_Simple

   c. Zend_Log_Formatter_Text

★  d. Zend_Log_Formatter_Xml
```

CERTIFICATION TOPIC : FILTERING & VALIDATION



Zend Framework: Filtering & Validation

For the exam, here's what you should know already …

You should be able to create and work with Filters, while understanding their role within an application.

In addition, you should be able to create and utilize Validators.

ZEND_FILTER

Filters, in the general sense, function to produce some subset of the original input, based on some criteria or transformation. Within web applications, filters are used to remove illegal input, trim unnecessary white space, and perform other functions.

A common transformation applied within web application is to escape HTML entities (*see Security section for more information on this sub-topic*).

`Zend_Filter` provides a set of commonly needed data filters for web applications. It also provides a simple filter chaining mechanism by which multiple filters can be applied to a single datum in a user-defined order.

`Zend_Filter_Interface` provides the basic functionality for the component, and requires a single method `filter()`, to be implemented by a filter class. You can also use the static method `Zend_Filter::get()` as an alternative.

Here is an example of a filter that transforms ampersands and quotes:

```
<?php
require_once 'Zend/Filter/HtmlEntities.php';

$htmlEntities = new Zend_Filter_HtmlEntities();

echo $htmlEntities->filter('&'); //&amp;
echo $htmlEntities->filter('"'); //&quot;
```

Standard Filters
ZF offers a set of standard filter classes with the framework – the list is provided below. Information on each option can be found in the Programmers Guide – you should know how to utilize each.

- `Alnum`
- `Alpha`
- `BaseName`
- `Digits`
- `Dir`
- `HtmlEntities`
- `Int`
- `RealPath`
- `StringToLower`
- `StringToUpper`
- `StringTrim`
- `StripTags`

ZEND_FILTER

Filter Chains

When using multiple filters, it will often be necessary to impose them in a specific order, known as 'chaining'. `Zend_Filter` provides a simple way to chain filters together. Filters are run in the order in which they were added to the component.

Here is an example of filtering a username so it is transformed to be lowercase with alphabetic characters. Note that the non-alphabetic characters are removed before any uppercase characters conversion because of the order of filters within the code.

```
<?php
// Provides filter chaining capability
require_once 'Zend/Filter.php';

// Filters needed for the example
require_once 'Zend/Filter/Alpha.php';
require_once 'Zend/Filter/StringToLower.php';

// Create a filter chain and add filters to the chain
$filterChain = new Zend_Filter();
$filterChain->addFilter(new Zend_Filter_Alpha())
            ->addFilter(new Zend_Filter_StringToLower());

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

Creating Custom Filters

Creating a filter is a simple process - a class needs only implement the component `Zend_Filter_Interface`, consisting of the single method `filter()`. This method can be implemented by user classes. Objects that implement this interface can be added to a filter chain using `Zend_Filter::addFilter()`. Sample Code provided below:

```
<?php
require_once 'Zend/Filter/Interface.php';

class MyFilter implements Zend_Filter_Interface
{
    public function filter($value)
    {
// perform transformation upon $value to arrive on $valueFiltered

        return $valueFiltered;
    }
}
```

ZEND_FILTER

Zend_Filter_Input

`Zend_Filter_Input` provides a declarative interface to associate multiple filters and validators, apply them to collections of data, and retrieve input values after they have been processed by the filters and validators. Values are returned in escaped format by default for safe HTML output.

- **Filters** transform input values, by removing or changing characters within the value. The goal is to "normalize" input values until they match an expected format
- **Validators** check input values against criteria and report whether they passed the test or not. The value is not changed, but the check may fail
- **Escapers** transform a value by removing magic behavior of certain/special characters, which have meaning in some output contexts

The process for using `Zend_Filter_Input`:

Declare filter and validator rules

Create the filter and validator processor

Provide input data

Retrieve validated fields and other reports

For further information about these steps, see the online Reference Guide

ZEND_VALIDATE

Validators examine input against some set of requirements and produces a boolean result (True or False) as to whether it passed. It can also provide further information about which requirement(s) the input did not meet in the event of validation failure.

`Zend_Validate` provides a set of commonly needed validators for web applications. It also provides a simple validator chaining mechanism by which multiple validators can be applied to a single datum in a user-defined order.

`Zend_Validate_Interface` provides the basic functionality for the component, and defines two possible methods.

> `isValid()` performs validation upon the input value, returning TRUE for success in meeting the criteria, FALSE for failure. In the latter case, the next method is used.
>
> `getMessages()` returns an array of messages explaining the reason(s) for validation failure. The array *keys* identify the reasons with string codes, and the array *values* provide readable string messages. These key-value pairs are class-dependent, and each class also has a `const` definition that matches each identifier for a validation failure cause.
>
> Note that each call to `isValid()` overwrites the previous call. Therefore, each `getMessages()` output will apply to only the most recent validation.

```php
<?php
require_once 'Zend/Validate/EmailAddress.php';

$validator = new Zend_Validate_EmailAddress();

if ($validator->isValid($email)) {
    // email appears to be valid
} else {
    // email is invalid; print the reasons
    foreach($validator->getMessages()as $messageId => $message) {
        echo "Validation failure '$messageId': $message\n";
    }
}
```

ZEND_VALIDATE

You can also use the static method `Zend_Validate::is()` as an alternative. The first argument would be the data input value, passed to the `isValid()` method, and the second argument would be the string that corresponds to the basename of the validation class, relative to the `Zend_Validate` namespace. The `is()` method automatically loads the class, creates an instance, and applies the `isValid()` method to the data input.

```
<?php
require_once 'Zend/Validate.php';

if (Zend_Validate::is($email, 'EmailAddress')) {
    // Yes, email appears to be valid
}
```

Customizing Messages

Validate classes provide a `setMessage()` method with which you can specify the format of the message returned by `getMessages()` upon validation failure.

The first argument contains a string with the error message itself. Tokens can be incorporated within the string to customize the message with relevant data. The token `%value%` is supported by all validators, while other tokens are supported on a case-by-case basis, according to the validation class.

The second argument contains a string with the validation failure template to be set, used when a validation class defines more than one cause for failure.
If the second argument is missing, `setMessage()` assumes the message specified should be used for the first message template declared in the validation class. As many validation classes have only one error message template defined, there is no need to specify which template to change. Sample Code provided below:

```
<?php
require_once 'Zend/Validate/StringLength.php';

$validator = new Zend_Validate_StringLength(8);

$validator->setMessage(
    'The string \'%value%\' is too short;
        it must be at least %min% characters',
    Zend_Validate_StringLength::TOO_SHORT);

if (!$validator->isValid('word')) {
    $messages = $validator->getMessages();
    echo current($messages);

    // echoes "The string 'word' is too short;
        it must be at least 8 characters"
}
```

ZEND_VALIDATE

Standard Validation Classes

ZF offers a set of standard validation classes with the framework, as listed below. Information on each option can be found in the Programmers Guide – you should know how to utilize each.

- `Alnum`
- `Alpha`
- `Barcode`
- `Between`
- `Ccnum`
- `Date`
- `Digits`
- `EmailAddress`
- `Float`
- `GreaterThan`
- `Hex`
- `Hostname`
- `InArray`
- `Int`
- `Ip`
- `LessThan`
- `NotEmpty`
- `Regex`
- `StringLength`

Validation Chains

When using multiple validations, it will often be necessary to impose them in a specific order, known as 'chaining'. `Zend_Validate` provides a simple way to chain them together. Validators are run in the order in which they were added to `Zend_Validate` .
Ex: validating whether a username is between 6 and 12 alphanumeric characters.

```php
<?php
// Provides validator chaining capability
require_once 'Zend/Validate.php';

// Validators needed for the example
require_once 'Zend/Validate/StringLength.php';
require_once 'Zend/Validate/Alnum.php';

// Create a validator chain and add validators to it
$validatorChain = new Zend_Validate();
$validatorChain
    ->addValidator(new Zend_Validate_StringLength(6,12))
    ->addValidator(new Zend_Validate_Alnum());
// Validate the username
if ($validatorChain->isValid($username)) {
    // username passed validation
} else {
    // username failed validation; print reasons
    foreach ($validatorChain->getMessages() as $message) {
        echo "$message\n";
    }
}
```

ZEND_VALIDATE

Creating Custom Validators
Beyond the standard validators provided by ZF, `Zend_Validate_Interface` defines three methods, `isValid()`, `getMessages()`, and `getErrors()`, that can be implemented by user classes to create custom validation objects. Such objects can then be added to a validator chain using `Zend_Validate::addValidator()`, or with `Zend_Filter_Input`.

As mentioned earlier, validation results are returned as Boolean values, possibly along with information as to why the validation failed (useful for usability analysis). `Zend_Validate_Abstract` is used to implement the basic validation failure message functionality, which can be extended. The extending class would utilize the `isValid()` method logic and define message variables and message templates that correspond to the types of validation failures that can occur.

Generally, the `isValid()` method will not throw any exceptions unless it is impossible to determine whether or not the input value is valid (Ex: an LDAP server cannot be contacted, a file cannot be opened, a database unavailable, etc.)

TEST YOUR KNOWLEDGE : QUESTIONS

1

```
Which of the following could be used to validate an email
address (choose 2):

     a. Zend_Validate::is($email, 'EmailAddress');

     b. Zend_Validate::isValid($email, 'EmailAddress');

     c. $validator = new Zend_Validate_EmailAddress();
        if ($validator->isValid($email)) {
        // email appears to be valid
        }

     d. $validator = new
        Zend_Validate_EmailAddress($email);
        if ($validator->isValid()) {
        // email appears to be valid
        }
```

2

```
Which of the following can be used to produce:

   <a href="http://example.com">Click</a>

from the following input:

   <a href="http://example.com" onclick="doEvil()">Click</a>


     a.  Zend_Filter_StripTags

     b.  Zend_Text

     c.  Zend_Uri

     d.  Zend_View
```

Test Your Knowledge : Answers

★ = CORRECT

1

Which of the following could be used to validate an email address (choose 2):

★ a. `Zend_Validate::is($email, 'EmailAddress');`

b. `Zend_Validate::isValid($email, 'EmailAddress');`

★ c.
```
$validator = new Zend_Validate_EmailAddress();
   if ($validator->isValid($email)) {
   // email appears to be valid
   }
```

d.
```
$validator = new Zend_Validate_EmailAddress($email);
   if ($validator->isValid()) {
   // email appears to be valid
   }
```

2

Which of the following can be used to produce:

```
<a href="http://example.com">Click</a>
```

from the following input:

```
<a href="http://example.com" onclick="doEvil()">Click</a>
```

★ a. `Zend_Filter_StripTags`

b. `Zend_Text`

c. `Zend_Uri`

d. `Zend_View`

CERTIFICATION TOPIC : FORMS

Forms - Validation

Definitions •··• *Purpose / Use* •··• *Validator Chains*

Plugin Paths •··• *Error Messages*

Forms - Filtering

Definitions •··• *Purpose / Use* •··• *Plugin Paths*

Filtering

Forms - Decorators

Definitions •··• *Purpose / Use* •··• *Plugin Paths*

Forms - Elements

- *Definitions*
- *Purpose / Use*
- *Metadata*
- *Elements*
- *Options*

Forms – Display Groups, Sub-Forms

- *Definitions*
- *Purpose / Use*
- *Decorators*
- *Array Notation*

Forms – Configuration

- *Definitions*
- *Config Object*
- *Options*

Forms – Internationalization

- *Definitions*
- *Translate Object*

Zend Framework: Forms

For the exam, here's what you should know already …

You should know the basics of a form – what problems does it solve, what plugins does it use, what role does metadata play with forms, what methods are available and what do they control?

You should be able to explain the purpose of validators, and know how to utilize them (create, add and retrieve from elements, …).

You should be able to explain the role of filters, and know how to utilize them (create, chain, add and retrieve from elements, …).

You should know the purpose of decorators and how to manipulate them.

You should understand the relationship between a form and its elements.

You should know what a display group is, as well as a sub-form, and understand the difference between the two. You should know how they interact with decorators, and various use cases for each.

You should understand how to create a config object with form or element options, how to pass options, and how to pass a config object to a form or element.

Finally, you should know how to internationalize a form – what elements can be altered, how to connect a translate object to a form or element, and how to translate form error messages.

ZEND_FORM

Zend_Form simplifies form creation and handling in your web application. It accomplishes the following goals:

- Element input filtering and validation
- Element ordering
- Element and Form rendering, including escaping
- Element and form grouping
- Element and form-level configuration

It heavily leverages other Zend Framework components to accomplish its goals, including Zend_Config, _Validate, _Filter, _Loader_PluginLoader, and optionally _View

Create a Form Object

Instantiate Zend_Form to create a Form object:

```
<?php
$form = new Zend_Form;
```

It is a best practice to specify the form action and method; you can do this with the setAction() and setMethod() accessors.

```
<?php
$form->setAction('/resource/process')
     ->setMethod('post');
```

Add Elements to a Form

The 'pre-packaged' elements available with ZF are:

• button	• hidden	• image
• radio	• reset	• submit
• password	• text	• textarea
• checkbox / multiCheckbox		• select(regular, multiselect)

ZEND_FORM

Add Elements to a Form (continued)

There are several options for adding elements: (1) instantiate concrete elements, pass in these objects; (2) pass in the element type, have `Zend_Form` instantiate an object of the correct type; (3) use `createElement()` and then attach the element to the form using `addElement()`; (4) use configuration to specify elements for `Zend_Form` to create.

```
// Instantiating an element and passing to the form object:
$form->addElement(new Zend_Form_Element_Text('username'));

// Passing a form element type to the form object:
$form->addElement('text', 'username');
```

By default, these do not have any validators or filters. Therefore, you will need to configure your elements with validators, and potentially filters …

(a) before you pass the element to the form, or

(b) via configuration options passed in when creating an element via `Zend_Form`, or

(c) by pulling the element from the form object and configuring it after the fact.

Example: Creating Validators for a Concrete Element Instance

```
$username = new Zend_Form_Element_Text('username');

// Passing a Zend_Validate_* object:
$username->addValidator(new Zend_Validate_Alnum());

// Passing a validator name:
$username->addValidator('alnum');
```

In this example, you can either pass in `Zend_Validate_*` objects, or the name of a validator to use – in the latter case, the validator can accept constructor arguments, which can be passed in an array as the third parameter:

```
$username->addValidator('regex', false, array('/^[a-z]/i'));
```

Specify an Element as Required
This can be done using either an accessor or by passing an option when creating the element. When an element is required, a `'NotEmpty'` validator is added to the top of the validator chain, ensuring that the element has a value when required.

ZEND_FORM

Example: Creating Filters for a Concrete Element Instance
Filters are registered in basically the same way as validators. For illustration purposes, this code adds a filter to lowercase the final value.

```
$username->addValidator('alnum')
         ->addValidator('regex', false, array('/^[a-z]/'))
         ->setRequired(true)
         ->addFilter('StringToLower');
// or, more compactly:
$username->addValidators(
        array('alnum',
                array('regex', false, '/^[a-z]/i')
        ))
    ->setRequired(true)
    ->addFilters(array('StringToLower'));
```

Using a Factory for Adding Elements

When creating a new element using `Zend_Form::addElement()` as a factory, you can optionally pass in configuration options, including validators and filters to utilize.

```
$form->addElement('text', 'username', array(
    'validators' => array(
        'alnum',
        array('regex', false, '/^[a-z]/i')
    ),
    'required' => true,
    'filters'  => array('StringToLower'),
));
```

Render a Form

Most elements use a `Zend_View` helper to render themselves, thereby requiring a view object. Alternatively, you can either use the form's `render()` method, or simply echo it. By default, `Zend_Form` and `Zend_Form_Element` will attempt to use the view object initialized in the `ViewRenderer`, negating the need to set the view manually when using the Zend Framework MVC. However, if you are not using the MVC view renderer you will need to provide a `Zend_View` object to render the form elements. This is because `Zend_Form_Element` objects use the View helpers for rendering. Printing the form in a view script is then simple: `<?= $this->form ?>`

ZEND_FORM

Checking for Form Validity

When a form is assessed for validity, each element is checked against the data provided. If a key matching the element name is not present, and the item is marked as required, validations are run with a null value. The data sources are standard ($_GET, $_POST, Web Services, etc.).

```
if ($form->isValid($_POST)) {
    // success!
} else {
    // failure!
}
```

Partial Forms: Ajax requests may allow for checking on a single element or groups. `isValidPartial()` will validate a partial form but, in contrast to `isValid()`, if a particular key is not present, it will not run validations for that element.

Fetching Filtered and Unfiltered Values

Once the validations have been passed, filtered values can be retrieved with `getValues()`

```
$values = $form->getValues();
```

For unfiltered values:

```
$unfiltered = $form->getUnfilteredValues();
```

Retrieve Error Messages

Generally, when a form fails validation, it can simply be rendered again, and the errors will be displayed using the default decorators. To inspect the errors, either use the `getErrors()` method, which returns an associative array of element names/codes (= an array of error codes), or `getMessages(),` which returns an associative array of element names/messages (= an associative array of error code/error message pairs). If a given element does not have any errors, it will not be included in the array. `Zend_Form::getMessages()` returns an associative array of associative arrays; keys (element names) point to an associative array of code/message pairs.

FORMS – FILTERS (*also see the "Filtering and Validation" section of this guide*)

It is often useful and/or necessary to perform some kind of normalization on input prior to validation – for instance, to strip out all the HTML for security, but then run your validations on what remains to ensure the submission is valid; or, to trim empty space surrounding input so that a `StringLength` validator will not return a false positive.

These operations may be performed using `Zend_Filter`; `Zend_Form_Element` has support for filter chains, allowing you to specify multiple, sequential filters to utilize. Filtering happens both during validation and when you retrieve the element value via `getValue`().
Recall for unfiltered values to use: `$unfiltered` = `$form`->`getUnfilteredValues`();

```
<? php
$filtered = $element->getValue();
```

Filters can be added to the chain in two ways:

- passing in a concrete filter instance
- providing a filter name – either a short name or a fully qualified class name

```
<?php
// Concrete filter instance:
$element->addFilter(new Zend_Filter_Alnum());
// Fully qualified class name:
$element->addFilter('Zend_Filter_Alnum');
// Short filter name:
$element->addFilter('Alnum');
$element->addFilter('alnum');
```

Short names are typically the filter name minus the prefix. In the default case, this will mean minus the `'Zend_Filter_'` prefix. Additionally, the first letter need not be in upper-case.

Custom Filter Classes

Use `addPrefixPath`() with `Zend_Form_Element` to utilize custom filter classes. Example:

```
<?php
$element->addPrefixPath('My_Filter', 'My/Filter/', 'filter');
```

FORMS – VALIDATORS *(also see the "Filtering and Validation" section of this guide)*

Validators examine input against a set of requirements and produce a boolean result (True or False) as to whether the input passed. The process can also provide further information about which requirement(s) the input either met or failed. This is especially important in a security context ("filter input, escape output").

In `Zend_Form`, each element includes its own validator chain, consisting of `Zend_Validate_*` validators.

Validators may be added to the chain in two ways:

- passing in a concrete validator instance
- providing a validator name – either a short name or a fully qualified class name

```
// Concrete validator instance:
$element->addValidator(new Zend_Validate_Alnum());

// Fully qualified class name:
$element->addValidator('Zend_Validate_Alnum');

// Short validator name:
$element->addValidator('Alnum');
$element->addValidator('alnum');
```

Short names are typically the validator name minus the prefix. In the default case, this will mean minus the `'Zend_Validate_'` prefix. The first letter need not be in upper-case.

Custom Validator Classes

Use `addPrefixPath`() with `Zend_Form_Element` to utilize custom validator classes:

```
$element->addPrefixPath('My_Validator, 'My/Validator/',
                        'Validator');
```

Recall that the third argument indicates the plugin loader on which to perform the action. If a particular validation failure should prevent later validators from firing, then pass the boolean TRUE as the second parameter: `$element->addValidator('alnum', true);`

FORMS - VALIDATION

Custom Validator Messages

Some developers may wish to provide custom error messages for a validator. The `Zend_Form_Element::addValidator()$options` argument allows you to do so by providing the key 'messages' and setting it to an array of key/value pairs for the message templates. You will need to know the error codes of the various validation error types for the particular validator.

A better option is to use a `Zend_Translate_Adapter` with your form. Error codes are automatically passed to the adapter by the default errors decorator - you can then specify your own error message strings by setting up translations for the various error codes of your validators.

You can also set multiple validators at once, using `addValidators()`. The basic way to utilize this method is to pass an array of arrays, with each array containing 1 to 3 values, that matches the `addValidator()`constructor`.`

```
$element->addValidators(array(
    array('NotEmpty', true),
    array('alnum'),
    array('stringLength', false, array(6, 20)),
));
```

Alternatively, you can use the array keys `'validator'`, `'breakChainOnFailure'`, and `'options'`:

```
$element->addValidators(array(
    array(
        'validator'           => 'NotEmpty',
        'breakChainOnFailure' => true),
    array('validator' => 'alnum'),
    array(
        'validator' => 'stringLength',
        'options'   => array(6, 20)),
));
```

Note the use of '`breakChainOnFailure`' in the second argument. When set to '`TRUE`', any later validations in the chain are skipped.

FORMS - VALIDATION

Configure Validators

The previous example provided in the Custom Validator Error Messages section provides the scenario for showing how to configure validators in a config file:

```
element.validators.notempty.validator = "NotEmpty"
element.validators.notempty.breakChainOnFailure = true
element.validators.alnum.validator = "Alnum"
element.validators.strlen.validator = "StringLength"
element.validators.strlen.options.min = 6
element.validators.strlen.options.max = 20
```

Note that every item has a key, whether or not it needs one. This is a limitation of using configuration files, but it helps to clarify the purpose of each argument. Remember that any validator options must be specified in the correct order.

To validate an element, pass the value to `isValid()`:

```
if ($element->isValid($value)) {
    // valid
} else {
    // invalid
}
```

Recall: `Zend_Form_Element::isValid()` filters values through the provided filter chain prior to validation. It can take an optional second argument, `$context`. Usually, `Zend_Form` passes in the entire array of values being validated, which allows you to write validators that compare a value against other submitted values.

FORMS - VALIDATION

Zend_Form Elements as General-Purpose Validators

`Zend_Form_Element` implements `Zend_Validate_Interface,` illustrating how an element may also be used as a validator in other, non-form related validation chains. Methods include:

- `setRequired($flag) and isRequired()`

  allow you to set and retrieve the status of the 'required' flag. When set to boolean `true`, this flag requires that the element be in the data processed by `Zend_Form`.

- `setAllowEmpty() and getAllowEmpty()`

  allow you to modify the behavior of optional elements (i.e., elements where the required flag is false). When the 'allow empty' flag is true, empty values will not be passed to the validator chain.

- `setAutoInsertNotEmptyValidator($flag)`

  allows you to specify whether or not a `'NotEmpty'` validator will be prepended to the validator chain when the element is required. By default, this flag is true.

- `addValidator($nameOrValidator, $breakChainOnFailure = false, array $options = null)`

- `addValidators(array $validators)`

- `setValidators(array $validators)`

  overwrites all validators

- `getValidator($name)`

  retrieves a validator object by name

- `getValidators()`

  retrieves all validators

- `removeValidator($name)`

  removes validator by name

- `clearValidators()`

  removes all validators

ZEND_FORM

Plugin Loaders

`Zend_Form_Element` makes use of `Zend_Loader_PluginLoader` to specify the locations of alternate validators, filters, and decorators. Each has its own plugin loader associated with it, and general accessors are used to retrieve and modify each. The following loader types are used with the various plugin loader methods (the type names are case insensitive):

validate   filter   decorator

The methods used to interact with plugin loaders are as follows:

- `setPluginLoader($loader, $type):`

  `$loader` is the plugin loader object itself, while `$type` is one of the types. This sets the plugin loader for the given type to the newly specified loader object.

- `getPluginLoader($type):`

  retrieves the plugin loader associated with `$type`.

- `addPrefixPath($prefix, $path, $type = null):`

  adds a prefix/path association to the loader specified by `$type`. If `$type` is null, it will attempt to add the path to all loaders, by appending the prefix with each of "`_Validate`", "`_Filter`", and "`_Decorator`"; and appending the path with "`Validate/`", "`Filter/`", and "`Decorator/`". Having all the extra form element classes under a common hierarchy is a convenient method for setting the base prefix for them.

- `addPrefixPaths(array $spec):`

  allows the addition of many paths at once to one or more plugin loaders. It expects each array item to be an array with the keys 'path', 'prefix', and 'type'.

Custom validators, filters, and decorators are an easy way to share functionality among forms and encapsulate custom functionality. The online Reference Guide provides an extensive example for creating a custom decorator.

ZEND_FORM

Decorators

Zend_Form_Element uses "decorators", which are simply classes that have access to both the element and a method for rendering content. These decorators can replace content, append content, or prepend content, and have full introspection to the element passed to them. As a result, multiple decorators can be combined to achieve custom effects. By default, Zend_Form_Element actually combines four decorators to achieve its output. Example:

- ViewHelper: specifies a view helper to use for rendering the element. The 'helper' element attribute can be used to specify which view helper to use. By default, Zend_Form_Element specifies the 'formText' view helper, but individual subclasses specify different helpers.

- Errors: appends error messages to the element using Zend_View_Helper_FormErrors. If no errors are present, nothing is appended.

- HtmlTag: wraps the element and errors in an HTML <dd> tag.

- Label: prepends a label to the element using Zend_View_Helper_FormLabel, and wraps it in a <dt> tag. If no label is provided, just the definition term tag is rendered.

By default, the default decorators are loaded during object initialization. You can disable this by passing the 'disableLoadDefaultDecorators' option to the constructor:

```
<?php
$element = new Zend_Form_Element(
    'foo',
    array('disableLoadDefaultDecorators' => true)
);
```

This option may be mixed with any other options you pass, both as array options or in a Zend_Config object. The initial content is created by the 'ViewHelper' decorator, which creates the form element itself. Next, the 'Errors' decorator fetches error messages from the element, and, if any are present, passes them to the 'FormErrors' view helper to render. The next decorator, 'HtmlTag', wraps the element and errors in an HTML <dd> tag. Finally, the last decorator, 'label', retrieves the element's label and passes it to the 'FormLabel' view helper, wrapping it in an HTML <dt> tag; the value is prepended to the content by default.

ZEND_FORM

Multiple Decorators

Zend_Form_Element uses a decorator's class as the lookup mechanism when retrieving decorators. As a result, you cannot register multiple decorators of the same type; subsequent decorators will simply overwrite those that existed before. To get around this, you can use aliases. Instead of passing a decorator or decorator name as the first argument to addDecorator(), pass an array with a single element, with the alias pointing to the decorator object or name:

```
<?php
// Alias to 'FooBar':
$element->
    addDecorator(
       array('FooBar' => 'HtmlTag'),
       array('tag' => 'div')
    );

// And retrieve later:
$decorator = $element->getDecorator('FooBar');
```

In the addDecorators() and setDecorators() methods, you can either pass the 'decorator' option in the array representing the decorator, or pass it as the first element in the decorator portion of the array:

```
<?php
// Add two 'HtmlTag' decorators, aliasing one to 'FooBar':
$element->addDecorators(
     array(
                array('HtmlTag', array('tag' => 'div')),
                array(
                 'decorator' => array('FooBar' => 'HtmlTag'),
                   'options' => array('tag' => 'dd')
                  )
)
);

// And retrieve later:
$htmlTag = $element->getDecorator('HtmlTag');
$fooBar  = $element->getDecorator('FooBar');
```

ZEND_FORM

Configuration

The `Zend_Form_Element` constructor accepts either an array of options or a `Zend_Config` object containing options, and it can also be configured using either `setOptions()` or `setConfig()`. Generally speaking, keys are named as follows:

- If the 'set' + key refers to a `Zend_Form_Element` method, then the value provided will be passed to that method.
- Otherwise, the value will be used to set an attribute
- The first letter of a config key can be either lower or upper case, but the remainder of the key should follow the same case as the accessor method

Exceptions to the rule include:

- `prefixPath` will be passed to `addPrefixPaths()`
- The following setters cannot be set in this way:
  - `setAttrib` (although `setAttribs` *will* work)
  - `setConfig`
  - `setOptions`
  - `setPluginLoader`
  - `setTranslator`
  - `setView`

Display Groups

Display groups are a way to create virtual groupings of elements for display purposes. All elements remain accessible by name in the form, but when iterating over the form or rendering, any elements in a display group are rendered together. The most common use case for this is for grouping elements in fieldsets.

The base class for display groups is `Zend_Form_DisplayGroup`. While it can be instantiated directly, it is typically best to use the `Zend_Form` `addDisplayGroup()` method to do so. This method takes an array of elements as its first argument, and a name for the display group as its second argument. You may optionally pass in an array of options or a `Zend_Config` object as the third argument. Assuming that the elements 'username' and 'password' are already set in the form, this code would group these elements in a 'login' display group:

```
$form->addDisplayGroup(array('username', 'password'), 'login');
```

ZEND_FORM

Sub Forms

Sub forms serve several purposes:

- Creating logical element groups. Since sub forms are simply forms, you can validate subforms as individual entities.
- Creating multi-page forms. Since sub forms are simply forms, you can display a separate sub form per page, building up multi-page forms where each form has its own validation logic. Only once all sub forms validate would the form be considered complete.
- Display groupings. Like display groups, sub forms, when rendered as part of a larger form, can be used to group elements. Be aware, however, that the master form object will have no awareness of the elements in sub forms.

```
<?php
$form->addSubForm($subForm, 'subform');
```

You can retrieve a sub form using either `getSubForm($name)` or overloading using the sub form name.

```
<?php
// Using getSubForm():
$subForm = $form->getSubForm('subform');

// Using overloading:
$subForm = $form->subform;
```

ZEND_FORM

Internationalization of Forms

By default, no internationalization (I18n) is performed. To turn on I18n features in `Zend_Form`, you will need to instantiate a `Zend_Translate` object with an appropriate adapter, and attach it to `Zend_Form` and/or `Zend_Validate`.

Add to the Registry - this will be picked up by `Zend_Form`, `Zend_Validate`, and `Zend_View_Helper_Translate`.

```
<?php
// use the 'Zend_Translate' key; $translate is a
// Zend_Translate object:
Zend_Registry::set('Zend_Translate', $translate);
```

Error Messages: register the translation object with `Zend_Validate_Abstract`

```
<?php
//Tell all validation classes to use a specific translate adapter
Zend_Validate_Abstract::setDefaultTranslator($translate);
```

or attach to the `Zend_Form` object as a global translator. This has the side effect of also translating validation error messages.

```
<?php
//Tell all form classes to use a specific translate adapter, as
//well as use this adapter to translate validation error messages
Zend_Form::setDefaultTranslator($translate);
```

Permissible Translation Targets:

- *Validation error messages*. Use the various error code constants from `Zend_Validate` validation classes as the message IDs.
- *Labels*. Element labels will be translated, if a translation exists.
- *Fieldset Legends*. Display groups and sub forms rendered in fieldsets by default. The Fieldset decorator attempts to translate the legend before rendering the fieldset.
- *Form and Element Descriptions*. All form types (element, form, display group, sub form) allow specifying an optional item description - the Description decorator can render this, and by default will take the value and attempt to translate it.
- *Multi-option Values*. For items inheriting from `Zend_Form_Element_Multi` (MultiCheckbox, Multiselect, Radio elements), the option values (not keys) will be translated if one is available; option labels presented to the user will be translated.
- *Submit and Button Labels*. The various Submit and Button elements (Button, Submit, and Reset) will translate the label displayed to the user.

TEST YOUR KNOWLEDGE : QUESTIONS

1

Validators used with Zend_Form should implement which component?

a. Zend_Form_Validator_Abstract

b. Zend_Validate_Abstract

c. Zend_Validate_Interface

d. Zend_Form_Validate_Interface

2

Which of the following is NOT a valid mechanism for adding a decorator to an element?

a.
```
<code>
$element->addDecorator(new
Zend_Form_Decorator_ViewHelper());
</code>
```

b.
```
<code>
$element->attachDecorator('ViewHelper');
</code>
```

c.
```
<code>
$element->addDecorator('ViewHelper');
</code>
```

d.
```
<code>
$element->addDecorator('ViewHelper',
                       array('helper' => 'foo'));
</code>
```

TEST YOUR KNOWLEDGE : ANSWERS

★ = CORRECT

1

Validators used with Zend_Form should implement which class?

a. Zend_Form_Validator_Abstract

b. Zend_Validate_Abstract

★ c. Zend_Validate_Interface

d. Zend_Form_Validate_Interface

2

Which of the following is NOT a valid mechanism for adding a decorator to an element?

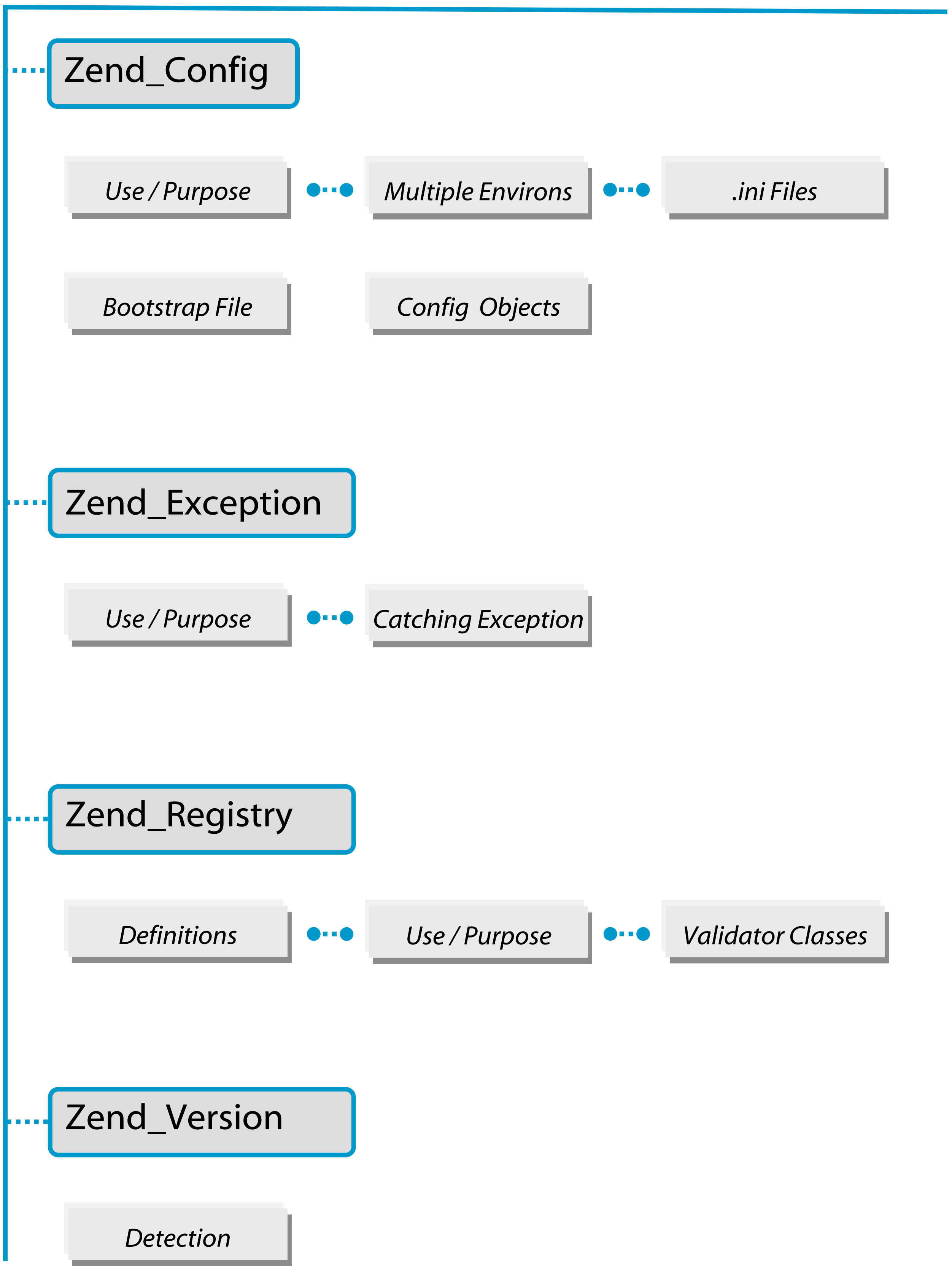
a.
```
<code>
$element->addDecorator(new
Zend_Form_Decorator_ViewHelper());
</code>
```

★ b.
```
<code>
$element->attachDecorator('ViewHelper');
</code>
```

c.
```
<code>
$element->addDecorator('ViewHelper');
</code>
```

d.
```
<code>
$element->addDecorator('ViewHelper',
                       array('helper' => 'foo'));
</code>
```

CERTIFICATION TOPIC : INFRASTRUCTURE



Zend_Loader

Use / Purpose

Autoloading

Conventions

Plugins

Zend_Session

Definitions

Use / Purpose

Bootstrap

Zend Framework: Infrastructure

For the exam, here's what you should know already …

CONFIGURATION
You should be able to explain the purpose of configuration for single and multiple environments within an MVC-based application.

You should be able to define a configuration object using an `ini` file, and then interact with and utilize that configuration object; you should know how to set up an application bootstrap file to utilize a configuration file.

EXCEPTIONS
You should know the purpose of framework-specific exceptions and what they represent; you should also be able to create code that catches exceptions thrown by any ZF component.

REGISTRY
You should know the purpose of having a registry within an object-oriented based application.

You should be able to assign application-wide objects to the global registry, as well as know when and how to create an alternative to the global registry.

LOADER
You should know when to use, and when not to use, `Zend_Loader` within an application and libraries, know how to utilize the autoload feature, know how `Zend_Loader` maps class names to files.

You should be able to explain the purpose of `Zend_Loader_PluginLoader` as well as utilize it within an application.

SESSION
You should understand the features and benefits that `Zend_Session` provides, how to set an application bootstrap file to use `Zend_Session`, and how to persist data between requests.

ZEND_CONFIG

Zend_Config is designed to simplify access to, and use of, configuration data within applications. It provides a nested object property-based user interface for accessing configuration data within application code.

The configuration data may come from a variety of media-supporting hierarchical data storage. Currently Zend_Config provides adapters for configuration data that are stored in text files with Zend_Config_Ini and Zend_Config_Xml (two most commonly used).

Configuration Data in PHP Array

If the config data is available in a PHP array, simply pass the data to the Zend_Config constructor to utilize a simple, object-oriented interface. Zend_Config also has get(), which will return the supplied default value if the data element doesn't exist.

```
<?php
// Given an array of configuration data
$configArray = array(
    'webhost'  => 'www.example.com',
    'database' => array(
        'adapter' => 'pdo_mysql',
        'params'  => array(
            'host'     => 'db.example.com',
            'username' => 'dbuser',
            'password' => 'secret',
            'dbname'   => 'mydatabase'
        )
    )
);

// Create the object-oriented wrapper upon the configuration data
require_once 'Zend/Config.php';
$config = new Zend_Config($configArray);

// Print a configuration datum (results in 'www.example.com')
echo $config->webhost;

// Use the configuration data to connect to the database
$db = Zend_Db::factory($config->database->adapter,
                       $config->database->params->toArray());

// Alternative usage: simply pass the Zend_Config object.
// The Zend_Db factory knows how to interpret it.
$db = Zend_Db::factory($config->database);
```

ZEND_CONFIG

Using Zend_Config with a PHP Configuration File:
It is often desirable to use a purely PHP-based configuration file. Example:

```
<?php
// config.php
return array(
    'webhost'  => 'www.example.com',
    'database' => array(
        'adapter' => 'pdo_mysql',
        'params'  => array(
            'host'     => 'db.example.com',
            'username' => 'dbuser',
            'password' => 'secret',
            'dbname'   => 'mydatabase'
        `
```

```
<?php
// Configuration consumption
require_once 'Zend/Config.php';
$config = new Zend_Config(require 'config.php');

// Print a configuration datum (results in 'www.example.com')
echo $config->webhost;
```

How Configuration Works:
Configuration data are made accessible to the `Zend_Config` constructor through an associative array, which may be multidimensional, in order to support organizing the data from general to specific. Concrete adapter classes are used in the process. User scripts may provide such arrays directly to the `Zend_Config` constructor, without using an adapter class, whenever it is more appropriate to do so.

ZEND_CONFIG

How Configuration Works (continued)

Each configuration data array value becomes a property of the Zend_Config object, with the key used as the property name. If a value is itself an array, then the resulting object property created is a new Zend_Config object, loaded with the array data. This occurs recursively, such that a hierarchy of configuration data may be created with any number of levels.

Zend_Config implements the Countable and Iterator interfaces in order to facilitate simple access to configuration data, allowing the use of the count() function and PHP constructs such as foreach upon Zend_Config objects.

Adapter classes inherit from the Zend_Config class since they utilize its functionality.
The Zend_Config family of classes enables configuration data to be organized into sections.
Zend_Config adapter objects may be loaded with a single specified section, multiple specified sections, or all sections (if none are specified).

Zend_Config adapter classes support a single inheritance model that enables configuration data to be inherited from one section of configuration data into another, reducing or eliminating the need for duplicating configuration data for different purposes. An inheriting section can also override the values that it inherits through its parent section.
Two Zend_Config objects can be merged into a single object using the merge() function.

ZEND_CONFIG

Zend_Config_Ini

`Zend_Config_Ini` enables developers to store configuration data in a familiar INI format and read them in the application using nested object property syntax. The INI format is specialized to provide both the ability to have a hierarchy of configuration data keys and inheritance between configuration data sections.

Configuration data hierarchies are supported by separating the keys with a period character (.)

A section may extend or inherit from another section by following the section name with a colon character (:) and the name of the section from which to inherit the data.

The configuration data may represent multiple environments (Ex: production and staging). The developer would simply specify which environment to engage by specifying the INI file and the specified environment's section (path). In this example, the configuration data for the staging environment is being used. Given that the configuration data is contained in `/path/to/config.ini`:

```
; Production site configuration data
[production]
webhost                  = www.example.com
database.adapter         = pdo_mysql
database.params.host     = db.example.com
database.params.username = dbuser
database.params.password = secret
database.params.dbname   = dbname

; Staging site configuration data inherits from production and
; overrides values as necessary
[staging : production]
database.params.host     = dev.example.com
database.params.username = devuser
database.params.password = devsecret
```

```
<?php
$config = new Zend_Config_Ini('/path/to/config.ini', 'staging');

echo $config->database->params->host; // prints "dev.example.com"
echo $config->database->params->dbname; // prints "dbname"
```

ZEND_CONFIG

Zend_Config_Xml

`Zend_Config_Xml` enables developers to store configuration data in a simple XML format and read them via nested object property syntax. Configuration data read into `Zend_Config_Xml` are always returned as strings.

The root element of the XML file is irrelevant and the first level corresponds with configuration data sections. The XML format supports hierarchical organization through nesting of XML elements below the section-level elements. The content of a leaf-level XML element corresponds to the value of a configuration datum. Section inheritance is supported by a special XML attribute named `extends`, and the value of this attribute corresponds with the section from which data are to be inherited by the extending section.

Example: using `Zend_Config_Xml` for loading configuration data from an XML file with multiple environments (staging and production). Given that the configuration data is contained in `/path/to/config.xml`, and the application needs the staging configuration data:

```
<?xml version="1.0"?>
<configdata>
    <production>
        <webhost>www.example.com</webhost>
        <database>
            <adapter>pdo_mysql</adapter>
            <params>
                <host>db.example.com</host>
                <username>dbuser</username>
                <password>secret</password>
                <dbname>dbname</dbname>
            </params>
        </database>
    </production>
    <staging extends="production">
        <database>
            <params>
                <host>dev.example.com</host>
                <username>devuser</username>
                <password>devsecret</password>
            </params>
        </database>
    </staging>
</configdata>

$config = new Zend_Config_Xml('/path/to/config.xml', 'staging');

echo $config->database->params->host; // prints "dev.example.com"
echo $config->database->params->dbname; // prints "dbname"
```

ZEND_EXCEPTION

All exceptions thrown by Zend Framework classes should derive from the base class `Zend_Exception`.

Catching Exceptions

The following example illustrates code for catching exceptions within a ZF application:

```
<?php

try {
    Zend_Loader::loadClass('nonexistantclass');
} catch (Zend_Exception $e) {
    echo "Caught exception: " . get_class($e) . "\n";
    echo "Message: " . $e->getMessage() . "\n";
    // other code to recover from the failure.
}
```

ZEND_VERSION

Reading the Zend Framework Version

The class constant `Zend_Version::VERSION` contains a string that identifies the current version number of Zend Framework.

The static method `Zend_Version::compareVersion($version)`is based on the PHP function `version_compare()`. The method returns "`-1`" if the specified `$version` is older than the Zend Framework version, "`0`" if they are the same, and "`+1`" if the specified `$version` is newer than the Zend Framework version.

```
<?php
// returns -1, 0 or 1
$cmp = Zend_Version::compareVersion('1.0.0');
```

ZEND_REGISTRY

The registry is a container for storing objects and values in the application space. By storing the value in the registry, the same object is always available throughout your application. This mechanism is an alternative to using global storage.

The typical usage of the registry is through static methods in the `Zend_Registry` class. Alternatively, the class is an array object, so you can access elements stored within it with a convenient array-like interface

Constructing a Registry Object

In addition to accessing the static registry through static methods, you can create an instance directly and use it as an object. The registry instance you access through the static methods is simply one such instance, and it is for convenience that it is stored statically, so you can access it from anywhere in your application.

Use a traditional `new` constructor to create an instance of the registry. This gives you the opportunity to initialize the entries in the registry as an associative array.

Example: Constructing a Registry

```php
<?php
$registry = new Zend_Registry(array('index' => $value));
```

After constructing this instance, you can use it using array-object methods, or you can set this instance to become the static instance using `setInstance()`.

Example: Initializing the Static Registry

```php
<?php
$registry = new Zend_Registry(array('index' => $value));

Zend_Registry::setInstance($registry);
```

The `setInstance()` method throws a `Zend_Exception` if the static registry has already been initialized by its first access.

Example: Catching an Exception

```php
<?php
try {
    Zend_Loader::loadClass('nonexistantclass');
} catch (Zend_Exception $e) {
    echo "Caught exception: " . get_class($e) . "\n";
    echo "Message: " . $e->getMessage() . "\n";
    // other code to recover from the failure.
}
```

ZEND_LOADER

The Zend_Loader class includes methods to help you load files dynamically.

Loading Files

The static method `Zend_Loader::loadFile()` loads a PHP file, which may contain any PHP code. The method is a wrapper for the PHP function `include()`and throws `Zend_Exception` on failure.

The `$filename` argument specifies the filename to load – it can only contain alphanumeric characters, hyphens ("-"), underscores ("_"), or periods ("."), and must *not* contain any path information. A security check is run on this.

No similar restrictions are placed on the `$dirs` argument, which specifies the directories to search for the file. If it returns `NULL`, only the `include_path` is searched. If it returns a string or array, the directory or directories specified will be searched, and then the `include_path`.

The `$once` argument is a boolean. If TRUE, `Zend_Loader::loadFile()` uses the PHP function `include_once()` for loading the file, otherwise the PHP function `include()` is used.

Loading Classes

The static method `Zend_Loader::loadClass($class, $dirs)` loads a PHP file and then checks for the existence of the class. The string specifying the class is converted to a relative path by substituting directory separates for underscores, and appending `'.php'`. Ex: `'Container_Tree'` would become `'Container/Tree.php'`.

```
<?php
Zend_Loader::loadClass('Container_Tree',
    array(
        '/home/production/mylib',
        '/home/production/myapp'
    )
);
```

If `$dirs` is a string or an array, `Zend_Loader::loadClass()` searches the directories in the order supplied, loading the first matching file. If the file does not exist in the specified `$dirs`, then the `include_path` is searched. If the file is not found or the class does not exist after the load, a `Zend_Exception` is thrown.

When `Zend_Loader::loadClass()` is used, the class name can contain only alphanumeric characters and hyphens ("-"), underscores ("_"), or periods (".")

ZEND_LOADER

Plugin Loader

A number of Zend Framework components are 'pluggable', and allow loading of dynamic functionality by specifying a class prefix and path to class files that are not necessarily on the `include_path`, or do not necessarily follow traditional naming conventions. `Zend_Loader_PluginLoader` provides common functionality for this.

The basic usage of the `PluginLoader` follows Zend Framework naming conventions - one class per file, using the underscore as a directory separator when resolving paths. It allows passing an optional class prefix to prepend when determining if a particular plugin class is loaded, and paths are searched in LIFO order ("Last In, First Out"). This allows for namespacing plugins, and thus overriding plugins from paths registered earlier.

`Zend_Loader_PluginLoader` also optionally allows plugins to be shared across plugin-aware objects, without needing to utilize a singleton instance, via a static registry. Simply indicate the registry name at instantiation as the second parameter to the constructor. Other components that instantiate the `PluginLoader` using the same registry name will then have access to already loaded paths and plugins.

Example: given the following directory structure, the sample code follows:

```
application/
    modules/
        foo/
            views/
                helpers/
                    FormLabel.php
                    FormSubmit.php
        bar/
            views/
                helpers/
                    FormSubmit.php
library/
    Zend/
        View/
            Helper/
                FormLabel.php
                FormSubmit.php
                FormText.php
```

```
<?php
$loader = new Zend_Loader_PluginLoader();
$loader->addPrefixPath('Zend_View_Helper', 'Zend/View/Helper/')
       ->addPrefixPath('Foo_View_Helper',
         'application/modules/foo/views/helpers')
       ->addPrefixPath('Bar_View_Helper',
         'application/modules/bar/views/helpers');
```

ZEND_SESSION

For PHP applications, a session represents a logical, one-to-one connection between server-side, persistent state data and a particular user agent client (e.g., a web browser). `Zend_Session` helps manage and preserve session data, a logical complement of cookie data, across multiple page requests by the same client.

Session Data

Unlike cookie data, session data are not stored on the client side and are only shared with the client when server-side source code voluntarily makes the data available in response to a client request. (Here, the term "session data" refers to the server-side data stored in `$_SESSION`, managed by `Zend_Session`, and individually manipulated by `Zend_Session_Namespace` accessor objects.)

Zend_Session_Namespace

Session namespaces provide access to session data using classic namespaces implemented logically as named groups of associative arrays, keyed by strings (similar to normal PHP arrays).

`Zend_Session_Namespace` instances are accessor objects for namespaced slices of `$_SESSION`. The `Zend_Session` component wraps the existing PHP ext/session with an administration and management interface, as well as providing an API for `Zend_Session_Namespace` to persist session namespaces. `Zend_Session_Namespace` provides a standardized, object-oriented interface for working with namespaces persisted inside PHP's standard session mechanism.

Support exists for both anonymous and authenticated (e.g., "login") session namespaces. `Zend_Auth`, the authentication component of ZF, uses `Zend_Session_Namespace` to store some information associated with authenticated users.

Since `Zend_Session` uses the normal PHP extension/session functions internally, all the familiar configuration options and settings apply, with the convenience of an object-oriented interface and default behavior that provides both best practices and smooth integration with the Zend Framework. Thus, a standard PHP session identifier, whether conveyed by cookie or within URLs, maintains the association between a client and session state data.

Operation

When the first session namespace is requested, `Zend_Session` will automatically start the PHP session, unless already started with `Zend_Session::start()`. The PHP session will use defaults from `Zend_Session`, unless modified by `Zend_Session::setOptions()`.

ZEND_SESSION

Operation (continued)

To set a session configuration option, include the basename as a key of an array passed to `Zend_Session::setOptions()`. The corresponding value in the array is used to set the session option value. If no options are set, `Zend_Session` will utilize recommended default options first, then default php.ini settings.

Bootstrap File

If you want all requests to have a session facilitated by `Zend_Session`, then start the session in the bootstrap file. This also prevents the session from starting after headers have been sent to the browser, which results in an exception, and possibly a broken page for website viewers.
Example: Starting the Global Session

```
<?php
require_once 'Zend/Session.php';

Zend_Session::start();
```

For more information about the right, and wrong, way to start a session, see the Programmers Guide.

Persist Data and Objects

Limits can be placed on the longevity of both namespaces and individual keys in namespaces. Expiration can be based on either elapsed seconds or the number of "hops", where a hop occurs for each successive request that instantiates the namespace at least once. When working with data expiring from the session in the current request, care should be used when retrieving them. Although the data is returned by reference, modifying the data will not make expiring data persist past the current request. In order to "reset" the expiration time, fetch the data into temporary variables, use the namespace to unset them, and then set the appropriate keys again.

Objects persisted in the PHP session are serialized for storage, and so must be unserialized upon retrieval – but – it is important to realize that the classes for the persisted objects must have been defined before the object is unserialized. If objects are unserialized to an unknown class, they become `__PHP_Incomplete_Class_Name` objects.

TEST YOUR KNOWLEDGE : QUESTIONS

1

Zend_Config allows hierarchical and sectioned key-value pairs to exist in a single file.

a. True

b. False

2

Given `$tree = new Tree(array('type' => 'cedar'));` , and you wish to persist this object via Zend_Session, which call will put this object in the default namespace?

a.
```
Zend_Session_Namespace::set('tree', $tree);
```

b.
```
$sess = new Zend_Session_Namespace();
$sess->tree = $tree;
```

c.
```
$sess = new Zend_Session();
$sess->tree = $tree;
```

d.
```
$sess = Zend_Session::getInstance()
$sess->set('tree', $tree);
```

TEST YOUR KNOWLEDGE : ANSWERS

★ = CORRECT

1

Zend_Config allows hierarchical and sectioned key-value pairs to exist in a single file.

★ a. True

b. False

2

Given $tree = new Tree(array('type' => 'cedar')); , and you wish to persist this object via Zend_Session, which call will put this object in the default namespace?

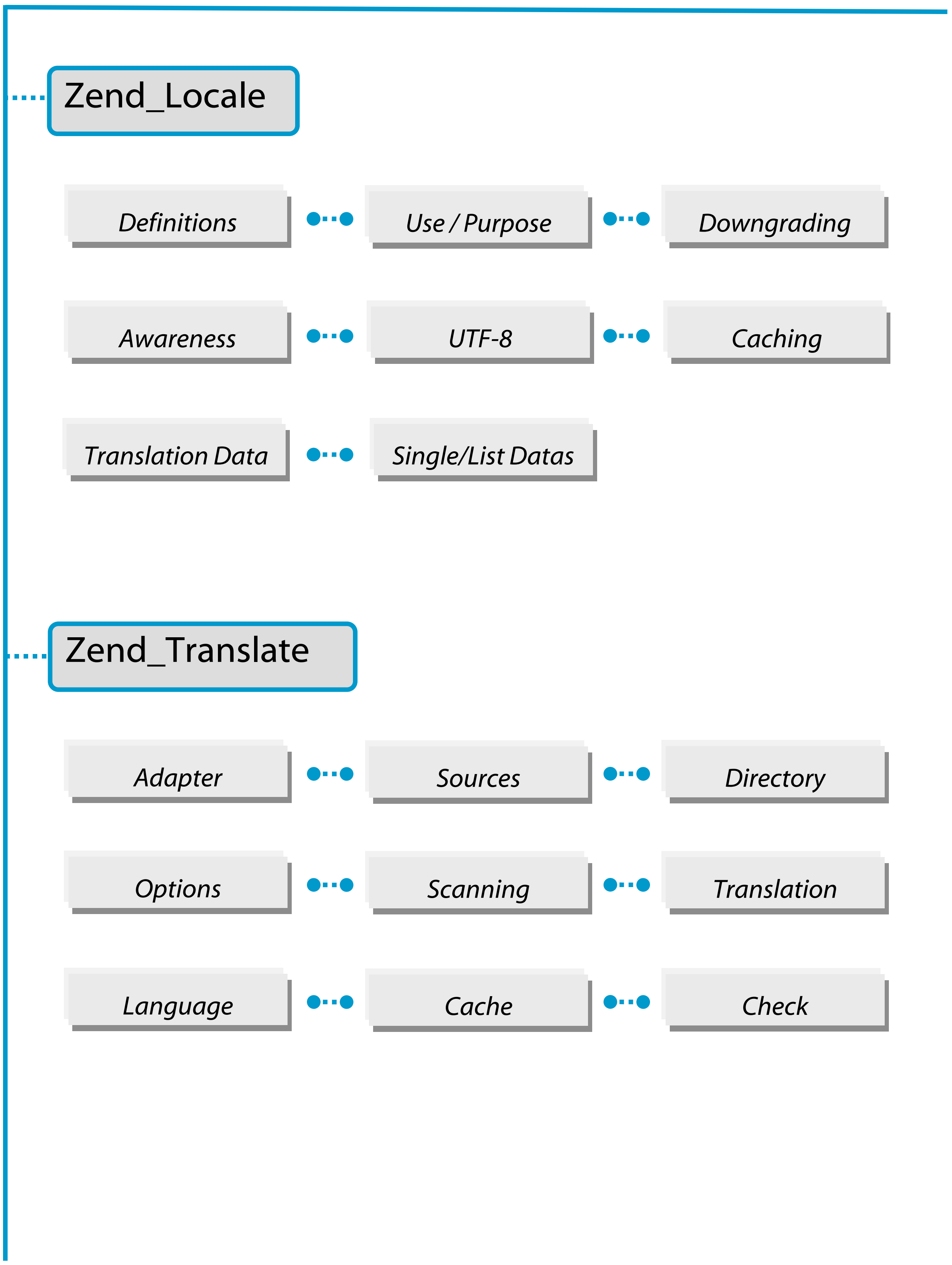
a. `Zend_Session_Namespace::set('tree', $tree);`

★ b.
```
$sess = new Zend_Session_Namespace();
$sess->tree = $tree;
```

c.
```
$sess = new Zend_Session();
$sess->tree = $tree;
```

d.
```
$sess = Zend_Session::getInstance()
$sess->set('tree', $tree);
```

CERTIFICATION TOPIC : INTERNATIONALIZATION



Zend_Date

Definitions · *Options* · *Formats*

Timezone · *API* · *Localization*

Zend_Currency

Creation · *Output*

Zend_View_Helper_Trans

Adapter · *Set* · *Parameters*

Fluidity · *Standalone*

Zend Framework: Internationalization

For the exam, here's what you should know already …

You should be able to demonstrate a thorough knowledge of locales – what they are, where they are stored, how to create one, downgrading & checking, etc.

In addition, you should know how the translation process works, how to select and use adapters, access different sources, change languages, etc.

Also, you should be able to work with both dates and with currency, and their corresponding options and functions.

ZEND_LOCALE

Internationalization of a web site requires two basic processes: Localization (L10N) and Internationalization (I18N). Many considerations go into adjusting web sites for regions, as revealed by the various needed ZF components. For more detail, see the Programmers Guide.

Localization:
Within Zend Framework, the following components are used in the localization process – the component name reveals its role in most cases:

- `Zend_Locale`
- `Zend_Translate`
- `Zend_Date` *includes date and time*
- `Zend_Currency`
- `Zend_Locale_Format`
- `Zend_Locale_Data`

Zend Locale
A locale string or object provides `Zend_Locale` and its subclasses access to information about the language and region expected by the user. Correct formatting, normalization, and conversions are made based on this information. Locale identifier strings used in Zend Framework are internationally defined standard abbreviations of language and region, written as `language_REGION`. Both parts are abbreviated as alphabetic, ASCII characters. `Zend_Locale`, unlike PHP's `setlocale`(), is thread-safe.

Generally, new `Zend_Locale`() will automatically select the correct locale, with preference given to information provided by the user's web browser. If `Zend_Locale(Zend_Locale::ENVIRONMENT)` is used, then preference will be given to the host server's environment configuration, as shown below.

```
<?php
require_once 'Zend/Locale.php';
$locale  = new Zend_Locale();
$locale1 = new Zend_Locale(Zend_Locale::BROWSER);
// default behavior, same as above
$locale2 = new Zend_Locale(Zend_Locale::ENVIRONMENT);
// prefer settings on host server
$locale3 = new Zend_Locale(Zend_Locale::FRAMEWORK);
// prefer framework app default settings
```

ZEND_LOCALE

Automatic Locales

`Zend_Locale` provides three "automatic" locales that do not belong to any language or region. They have the same effect as the method `getDefault()` but without the negative consequences, like creating an instance. These locales can be used anywhere, including with standard locales, with their definition, and with their string representation. These automatic locales offer an easy solution to a variety of situations, as when locales are provided within a browser. Three locales have a slightly different behavior:

`'browser'`
`Zend_Locale` should work with the information provided by the user's Web browser. It is published by PHP in the global variable HTTP_ACCEPT_LANGUAGE. If a user provides more than one locale within the browser, `Zend_Locale` will use the highest quality locale. If the user provides no locale in the browser, or the script is being called from the command line, the automatic locale `'environment'` will be used and returned.

`'environment'`
`Zend_Locale` works with the information provided by the host server, and is published by PHP via the internal function `setlocale()`. If a environment provides more than one locale, `Zend_Locale` will use the first locale encountered. If the host does not provide a locale, the automatic locale `'browser'` will be used and returned.

`'auto'`

`Zend_Locale` should automatically detect any locale which can be worked with. It will first search for a user's locale and then, if unsuccessful, search for the host locale. If no locale can be detected, it will degrade to a "default" locale which is "en" when the user does not set it. This setting can be overwritten. Options are to set a new locale manually, or define a default location. Example: With, and Without, Auto-Detection

```
<?php
require_once 'Zend/Locale.php';
require_once 'Zend/Date.php';

// without automatic detection
// $locale = new Zend_Locale(Zend_Locale::BROWSER);
// $date = new Zend_Date($locale);

// with automatic detection
$date = new Zend_Date('auto');
```

ZEND_LOCALE

Performance Considerations

The performance of `Zend_Locale` and its subclasses can be boosted by the use of `Zend_Cache`, specifically the static method `Zend_Locale::setCache($cache)`.

`Zend_Locale_Format` can be sped up the using the option `cache` within `Zend_Locale_Format::setOptions(array('cache' => $adapter));`.
If both classes are used, be sure to only set a cache for `Zend_Locale;` otherwise, the data will be cached twice.

Locale Objects – Copying, Cloning, Serializing

`Zend_Locale` provides localized information for each locale, including local names for days of the week, months, etc. Use object cloning to duplicate a locale object exactly and efficiently. Most locale-aware methods also accept string representations of locales, such as the result of `$locale->toString()`.

If you know many objects should all use the same default locale, explicitly specify the default locale to avoid the overhead of each object determining the default locale.

```
<?php
require_once 'Zend/Locale.php';

$locale = new Zend_Locale('ar');

// Save the $locale object as a serialization
$serializedLocale = $locale->serialize();
// re-create the original object
$localeObject = unserialize($serializedLocale);

// Obtain a string identification of the locale
$stringLocale = $locale->toString();

// Make a cloned copy of the $local object
$copiedLocale = clone $locale;
```

ZEND_LOCALE

Equality – Comparing Locales

Zend_Locale provides a function to compare two locales – all locale-aware classes should provide a similar equality check.

```
<?php
require_once 'Zend/Locale.php';

$locale = new Zend_Locale();
$mylocale = new Zend_Locale('en_US');

// Check if locales are equal
if ($locale->equals($mylocale)) {
```

Default Locales

getDefault() returns an array of relevant locales using information from the user's web browser (if available), information from the environment of the host server, and ZF settings. As with the constructor for Zend_Locale, the first parameter sets the preference for consideration of information (BROWSER, ENVIRONMENT, or FRAMEWORK). The second parameter toggles between returning all matching locales or only the first/best match. Locale-aware components normally use only the first locale. A quality rating is included, as available.

```
<?php
require_once 'Zend/Locale.php';

$locale = new Zend_Locale();

// Return all default locales
$found = $locale->getDefault();
print_r($found);

// Return only browser locales
```

Use corresponding methods to obtain the default locales relevant only to the:

• **BROWSER**	getBrowser()
• **ENVIRONMENT**	getEnvironment()

ZEND_LOCALE

Set New Locales

A new locale can be set with the function `setLocale()`. This function takes a locale string as a parameter. If no locale is given, one is automatically chosen. As `Zend_Locale` objects are 'light', this method exists primarily to cause side-effects for code that have references to the existing instance object.

```php
<?php
require_once 'Zend/Locale.php';
$locale = new Zend_Locale();

// Actual locale
print $locale->toString();

// new locale
$locale->setLocale('aa_DJ');
print $locale->toString();
```

Obtaining the Language and Region

Use `getLanguage()` to obtain a string containing the language code from the string locale identifier, as the method is able to properly interpret language and region codes. Use `getRegion()` to obtain a string containing the two character region code from the string locale identifier.

Obtaining Localized Strings

`getTranslationList()` provides convenient access to localized information of various types, helping to display localized data to a customer without the need for translation. These strings are available as part of Zend Framework. The requested data is always returned as an array. Use an array, not multiple values, for giving more than one value to an explicit type. Example: Some of the `getTranslationList()`Data Available (*see Programmers Guide*)

- `Language`
- `Script`
- `Territory`
- `Variant`
- `Months`
- `Days`
- `Week`
- `Key`
- `Month`
- `Day`
- `Quarters`
- `Type`

For a single translated value, use `getTranslation($value=null, $type=null, $locale=null)` and specify which value to be returned. The Programmers Guide has the complete list of parameters.

ZEND_TRANSLATE

Zend_Translate is Zend Framework's solution for multilingual applications, where the content must be translated into several languages and displayed depending on the user's language. While PHP already offers ways to handle internationalization, they have their own issues: inconsistent API, support only for native arrays or `gettext` (which is not thread-safe), no detection of default language. `Zend_Translate` has none of these issues.

Establishing Multi-Lingual Sites

There are four basic steps in setting up a multi-lingual site:

- Decide which adapter to use
- Create the View and integrate `Zend_Translate` into the code
- Create the source file from the code
- Translate the source file to the desired language

Choose Adapter

Adapters for Zend Translate

`Zend_Translate` can handle different adapters for translation, each of which has its own advantages and disadvantages.

- `Array`
- `Csv`
- `Gettext`
- `Tbx`
- `Tmx`
- `Qt`
- `Xliff`
- `XmlTm`
- `Others`

Deciding which adapter to use is often influenced by outside criteria, such as those of a project or client – consult the online Reference Guide for a in-depth discussion of the good and bad points of using each adapter.

Utilize Adapter Example: Multi-Lingual PHP Code (more examples in Reference Guide)

```
<?php
$translate = new Zend_Translate('gettext', '/my/path/source-
de.mo', 'de');
$translate->addTranslation('//my/path/fr-source.mo', 'fr');

print $translate->_("Example")."\n";
print "=======\n";
print $translate->_("Here is line one")."\n";
printf(
    $translate->_("Today is the %1\$s") . "\n", date("d.m.Y")
);
print "\n";

$translate->setLocale('fr');
print $translate->_("Fix language here is line two") . "\n";
usage here is line two\n";
```

ZEND_TRANSLATE

Performance Considerations

Using `Zend_Cache` internally with `Zend_Translate` accelerates the loading of translation sources, useful for multiple sources or extensive source formats like XML-based files. Give a cache object to the `Zend_Translate::setCache()` method, which takes a instance of `Zend_Cache` as the only parameter. Direct use of an adapter allows for the use of the `setCache()` method.

```
<?php
require_once 'Zend/Translate.php';
require_once 'Zend/Cache.php';

$cache = Zend_Cache::factory('Page', 'File', $frontendOptions,
                             $backendOptions);
Zend_Translate::setCache($cache);
$translate = new Zend_Translate('gettext',
                                '/path/to/translate.mo', 'en');
```

Translation Source Adapters

`Zend_Translate` offers flexibility when it comes to storage of the translation files. The following structures are preferable, though:

• Single-structured Source:	All source files in one directory
• Language-structured Source:	One language per directory
• Application-structured Source:	One dir; multiple-files per language
• Gettext-structured Source:	Utilize old gettext sources within structure
• File-structured Source:	Each file is related to its translation source

Using UTF-8 Source Encoding

To avoid the problems encountered when using two different source encodings, it is a best practice to always use UTF-8 encoding. Otherwise, if multiple forms are used, it is highly probable that one of the languages will not display correctly, as there can only be only one encoding per source file. UTF-8 is a portable format which supports all languages.

ZEND_TRANSLATE

Adapter Options

All adapters support various options – these options are set when the adapter is created. There is one option available to all adapters. '`clear`' decides if translation data should be added to existing content or not. The standard behavior is to add new translation data to existing data. If the translation data is cleared, it is important to note that this step will apply only to the selected language - all other languages will remain untouched.

ADAPTER	OPTION	STANDARD VALUE	DESCRIPTION
all	clear	false	if set to true, translations already read will be cleared. This can be used instead of creating a new instance when reading new translation data
all	scan	null	If set to null, no scanning of the directory structure will be done. If set to `Zend_Translate::LOCALE_DIRECTORY` the locale will be detected within the directory. If set to `Zend_Translate::LOCALE_FILENAME` the locale will be detected within the filename.
Csv	separator	;	Defines which sign is used for separating source and translation

Self-defined options can also be used with all adapters. Utilize the `setOptions()` method, and provide an array with the desired options.

ZEND_TRANSLATE

Source Files

Creating CSV Source Files

CSV source files are small, readable files, easy to manage.

Creating Array Source Files

Array source files provide an easy form of adapter for the translation process, especially for message feedback on whether the code is performing as expected.

Creating Gettext Source Files

Gettext source files are created by GNU's gettext library. There are several free tools available that can parse code files and create the needed gettext source files. These binary files have the ending ***.mo**.

Creating TMX Source Files

TMX source files are a new industry standard, with the advantage of being XML files – they are readable by all editors and by people. TMX files can either be created manually with a text editor, or via tools available on the market.

Source Auto-Detection:

`Zend_Translate` can detect translation sources automatically. It involves the same process as initiating a single-translation source, only a directory is provided instead of a file.

ZEND_TRANSLATE

Handling Languages – Useful Methods

`getLocale():`	can be used to get the actual set language; it can either hold an instance of `Zend_Locale` or the identifier of a locale
`setLocale():`	sets a new standard language for translation. This prevents having to set the optional language parameter more than once to the `translate()` method. If the given language does not exist, or no translation data is available for the language, `setLocale()` tries to downgrade to the language without the region, if any was given. A language of `en_US` would be downgraded to `en`. If the downgraded language cannot be found, an exception is thrown
`isAvailable():`	checks if a given language is already available – returns TRUE if data for the given language exists.
`getList():`	used to get all the set languages for an adapter returned as an array

```
...
// returns the actual set language
$actual = $translate->getLocale();
...
// you can use the optional parameter while translating
echo $translate->_("my_text", "fr");
// or set a new standard language
$translate->setLocale("fr");
echo $translate->_("my_text");
// refer to base language... fr_CH is downgraded to fr and used
$translate->setLocale("fr_CH");
echo $translate->_("my_text");
...
// check if this language exist
if ($translate->isAvailable("fr")) {
    // language exists
}
```

ZEND_TRANSLATE

Handling Languages – Auto-Detection

As long as new translation sources are added only with the `addTranslation()` method then `Zend_Translate` will automatically set the language best-suited for the environment.

Language through Named Directories

Automatic language detection can be invoked by naming the language source directories and supplying the `‘scan’` option to `Zend_Translate.` (not for TMX source files)

Language through Filenames

Another way to detect the language automatically is to use special filenames, for either the entire language file or parts of it. Again, it is required that the `‘scan’` option is set for `Zend_Translate.` There are several ways of naming the source files - complete filename, file extension, and filename tokens.

Checking for Translations

The `isTranslated()` method can be used to determine whether text within a source file has been translated. `isTranslated($messageId, $original = false, $locale = null)` takes as the first parameter the text in question; the optional second parameter determines whether the translation is fixed to the declared language or a lower set of translations can be used; the optional third parameter is the related locale. If you have text which can be translated by `'en'` but not for `'en_US'` you will normally get the translation returned, but by setting `$original` to TRUE, the `isTranslated()` method will return FALSE.

Access to Source Data

Two functions can be used to access the translation source data:

`getMessageIds($locale = null)`, which returns all know messageIDs as an array,

`getMessages($locale = null)`, which returns the complete translation source as an array - the message ID is used as the key and the translation data as the value.

ZEND_DATE

The `Zend_Date` component offers a detailed but simple API for manipulating dates and times. Its methods accept a wide variety of information types , including date parts. `Zend_Date` also supports abbreviated names of months in many languages. `Zend_Locale` facilitates the normalization of localized month and weekday names to timestamps, which may, in turn, be shown localized to other regions.

Internals

- UNIX Timestamp

  All dates and times are represented internally as absolute moments in time, as a UNIX timestamp expressing the difference between the desired time and 1/1/1970, 00:00 GMT

- Date parts as timestamp offsets

  An instance object representing three hours would be expressed as three hours after 1/1/1970, 00:00 GMT

- PHP Functions

  Where possible, `Zend_Date` uses PHP functions to improve performance

Setting a Default Time zone

Before using any date-related functions in PHP or the Zend Framework, make certain the application has a correct default time zone, either by setting the TZ environment variable with the `date.timezone` php.ini setting, or using `date_default_timezone_set()`

```
<?php
date_default_timezone_set('America/Los_Angeles');
// time zone for an American in California
date_default_timezone_set('Europe/Berlin');
```

BASIC METHODS

Create a Date – Instantiate or Static Method

The simplest way of creating a date object is to create the actual date, either by creating a new instance with **`new Zend_Date()`** or by using the static method **`Zend_Date::now().`** Both will return the actual date as new instance of `Zend_Date`. The actual date always includes the actual date and time for the actual set timezone.

ZEND_DATE

Create a Date - Database

Zend_Date makes it very easy to create a date from database date values. So we have one quick and one convenient way of creating dates from database values. The standard output of all databases is quite different even if it looks the same at first. All are part of the ISO Standard and explained through it. So, the easiest way to create a date is to use Zend_Date::TIMESTAMP. Zend_Date::ISO_8601 works with most formats, but is not the most efficient way.

```
<?php
// SELECT UNIX_TIMESTAMP(my_datetime_column) FROM my_table
require_once 'Zend/Date.php';

$date = new Zend_Date($unixtimestamp, Zend_Date::TIMESTAMP);
```

Output a Date

The date in a Zend_Date object may be obtained as an integer or localized string using the get() method. There are many available options.

Setting a Date

The set() method alters the date stored in the object, and returns the final date value as a timestamp (not an object). There are many available options, the same for add() and sub().

Adding/Subtracting a Date

Adding two dates with add() usually involves adding a real date in time with an artificial timestramp representing a date part, such as 12 hours.

Comparing Dates

All basic Zend_Date methods can operate on entire dates contained in the objects, or on date parts, such as comparing the minutes value in a date to an absolute value.

```
<?php
require_once 'Zend/Date.php';
$date = new Zend_Date();
// Comparison of both times
if ($date->compare(10, Zend_Date::MINUTE) == -1) {
    print "This hour is less than 10 minutes old";
} else {
    print "This hour is at least 10 minutes old";
}
```

ZEND_DATE

Working with Dates

Beware of mixing and matching operations with date parts between date objects for different timezones, which generally produce undesireable results, unless the manipulations are only related to the timestamp. Operating on `Zend_Date` objects having different timezones generally works, except as just noted, since dates are normalized to UNIX timestamps on instantiation of `Zend_Date`.

The methods `add()`, `sub()`, `compare()`, `get()`, and `set()` operate generically on dates. In each case, the operation is performed on the date held in the instance object. The `$date` operand is required for all of these methods, except `get()`, and can be a `Zend_Date` instance object, a numeric string, or an integer.

These methods assume `$date` is a timestamp, if it is not an object. However, the `$part` operand controls which logical part of the two dates are operated on, allowing operations on parts of the object's date, such as year or minute, even when `$date` contains a long form date string, such as, "December 31, 2007 23:59:59". The result of the operation changes the date in the object, except for `compare()`, and `get()`.

```php
<?php
require_once 'Zend/Date.php';

$date = new Zend_Date();

// Comparation of both times
if ($date->compare(10, Zend_Date::MINUTE) == -1) {
    print "This hour is less than 10 minutes old";
```

See the online Reference Guide for more detail on date parts, methods, and options.

Using Constants

Whenever a `Zend_Date` method has a `$parts` parameter, one of a particular set of constants can be used as the argument for that parameter, in order to select a specific part of a date or to indicate the date format used or desired (e.g. RFC 822). See Tables 9.7 and 9.8 in the Reference Guide for a list of relevant constants.

ZEND_DATE

Checking Dates

Dates as input are often strings, and it is difficult to ascertain whether these strings are real dates. Therefore, `Zend_Date` has an own static function to check date strings. `Zend_Locale` has an own function `getDate($date, $locale);` which parses a date and returns the proper and normalized date parts. As `Zend_Locale` does not know anything about dates because it is a normalizing and localizing class, the components has an integrated own function `isDate($date);` that checks this. `isDate($date, $format, $locale);` requires one, but can take up to 3 parameters.

```php
<?php
require_once 'Zend/Date.php';

// Checking dates
$date = '01.03.2000';
if (Zend_Date::isDate($date)) {
    print "String $date is a date";
} else {
    print "String $date is NO date";
}

// Checking localized dates
$date = '01 February 2000';
if (Zend_Date::isDate($date,'dd MMMM yyyy', 'en')) {
    print "String $date is a date";
} else {
    print "String $date is NO date";
}

// Checking impossible dates
$date = '30 February 2000';
if (Zend_Date::isDate($date,'dd MMMM yyyy', 'en')) {
    print "String $date is a date";
} else {
    print "String $date is NO date";
}
```

Timezones

Timezones are integral to dates, and the default has to be set in either `php.ini` or by definition within the bootstrap file.

ZEND_CURRENCY

Zend_Currency is part of the I18N core of the Zend_Framework, and handles all issues related to currency, money representation, and formating. It also provides additional informational methods which include localized informations on currencies, which currency is used in which region, etc. Zend_Currency provides the following functions for handling currency and money-related work:

- Complete Locale Support
  Zend_Currency works with all available locales and can access inforrmation on over 100 different localized currency information (currency names, abbreviations, money signs, etc.).

- Reusable Currency Definitions
  While Zend_Currency does not include the value of the currency, and therefore its functionality is not included in Zend_Locale_Format, it does have the advantage that already defined currency representations can be reused.

- Fluid Interface
  Zend_Currency includes the fluid interface where possible.

- Additional Informational Methods
  Zend_Currency includes additional methods that offer information about regions and their corresponding currencies.

Working with Currencies
Creating an instance of Zend_Currencywithout any parameters will force the use of the actual locale, and its related currency. If the system has no default locale, or the locale cannot be detected automatically, Zend_Currency will throw an exception. If this case, set the locale manually.

Optional parameters include:

Currency:	currency itself (for use when multiple currencies exist within one location)
Locale:	use to format the currency

```
<?php
// expect standard locale 'de_AT'
require_once 'Zend/Currency.php';

// creates an instance from 'en_US' using 'USD' which is the
// default currency for 'en_US'
$currency = new Zend_Currency('en_US');
```

ZEND_CURRENCY

Creating Output from a Currency

Use the `toCurrency()` method to convert an existing value to a currency formatted output. The value to be converted can be any normalized number. Localized numbers first have to be converted to an normalized number with `Zend_Locale_Format::getNumber()`. Afterwards it can be used with `toCurrency()` to create a currency output. `toCurrency(array $options)` accepts an array with options which can be used to temporarily set another format or currency representation.

Changing Currency Format

Occasionally, it may be necessary to change the currency format, which includes the following parts:

- Currency Symbol
- Currency Position
- Script
- Number Formatting

Use the `setFormat()` method, which takes an array containing all the options to be changed (display position, script, format, display, precision, name, currency, symbol).

ZEND_VIEW_HELPER_TRANSLATE

Translate Helper

The display of translated content uses not only `Zend_Translate`, but also the Translate View Helper. It is possible to use any instance of`Zend_Translate` and also any subclass of `Zend_Translate_Adapter`. Ways to initiate the Translate View Helper include:

1. Registered, through a previously registered instance in `Zend_Registry` (preferred)
2. Afterwards, through the fluent interface
3. Directly, through initiating the class

Example: Registered

```
<?php
// our example adapter
$adapter = new Zend_Translate('array', array( 'simple' =>
                                              'einfach'), 'de');
Zend_Registry::set('Zend_Translate', $adapter);

// within your view
echo $this->translate('simple');
// this returns 'einfach'
```

ZEND_VIEW_HELPER_TRANSLATE

Example: Within the View

```
<?php
// within your view
$adapter = new Zend_Translate('array', array( 'simple' =>
                                              'einfach'), 'de');
$this->translate()->setTranslator($adapter)->translate('simple');
// this returns 'einfach'
```

Example: Direct Usage

```
<?php
// our example adapter
$adapter = new Zend_Translate('array', array('simple' => 'einfach'
), 'de');

// initiate the adapter
$translate = new Zend_View_Helper_Translate($adapter);
print $translate->translate('simple'); // this returns 'einfach'
```

Adding Parameters

Parameters (single, list, array) can simply be added to the method. Example: Array of parameters

```
<?php
// within your view
$date = array("Monday", "April", "11:20:55");
$this->translate('Today is %1\$s in %2\$s. Actual time:
                 %3\$s', $date);
// Return 'Heute ist Monday in April. Aktuelle Zeit: 11:20:55'
```

Changing Locale (Dynamically or Statically)

Changling Locales, dynamically or statically, also allows for a paramter array or list. In both cases, the locale must be given as the last, single parameter. Example: Dynamically

```
<?php
// within your view
$date = array("Monday", "April", "11:20:55");
$this->translate('Today is %1\$s in %2\$s. Actual time: %3\$s',
                 $date, 'it');
```

TEST YOUR KNOWLEDGE : QUESTIONS

1 Which method of Zend_Locale will check if a given string is a locale?

____

2 Which Zend_Date constant should you use when you want to have the date formatted for an RSS feed?

a. Zend_Date::RSS_FEED

b. Zend_Date::RSS2

c. Zend_Date::RSS

d. Zend_Date::RSSFEED

TEST YOUR KNOWLEDGE : ANSWERS

★ = CORRECT

1

Which method of Zend_Locale will check if a given string is a locale?

★ isLocale

2

Which Zend_Date constant should you use when you want to have the date formatted for an RSS feed?

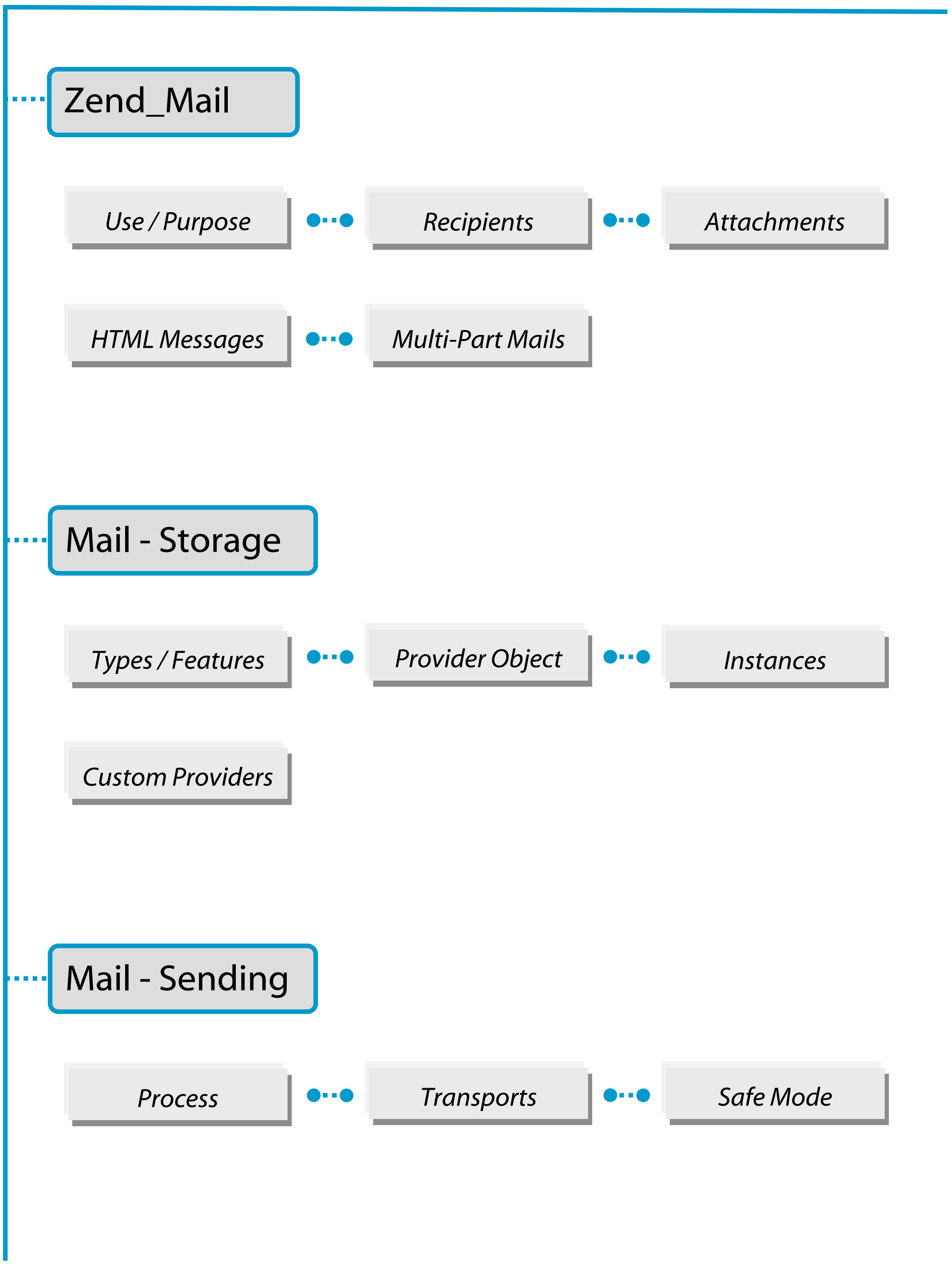
a. Zend_Date::RSS_FEED

b. Zend_Date::RSS2

★ c. Zend_Date::RSS

d. Zend_Date::RSSFEED

CERTIFICATION TOPIC : MAIL



ZEND FRAMEWORK: MAIL

For the exam, here's what you should know already …

You should be able to explain the purpose of the Mail component (generating, sending, receiving emails).

You should know how to add recipients (To, CC, BCC) and attachments to an email.

You should know how to compose an HTML message.

You should know how to create a multi-part message, message attachments, and ways to change `mime_part` properties.

You should know which storages are supported by `Zend_Mail` (`Mbox`, `Maildir`, `Pop3`, `IMAP`).

You should know which features are supported by different mail storage providers (local/remote. folders, flags, quotas).

ZEND_MAIL

`Zend_Mail` provides a general feature set to compose and send both text and MIME-compliant multi-part email messages. Mail can be sent via `Zend_Mail_Transport_Smtp` or `Zend_Mail_Transport_Sendmail` (default).

```
<?php
require_once 'Zend/Mail.php';
$mail = new Zend_Mail();
$mail->setBodyText('This is the text of the mail.');
$mail->setFrom('somebody@example.com', 'Some Sender');
$mail->addTo('somebody_else@example.com', 'Some Recipient');
$mail->setSubject('TestSubject');
$mail->send();
```

For most mail attributes there are "`get`" methods to read the information stored in the mail object. For further details, please refer to the API documentation. A special one is `getRecipients()`. It returns an array with all recipient email addresses that were added prior to the method call. For security reasons, `Zend_Mail` filters all header fields to prevent header injection with `newline (\n)` characters.

ZF also provides a convenient fluent interface for using most methods of the `Zend_Mail` object. A fluent interface means that each method returns a reference to the object on which it was called, allowing an immediate call to another method.

```
<?php
require_once 'Zend/Mail.php';
$mail = new Zend_Mail();
$mail->setBodyText('This is the text of the mail.')
     ->setFrom('somebody@example.com', 'Some Sender')
     ->addTo('somebody_else@example.com', 'Some Recipient')
     ->setSubject('TestSubject')
     ->send();
```

ZEND_MAIL

Sending Multiple Emails (using SMTP)

By default, a single SMTP transport creates a single connection and re-uses it for the lifetime of the script execution. Multiple emails can be sent through this SMTP connection. An RSET command is issued before each delivery to ensure the correct SMTP handshake is followed. For each mail delivery to have a separate connection, the transport would need to be created and destroyed before and after each `send()` method call.

```php
<?php
// Load classes
require_once 'Zend/Mail.php';

// Create transport
require_once 'Zend/Mail/Transport/Smtp.php';
$transport = new Zend_Mail_Transport_Smtp('localhost');

// Loop through messages
for ($i = 0; $i > 5; $i++) {
    $mail = new Zend_Mail();
    $mail->addTo('studio@peptolab.com', 'Test');
    $mail->setFrom('studio@peptolab.com', 'Test');
    $mail->setSubject('Demonstration -
 Sending Multiple Mails per SMTP Connection');
    $mail->setBodyText('...Your message here...');
    $mail->send($transport);
}
```

HTML Email

To send an email in HTML format, set the body using the method `setBodyHTML()` instead of `setBodyText()`. The `MIME` content type will automatically be set to `text/html` then. If you use both HTML and Text bodies, a multipart/alternative `MIME` message will be automatically generated.

```php
<?php
require_once 'Zend/Mail.php';
$mail = new Zend_Mail();
$mail->setBodyText('My Nice Test Text');
$mail->setBodyHtml('My Nice <b>Test</b> Text');
$mail->setFrom('somebody@example.com', 'Some Sender');
$mail->addTo('somebody_else@example.com', 'Some Recipient');
$mail->setSubject('TestSubject');
$mail->send();
```

ZEND_MAIL

Sending Attachments

Files can be attached to an e-mail using the `createAttachment()` method. The default behavior of `Zend_Mail` is to assume the attachment is a binary object (application/octet-stream), should be transferred with base64 encoding, and is handled as an attachment. These assumptions can be overridden by passing more parameters to `createAttachment()`.

```
<?php
require_once 'Zend/Mail.php';
$mail = new Zend_Mail();
// build message...
$mail->createAttachment($someBinaryString);
$mail->createAttachment($myImage, 'image/gif',
        Zend_Mime::DISPOSITION_INLINE, Zend_Mime::ENCODING_8BIT);
```

For more control over the MIME part generated for this attachment, use the return value of `createAttachment()` to modify its attributes. The `createAttachment()` method returns a `Zend_Mime_Part` object.

```
<?php
require_once 'Zend/Mail.php';
$mail = new Zend_Mail();

$at = $mail->createAttachment($myImage);
$at->type        = 'image/gif';
$at->disposition = Zend_Mime::DISPOSITION_INLINE;
$at->encoding    = Zend_Mime::ENCODING_8BIT;
$at->filename    = 'test.gif';

$mail->send();
```

Using Mixed Transports

To send different emails through different connections, the transport object can be directly passed to `send()` without a prior call to `setDefaultTransport()`. The passed object will override the default transport for the actual `send()` request generated.

```
<?php
require_once 'Zend/Mail.php';
$mail = new Zend_Mail();              // build message...
require_once 'Zend/Mail/Transport/Smtp.php';
$tr1 = new Zend_Mail_Transport_Smtp('server@example.com');
$tr2 = new Zend_Mail_Transport_Smtp('other_server@example.com');
$mail->send($tr1);
$mail->send($tr2);
$mail->send();  // use default again
```

ZEND_MAIL

Adding Recipients

Recipients can be added in three ways:

• `addTo()`	Adds a recipient to the email with a 'To' header
• `addCc()`	Adds a recipient to the email with a 'Cc' header
• `addBcc()`	Adds a recipient to the email not visible in the header

Note: `addTo()` and `addCc()` both accept a second, optional parameter that is used to provide a readable name for the recipient in the header.

Additional Headers

Arbitrary mail headers can be set by using the `addHeader()` method. It requires two parameters containing the name and the value of the header field. A third optional parameter determines if the header should have one or multiple values.

```
<?php
require_once 'Zend/Mail.php';
$mail = new Zend_Mail();
$mail->addHeader('X-MailGenerator', 'MyCoolApplication');
$mail->addHeader('X-greetingsTo', 'Mom', true);
// multiple values
$mail->addHeader('X-greetingsTo', 'Dad', true);
```

Encoding

Text and HTML message bodies are encoded with the '`quotedprintable`' mechanism by default. All other attachments are encoded via base64 if no other encoding is given in the `addAttachment()`call or assigned to the `MIME` part object later. 7Bit and 8Bit encoding currently only pass on the binary content data.

`Zend_Mail_Transport_Smtp` encodes lines starting with one dot or two dots so that the mail does not violate the SMTP protocol.

ZEND_MAIL

Reading Mail Messages

Zend_Mail can read mail messages from several local or remote mail storages, all with the same basic API to count and fetch messages. Some of them implement additional interfaces for relatively uncommon features. The table below provides a feature overview:

FEATURE	Mbox	Maildir	Pop3	IMAP
Storage Type	local	local	remote	remote
Fetch message	Yes	Yes	Yes	Yes
Fetch mime-part	Emulated	Emulated	Emulated	Emulated
Folders	Yes	Yes	No	Yes
Create message/ folder	No	todo	No	todo
Flags	No	Yes	No	Yes
Quota	No	Yes	No	No

Code Example: Pop3

```
<?php
$mail = new Zend_Mail_Storage_Pop3(array('host'     =>'localhost',
                                         'user'     =>'test',
                                         'password'=>'test'));
echo $mail->countMessages() . " messages found\n";
foreach ($mail as $message) {
    echo "Mail from '{$message->from}': {$message->subject}\n";
}
```

Opening a Local Storage

Mbox and Maildir are the two supported formats for local mail storages, both in their most simple formats. To read from a Mbox file, give the filename to the constructor of Zend_Mail_Storage_Mbox. (*see Programmers Guide for other examples*)

```
<?php
$mail = new Zend_Mail_Storage_Mbox(array('filename' =>
        '/home/test/mail/inbox'))
```

ZEND_MAIL

Fetching Mail Messages

Messages can be retrieved once the storage has been opened, using the message number and the method `getMessage()`. Array access is also supported, but only with the default values.

```php
<?php
$message = $mail->getMessage($messageNum);
```

Headers can be fetched via properties, or the method `getHeader()` for greater control, unusual header names, or to retrieve multiple headers with the same name as an array (Example given below).

```php
<?php
// get header as property – the result is always a string, with
// new lines between the single occurrences in the message
$received = $message->received;

// the same via getHeader() method
$received = $message->getHeader('received', 'string');

// better an array with a single entry for every occurrences
$received = $message->getHeader('received', 'array');
foreach ($received as $line) {
    // do stuff
}

// if you don't define a format you'll get the internal
// representation (string for single headers, array for multiple)
$received = $message->getHeader('received');
if (is_string($received)) {
    // only one received header found in message
}
```

Content can be fetched using `getContent()`, as long as the email does not have a multi-part message. Content is retrieved only when needed (“late-fetch”). To check for a multi-part message, use the method `isMultipart()`. With multi-part messages, use the method `getPart()` to create an instance of `Zend_Mail_Part` (the base class of `Zend_Mail_Message`). This will make available the same methods: `getHeader()`, `getHeaders()`, `getContent()`, `getPart()`, `isMultipart` and the properties for headers.

ZEND_MAIL

Checking for Flags

Maildir and IMAP support storing flags. The class `Zend_Mail_Storage` has constants for all known Maildir and IMAP system flags, `Zend_Mail_Storage::FLAG_<flagname>`. `Zend_Mail_Message` uses the method `hasFlag()` to check for flags. `getFlags()` retrieves all set flags.

```
<?php
// find unread messages
echo "Unread mails:\n";
foreach ($mail as $message) {
    if ($message->hasFlag(Zend_Mail_Storage::FLAG_SEEN)) {
        continue;
    }
    // mark recent/new mails
    if ($message->hasFlag(Zend_Mail_Storage::FLAG_RECENT)) {
        echo '! ';
    } else {
        echo '  ';
    }
    echo $message->subject . "\n";
}
// check for known flags
$flags = $message->getFlags();
echo "Message is flagged as: ";
foreach ($flags as $flag) {
    switch ($flag) {
        case Zend_Mail_Storage::FLAG_ANSWERED:
            echo 'Answered ';
            break;
        case Zend_Mail_Storage::FLAG_FLAGGED:
            echo 'Flagged ';
            break;

        // ...
        // check for other flags
        // ...

        default:
            echo $flag . '(unknown flag) ';
    }
}
```

ZEND_MAIL

Using Folders

All storage types except Pop3 support folders, or 'mailboxes'. The interface implemented by all storage-supporting folders is called `Zend_Mail_Storage_Folder_Interface`.

All related classes have an additional optional parameter called `folder`, which is the folder selected after login, in the constructor.

For local storage, use the separate classes `Zend_Mail_Storage_Folder_Mbox` or `Zend_Mail_Storage_Folder_Maildir`. Both need one parameter, `dirname,` with the name of the base dir.

The format for `maildir` is as defined in `maildir++` (with a dot as the default delimiter). `Mbox` is a directory hierarchy with `Mbox` files. If the `Mbox` file called `INBOX` is missing from the `Mbox` base directory, set another folder in the constructor. `Zend_Mail_Storage_Imap` already supports folders by default.

Example: opening storages

```php
<?php
// Mbox with folders
$mail = new Zend_Mail_Storage_Folder_Mbox(array
               ('dirname' => '/home/test/mail/'));

// Mbox with default folder not called INBOX; also works for
// Zend_Mail_Storage_Folder_Maildir and Zend_Mail_Storage_Imap
$mail = new Zend_Mail_Storage_Folder_Mbox(array
               ('dirname'   => '/home/test/mail/',
                'folder'    => 'Archive'));

// maildir with folders
$mail = new Zend_Mail_Storage_Folder_Maildir(array
               ('dirname'   => '/home/test/mail/'));

// maildir with colon as delimiter, as suggested in Maildir++
$mail = new Zend_Mail_Storage_Folder_Maildir(array
               ('dirname'   => '/home/test/mail/'
                'delim'     => ':'));

// imap is the same with and without folders
$mail = new Zend_Mail_Storage_Imap(array
               ('host'      => 'example.com'
               'user'       => 'test',
               'password'   => 'test'));
```

ZEND_MAIL

Using Folders (continued)
The method `getFolders($root=null)`returns the folder hierarchy starting with either the root or given folder. The return is an instance of `Zend_Mail_Storage_Folder`, which implements `RecursiveIterator;` all 'children' are also instances of `Zend_Mail_Storage_Folder`. Each of these instances has a local and a global name returned by the methods `getLocalName()` and `getGlobalName()`. The global name is the absolute name from the *root* folder (including delimiters); the local name is the name in the *parent* folder.

GLOBAL NAME	LOCAL NAME
/INBOX	INBOX
/Archive/2005	2005
List.ZF.General	General

The key of the current element is the local name when using the iterator. The global name is also returned by the magic method `__toString()`. Some folders may not be selectable - they cannot store messages and selecting them results in an error. This can be checked with the method `isSelectable()`. Selected folders are returned by the method `getSelectedFolder()` – changing folders is accomplished with `selectFolder()` and the global name as a parameter. (Code Example: Folder Hierarchy view)

```
<?php
$folders=new RecursiveIteratorIterator($this->mail->getFolders(),
            RecursiveIteratorIterator::SELF_FIRST);
echo '<select name="folder">';
foreach ($folders as $localName => $folder) {
    $localName = str_pad('', $folders->getDepth(), '-',
                         STR_PAD_LEFT) . $localName;
    echo '<option';
    if (!$folder->isSelectable()) {
        echo ' disabled="disabled"';
    }
    echo ' value="' . htmlspecialchars($folder) . '">'
        . htmlspecialchars($localName) . '</option>';
}
echo '</select>';
```

ZEND_MAIL

Caching Instances

Zend_Mail_Storage_Mbox, Zend_Mail_Storage_Folder_Mbox, Zend_Mail_Storage_Maildir and Zend_Mail_Storage_Folder_Maildir implement the magic methods __sleep() and __wakeup(), which means they are serializable. This avoids parsing the files or directory tree more than once.

The disadvantage is that the Mbox or Maildir storage should not change. Some easy checks are done, like reparsing the current Mbox file if the modification time changes or reparsing the folder structure if a folder has vanished – this still results in an error, but a later search can be made for another folder. A best practice is to have something like a signal file for changes and check it before using the cached instance.

```php
<?php
// there's no specific cache handler/class used here,
// change the code to match your cache handler
$signal_file = '/home/test/.mail.last_change';
$mbox_basedir = '/home/test/mail/';
$cache_id = 'example mail cache ' . $mbox_basedir . $signal_file;

$cache = new Your_Cache_Class();
if (!$cache-
>isCached($cache_id) || filemtime($signal_file) > $cache-
>getMTime($cache_id)) {
    $mail = new Zend_Mail_Storage_Folder_Pop3(array('dirname' =>
$mbox_basedir));
} else {
    $mail = $cache->get($cache_id);
}

// do stuff ...

$cache->set($cache_id, $mail);
```

Extending Protocol Classes

Remote storage uses two classes: Zend_Mail_Storage_<Name> and Zend_Mail_Protocol_<Name>. The protocol class translates the protocol commands and responses from and to PHP, like methods for the commands or variables with different structures for data. The other/main class implements the common interface. Additional protocol features can extend the protocol class and be used within the constructor of the main class.

Code Example given on next page…

ZEND_MAIL

Code Example: there is a need to knock different ports before connecting to POP3

```
<?php
require_once 'Zend/Loader.php';
Zend_Loader::loadClass('Zend_Mail_Storage_Pop3');

class Example_Mail_Exception extends Zend_Mail_Exception
{
}
class Example_Mail_Protocol_Exception extends Zend_Mail_Protocol_Exception
{
}
class Example_Mail_Protocol_Pop3_Knock extends Zend_Mail_Protocol_Pop3
{
    private $host, $port;
    public function __construct($host, $port = null)
    {
        // no auto connect in this class
        $this->host = $host;
        $this->port = $port;
    }
    public function knock($port)
    {
        $sock = @fsockopen($this->host, $port);
        if ($sock) {
            fclose($sock);
        }
    }
    public function connect($host = null, $port = null, $ssl = false)
    {
        if ($host === null) {
            $host = $this->host;
        }
        if ($port === null) {
            $port = $this->port;
        }
        parent::connect($host, $port);
    }
}
class Example_Mail_Pop3_Knock extends Zend_Mail_Storage_Pop3
{
    public function __construct(array $params)
    {
        // ... check $params here! ...
        $protocol = new Example_Mail_Protocol_Pop3_Knock($params['host']);

        // do our "special" thing
        foreach ((array)$params['knock_ports'] as $port) {
            $protocol->knock($port);
        }

        // get to correct state
        $protocol->connect($params['host'], $params['port']);
        $protocol->login($params['user'], $params['password']);

        // initialize parent
        parent::__construct($protocol);
    }
}
$mail = new Example_Mail_Pop3_Knock(array('host'       => 'localhost',
                                          'user'       => 'test',
                                          'password'   => 'test',
                                          'knock_ports' => array(1101, 1105, 1111)));
```

SAMPLE EXAM QUESTIONS

TEST YOUR KNOWLEDGE : QUESTIONS

1

```
With quotas enabled which methods might fail because you're
over quota? (choose TWO)
   a. appendMessage()
   b. removeMessage()
   c. createFolder()
   d. getMessage()
```

2

```
How would you connect to a Pop3 server using TLS??

   a. <?php
      $mail = new Zend_Mail_Storage_Pop3_Tls(array('host'
                     ='example.com','user' ='test')); ?>

   b. <?php
      $mail = new Zend_Mail_Storage_Pop3(array('host'
                 ='example.com','user' ='test',
                                 'ssl' = true)); ?>

   c. <?php
      $mail = new Zend_Mail_Storage_Pop3(array('host'
                 ='example.com', 'user' ='test',
                                'ssl' = 'tls')); ?>

   d. <?php
      $mail = new Zend_Mail_Storage_Pop3(array('host'
                 ='example.com', 'user' ='test',
                                 'tls' = true)); ?>
```

Test Your Knowledge : Answers

★ = CORRECT

1

With quotas enabled which methods might fail because you're over quota? (choose TWO)

★ a. appendMessage()

★ b. removeMessage()

c. createFolder()

d. getMessage()

2

Which one of the following will NOT assign the values to the view object?

```
a. <?php
   $mail = new Zend_Mail_Storage_Pop3_Tls(array('host'
                      ='example.com','user' ='test')); ?>

b. <?php
   $mail = new Zend_Mail_Storage_Pop3(array('host'
                 ='example.com','user' ='test',
                                  'ssl' = true)); ?>

★ c. <?php
   $mail = new Zend_Mail_Storage_Pop3(array('host'
                 ='example.com', 'user' ='test',
                                 'ssl' = 'tls')); ?>

d. <?php
   $mail = new Zend_Mail_Storage_Pop3(array('host'
                 ='example.com', 'user' ='test',
                                   'tls' = true)); ?>
```

CERTIFICATION TOPIC : MODEL-VIEW-CONTROLLER

Pattern Basics

Components	*Purpose of Pattern*	*Definitions*

Component Areas of Responsibility

Model Function	*View Function*	*Controller Function*
Front Controller	*Action Controllers*	*Action Helpers*
Plugins	*ViewRenderer*	*Dispatcher*
Router	*Request Object*	*Response Object*

Dispatch Loop / Bootstrap / Rewrite

Definitions	*Use / Purpose*	*Hooks*

Zend_Controller

Definitions · Use / Purpose · Front Controller

Action Controllers · Action Methods · Modules

Zend_View

Definitions · View Scripts · View Helpers

View Rendering · View Partials · View Placeholders

View Filters · View Variables · Content (Escape)

Zend_Layout

Definitions · Initialization · Access Content

Zend Framework: MVC Pattern

For the exam, here's what you should know already…

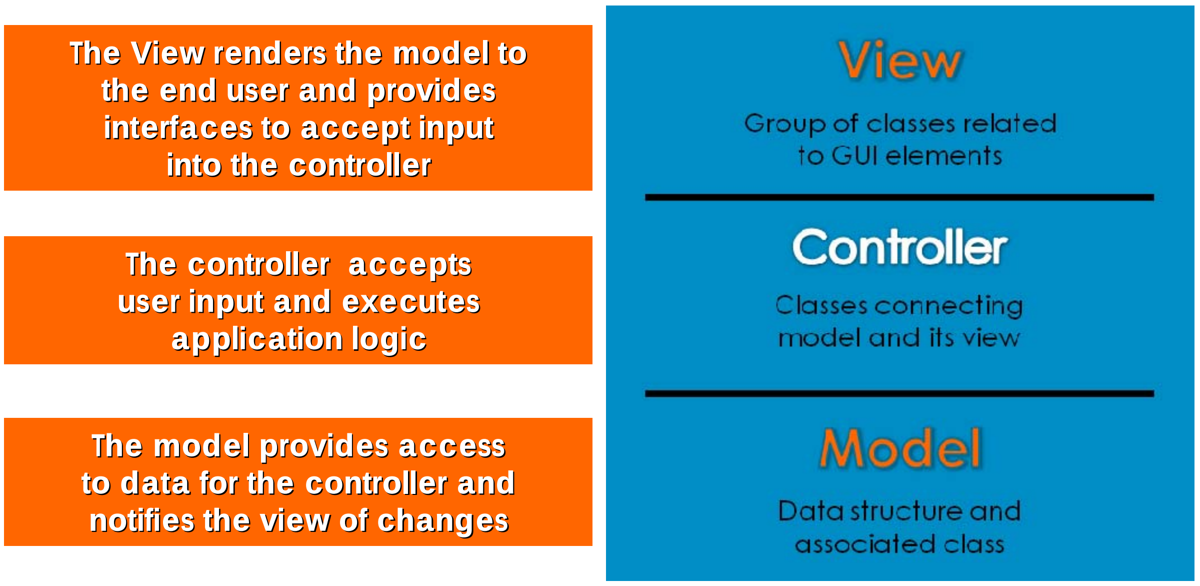
The Model – View – Controller (MVC) design pattern is an object-oriented design pattern for creating applications, useful for both client and web GUI (Graphical User Interfaces). This pattern was originally designed for the client side, but has since evolved options especially suited for the web (Example: current Controller design).

It serves to modularize an application – to separate the business logic from the user display/interaction – by the use of controllers. This separation makes it easier to alter one facet of the application (Example: data source) without affecting the other, thereby avoiding an entire application re-design for a simple change.

Zend Framework has been based on the MVC design, because of the pattern's flexibility, efficiency, and ease of use.

The MVC structure is most often employed when creating *new* applications.

The diagram below illustrates the three component layers that compose the pattern, and the general purpose for each.


The View renders the model to the end user and provides interfaces to accept input into the controller
The controller accepts user input and executes application logic
The model provides access to data for the controller and notifies the view of changes
View
Group of classes related to GUI elements
Controller
Classes connecting model and its view
Model
Data structure and associated class

ESSENTIAL TERMS

Model: The Model is used to implement domain (business) logic, such as performing calculations. This often involves accessing data sources. The application designer has complete freedom to decide where it comes from and how it is used, if it is used at all. Data within the Model can exist in a database, LDAP server, web service, etc.

As the MVC structure provides the ability to later alter aspects without an entire re-design, it is relatively easy to change sources by changing the Model, not the entire application (Example: authentication from a database to an LDAP server with expansion…).

View: The View is basically synonymous with HTML for many applications. It is the template layer, where all HTML rendering takes place, and where everything to be displayed to a user is assembled. It does not contain either business or application logic.

In most modern (2.0) applications, such as AJAX-based applications, the View does not have to be HTML -- it can render XML, JSON, or any other format required by the application.

Controller: The Controller layer of the pattern is actually the only one of the three that is required. The Controller accepts and then processes the input, while interacting with component libraries, Models, and Views.

The Controller ultimately handles the request and outputs the results to the user based on business logic. The system within ZF is designed to be lightweight, modular, and extensible.

More in-depth information on **`Zend_Controller`** and **`Zend_View`** can be found in later sections of this guide.

Bootstrap: The bootstrap file is the single entry point to the application – all requests are routed through this page. It is responsible for loading and dispatching the Front Controller. Example:

```
require_once 'Zend/Controller/Front.php';
Zend_Controller_Front::run('/path/to/app/controllers');
```

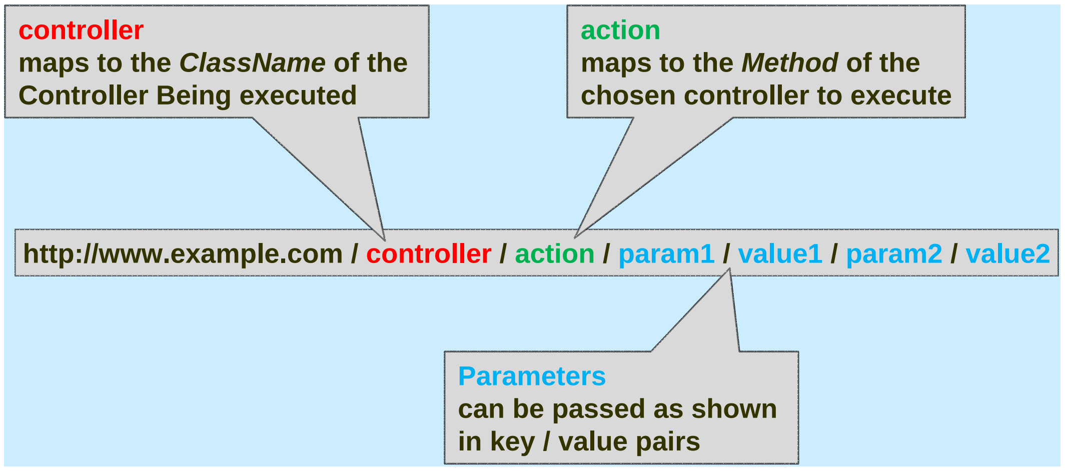
Zend_Controller

`Zend_Controller` is the core of the MVC pattern within Zend Framework. Controllers allow the Business logic (Model) to be separated from the Display logic (View) and yet still function integrally through the use of requests and responses executed by action controllers.

Diagrams for Context

MVC Mapping to URL

This diagram depicts a sample URL and how its structure maps to the MVC pattern. It clearly illustrates how an action is mapped to the controller, along with any parameters assigned in key-value pairs. Keep this structure in mind as the various controllers are further discussed.


controller
maps to the *ClassName* of the Controller Being executed
action
maps to the *Method* of the chosen controller to execute
http://www.example.com / controller / action / param1 / value1 / param2 / value2
Parameters
can be passed as shown in key / value pairs

Dispatch Process

The Dispatch process is composed of three steps:

Route :	takes place only once, using data in Request Object when `dispatch()` is called
Dispatch :	takes place in a Loop… Request Object indicates multiple actions, or Controller (Plugin) resets Request Object, forcing additional actions
Response :	sends final response, including all headers and content

Zend_Controller: Extensions

The Zend Controller requires a set of ZF component extensions and plugins to function properly, with at least one or more paths to directories containing action controllers. A variety of methods may also be invoked to further tailor the front controller environment and that of its helper classes. The essential extensions, both for development and for the exam, are provided below. (*See the online* Reference Guide *for more detail.)*

Zend_Controller_Front

`Zend_Controller_Front` processes all requests received by the server, then routes and dispatches them to the appropriate action controllers. Key points about this component and its extensions:

- It implements the Singleton pattern (one instance available at a time) **
- It registers a plugin broker with itself, allowing user code to be called when certain events occur in controller process; the event methods are defined in `Zend_Controller_Front_Plugin_Abstract`
- The ErrorHandler plugin and ViewRenderer action helper plugin are loaded by default; the plugins are created by including and extending the abstract class

```
require_once 'Zend/Controller/Plugin/Abstract.php';
class MyPlugin extends Zend_Controller_Plugin_Abstract
{
```

- `Zend_Controller_Front::run($path)` is a static method, which sets the path to a directory of controllers and dispatches the request; it performs the following 3 methods at once:

`getInstance`()	retrieves a Front Controller instance; it is the only way to instantiate a front controller object **
`setControllerDirectory`()	directs the Dispatcher where to find action controller class files (register path);
`dispatch``()`	can accept customized Request & Response objects; otherwise, it looks for previously registered objects to use, or instantiates default versions (HTTP default); it also checks for registered router and dispatcher objects, instantiating the defaults if none are found

Zend_Controller_Action

`Zend_Controller_Action` is an abstract class that implements action controllers within the MVC pattern. Key points about this controller and its related extensions:

- To utilize the component, extend the action controller itself, placing the class file in the appropriate controller directory; then create action methods corresponding to the desired actions
- It can execute a variety of actions, such as custom initializations, default actions, pre-/post-dispatch hooks, helper methods

Hooks: specify 2 methods to bracket a requested action:

`preDispatch()`	can be used to verify authentication and ACLs prior to an action
`postDispatch()`	can perform post-processing actions, such as determining additional actions to dispatch

Action Helpers: inject runtime and/or on-demand functionality into any action controller that extends `Zend_Controller_Action`, reducing the need for extensions

correspond to the most common action controller functions

utilize a Broker system (as does `Zend_Controller_Action_`**`Plugin`**)

may be loaded / called on-demand, or instantiated at the time a request is made [bootstrap], or an action controller is created, using `init()`

Helper Methods: use `getHelper()` *or* the Plugin Broker's `__get()` function *or* the Helper's method `direct()` … this last option is illustrated below:

```
$this->_helper->MessageName
        ('We did something in the last request');
```

Zend_Controller_Action

ViewRenderer:	one of several actions helpers included by default within ZF It automates the process of setting up the view object in the controllers and rendering the view loaded by the front controller, by default; called by the action controller to render the appropriate view script for a given controller/action request.

Creating an Action Helper

Action helpers extend `Zend_Controller_Action_Helper_Abstract`, an abstract class that provides the basic interface and functionality required by the helper broker. These include the following methods:

- `setActionController`() is used to set the current action controller
- `init`(), triggered by the helper broker at instantiation, can be used to trigger initialization in the helper; this can be useful for resetting state when multiple controllers use the same helper in chained actions
- `preDispatch`(), is triggered prior to a dispatched action
- `postDispatch`() is triggered when a dispatched action is done -- even if a `preDispatch`() plugin has skipped the action; mainly useful for cleanup
- `getRequest`() retrieves the current request object
- `getResponse`() retrieves the current response object
- `getName`() retrieves the helper name. It retrieves the portion of the class name following the last underscore character, or the full class name otherwise.
    - For example, if the class is named `Zend_Controller_Action_Helper_Redirector`, it will return `Redirector`; a class named `FooMessage` will simply return itself

You may optionally include a `direct`() method in your helper class. If defined, it allows you to treat the helper as a method of the helper broker, in order to allow easy, one-off usage of the helper.

Zend_Controller_Request_Abstract

also known as the **Request Object**
It is a simple value object passed between **`Zend_Controller_Front`** and the router, dispatcher, and controller classes.

- Values for the controller, action, and parameters are packaged into a request extension
- Associated methods retrieve the request URI, path, $_GET & $_POST parameters, etc.
- Tracks whether action has been dispatched via **`Zend_Controller_Dispatcher`**

`Zend_Controller_Request_Http` The Default *request* in ZF

Methods

- The object provides methods for setting and retrieving controller & action names, along with associated parameters:

`getModuleName()`	`setModuleName()`	
`getControllerName()`	`setControllerName()`	
`getActionName()`	`setActionName()`	
`getParam()`	`setParam()`	*for arrays, use* Params

Zend_Controller_Response_Abstract

also known as the **Response Object**
This defines a base response class used to collect and return responses from action controllers; it collects both body and header information.

`Zend_Controller_Response_HTTP` The Default *response* in ZF, for use in an HTTP environment

Zend_Controller_Router

It defines all routers used in allocating requests to the correct controller and action, with optional parameters set in the Request object via **Dispatcher_Standard.**

The routing process occurs only once when the request is received and before the first controller is dispatched

Zend_Controller_Router_Interface	The component used to define routers
Zend_Controller_Router_Rewrite	Takes URI endpoint and breaks it into a controller action and parameters Based on URL path; also used for matching arbitrary paths Works in conjunction with mod_rewrite or other rewriters, based on the web server used; the simplest rule to use is a single Apache mod_rewrite rule, although others are supported User-defined routes are added by calling addRoute() and passing a new instance of a class implementing **Zend_Controller_Router_Interface**

```
RewriteEngine on
RewriteRule !\.(js|ico|gif|jpg|png|css)$ index.php
```

```
/* Create a router */
$router = $ctrl->getRouter();
// returns a rewrite router by default
$router->addRoute(
 'user' ,
 new Zend_Controller_Router_Route(
  'user/:username',
  array(
   'controller' => 'user',
   'action' -> 'info'
  )
 )
);
```

Zend_View

`Zend_View` is a class for working with the Display logic of the MVC design pattern, and provides a system of Helpers, Output Filters, and Variable escaping.

This class is independent of template type – you can use PHP as the template language, or use other template systems with view scripts. You can also create your own view implementation by implementing `Zend_View_Interface` in a custom class.

There are three distinct steps in using `Zend_View`:

(1) Create an instance of the View

(2) Assign the variables to the View via a Controller Script

(3) Render the View (the Controller instructs `Zend_View` to render the output, and transfers control of the display to the View Script)

Code Examples for Context: Retrieving and Displaying Data

Controller Script Example: Retrieve Book Data from Controller

```
// use a model to get the data for book authors and titles
$data = array(
    array(
        'author' => 'Isabelle Allende',
        'title' => 'The House of Spirits'
    ),
    array(
        'author' => 'Dr. Jacob Bronowski',
        'title' => 'The Ascent of Man'
    )
);

// now assign the book data to a Zend_View instance
Zend_Loader::loadClass('Zend_View');
$view = new Zend_View();
$view->books = $data;

// and render a view script called "booklist.php"
echo $view->render('booklist.php');
```

View Script Example: Display Book Data (`booklist.php`, as in Controller Script)

```
<?php if ($this->books): ?>

    <!-- A table of some books. -->
    <table>
        <tr>
            <th>Author</th>
            <th>Title</th>
        </tr>

        <?php foreach ($this->books as $key => $val): ?>
        <tr>
            <td>
               <?php echo $this->escape($val['author']) ?>
               </td>
            <td><?php echo $this->escape($val['title']) ?>
               </td>
        </tr>
        <?php endforeach; ?>

    </table>

<?php else: ?>

    <p>There are no books to display.</p>

<?php endif;
```

Note how the "`escape()`" method is used to apply output escaping to variables

Essential View Elements

View Helpers	Equate to a class Format: `Zend_View_Helper_SampleName`, and are called by using `$this->sampleName()`
Initial Helpers	Set of view helpers automatically available within ZF Most are related to Form element generation and output escaping Some create route-based URLs and HTML lists, and declare variables
Action View Helpers	Enable view scripts to dispatch a specific controller action; following the dispatch, the response object result is returned; Action view helpers are used when a particular action could generate re-usable content;
Partial Helper	Used to render a specific template within its own variable scope; Primarily used for re-usable template fragments with no chance of variable name conflicts; Also allows for calling partial view scripts from specific modules;

```
<?php // partial.phtml ?>
<ul>
    <li>From: <?= $this->escape($this->from) ?></li>
    <li>Subject: <?= $this->escape($this->subject) ?></li>
</ul>

// Call it from the View Script
<?= $this->partial('partial.phtml', array(
    'from' => 'Team Framework',
    'subject' => 'view partials')); ?>

// Which renders…
 <ul>
    <li>From: Team Framework</li>
    <li>Subject: view partials</li
</ul>
```

Placeholder Helper	Used to persist content between view scripts and view instances; Also, to aggregate content, capture view script content for re-use, add pre- and post-text to content (custom separators for aggregate);

```
<?php $this->placeholder('foo')->set("Text for later") ?>

<?php
 echo $this->placeholder('foo');// outputs "Text for later"
```

Essential View Elements

Helper Paths

The controller can specify a stack of paths where `Zend_View` should search for helper classes;

By default, it looks in `Zend/View/Helper*`;

```
<?php
$view = new Zend_View();

//Set path to /path/to/more/helpers, prefix 'My_View_Helper'
$view->setHelperPath('/path/to/more/helpers',
  'My_View_Helper');
```

To specify other locations to search, use `addHelperPath()/setHelperPath()`;

`Zend_View` will look at the most recently added path for a requested helper class

This allows for adding or overriding the initial distribution of helpers by using custom helpers

```
<?php
$view = new Zend_View();
// Add /path/to/some/helpers… class prefix 'My_View_Helper'
$view->addHelperPath('/path/to/some/helpers',
        'My_View_Helper');
// Add /other/path/to/helpers… prefix 'Your_View_Helper'
$view->addHelperPath('/other/path/to/helpers',
        'Your_View_Helper');

// When you call $this->helperName(),Zend_View will look
// first for "/other/path/to/helpers/HelperName.php" using
// class name "My_View_Helper_HelperName",
// then for "/path/to/some/helpers/HelperName" using class
// name "Your_View_Helper_HelperName", and
// finally for "Zend/View/Helper/HelperName.php" using
// class name "Zend_View_Helper_HelperName".
```

Filters

Can indicate a filter to use after rendering a script with `setFilter()` *or* `addFilter()` *or* the `filter` option to the constructor

Zend_Layout:

Zend_Layout implements a classic Two-Step View pattern, allowing developers to wrap application content within another view, usually the site template. Usually, this pattern has been called a Layout, so ZF has adopted this convention.

Among the goals of this component is:

- Automatic selection and rendering of layouts when used with the ZF MVC components
- Separate scope for layout-related variables and content
- Configuration of layout name, inflection, and path
- Disabling of layouts, changing scripts, etc., from within action controllers and view scripts
- Following script resolution rules as defined with ViewRenderer, but also the ability to override them
- Usage without ZF MVC components

Zend_Layout Sample Script:

```
<!DOCTYPE html
    PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>
    <meta http-equiv="Content-
Type" content="text/html; charset=utf-8" />
    <title>My Site</title>
</head>
<body>
<?php
    // fetch 'content' key using layout helper:
    echo $this->layout()->content;
    // fetch 'foo' key using placeholder helper:
    echo $this->placeholder('Zend_Layout')->foo;
    // fetch layout object and retrieve various keys from it:
    $layout = $this->layout();
    echo $layout->bar;
    echo $layout->baz;
?>
</body>
</html>
```

Zend_Layout Utilizing the Zend Framework MVC:

Zend_Layout takes advantage of the rich feature set provided by Zend_Controller (Example: Front Controller plugins, Action Controller Helpers) and Zend_View (Example: Helpers) as discussed in previous sections.

Initialize Zend_Layout for use with MVC (sample code examples follow)

`Zend_Layout::startMvc()`

- creates an instance of `Zend_Layout` with any optional configuration provided
- registers a front controller plugin that renders the layout with any application content once the dispatch loop is done
- registers an action helper to allow access to the layout object from the action controllers; can grab the layout instance from within a view script using the Layout View Helper at any time

```
<?php
// Put into the bootstrap file:
Zend_Layout::startMvc();
?>
```

Access Layout Instance as an Action Helper with the Action Controller

```
<?php
class FooController extends Zend_Controller_Action
{
    public function barAction()
    {
        // disable layouts for this action:
        $this->_helper->layout->disableLayout();
    }

    public function bazAction()
    {
        // use different layout script with this action:
        $this->_helper->layout->setLayout('foobaz');
    };
}
?>
```

Access Layout Object using the Layout View Helper

```
<?php $this->_layout()->setLayout('foo');
    // set alternate layout
?>
```

Fetch Layout Instance using getMvcInstance() Method

```
<?php
    // Returns Null if startMvc() has not been called
$layout = Zend_Layout::getMvcInstance();
?>
```

Zend_Layout Front Controller Plugin

- renders the layout
- retrieves all named segments from the Response Object
    - assigns them as layout variables
    - assigns the 'default' segment to the variable 'content' (which allows for access to application content and rendering within view scripts)
    - especially useful in combination with ActionStack Helper and Plugin (sets up stack of actions to loop through, creating widgetized pages)

```
/* In this code example, FooController::indexAction()
   renders content to the default response segment, and
   then forwards it to NavController::menuAction(), which
   renders the content to the 'nav' response segment…
   The final forward is to CommentController::fetchAtion(),
   to fetch comments and render them to the default
   response segment via an 'append'. The view script can
   render each separately. */
<body>
    <!-- renders /nav/menu -->
    <div id="nav"><?= $this->layout()->nav ?></div>

    <!-- renders /foo/index + /comment/fetch -->
    <div id="content"><?= $this->layout()->content ?></div>
</body>
```

Zend_Layout as a Standalone Component (without ZF MVC):

Although lacking many key features when used without the MVC structure `Zend_Layout` still retains two primary benefits:

- scoping of layout variables
- isolation of layout view script from other view scripts

Utilizing Zend_Layout as Standalone Component

```
<?php
$layout = new Zend_Layout();

// Set a layout script path:
$layout->setLayoutPath('/path/to/layouts');

// set some variables:
$layout->content = $content;
$layout->nav     = $nav;

// choose a different layout script:
$layout->setLayout('foo');

// render final layout
echo $layout->render();
?>
```

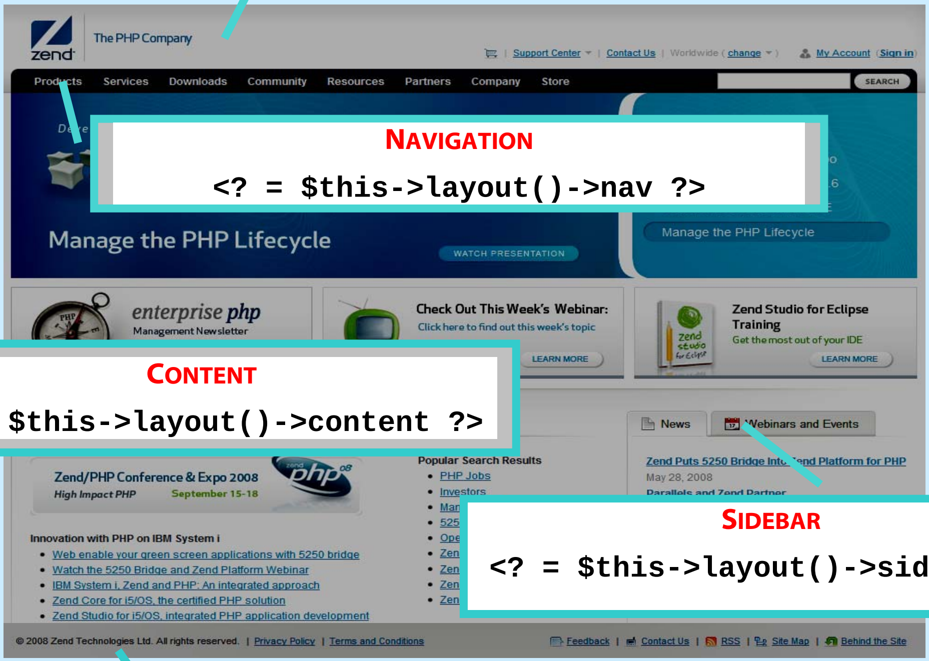
Zend_Layout as a Standalone Component (without ZF MVC):

Sample Layout Representation (order of elements will vary depending upon chosen CSS)

```
<?=$this->docType('XHTML1_STRICT') ?>
<html>
  <head>
       <? = $this->headTitle() ?>
       <? = $this->headScript() ?>
       <? = $this->headStylesheet() ?>
  </head>
  <body>
```

HEADER

```
<? = $this->partial('header.phtml')?>
```


The PHP Company
Support Center | Contact Us | Worldwide (change) | My Account (Sign in)
Products Services Downloads Community Resources Partners Company Store
SEARCH
Manage the PHP Lifecycle
WATCH PRESENTATION
enterprise php
Management Newsletter
Check Out This Week's Webinar:
Click here to find out this week's topic
LEARN MORE
Zend Studio for Eclipse Training
Get the most out of your IDE
LEARN MORE
News
Webinars and Events
Zend/PHP Conference & Expo 2008
High Impact PHP
September 15-18
Popular Search Results
PHP Jobs
Investors
Zend Puts 5250 Bridge Into Zend Platform for PHP
May 28, 2008
Parallels and Zend Partner
Innovation with PHP on IBM System i
Web enable your green screen applications with 5250 bridge
Watch the 5250 Bridge and Zend Platform Webinar
IBM System i, Zend and PHP: An integrated approach
Zend Core for i5/OS, the certified PHP solution
Zend Studio for i5/OS, integrated PHP application development
© 2008 Zend Technologies Ltd. All rights reserved. | Privacy Policy | Terms and Conditions
Feedback | Contact Us | RSS | Site Map | Behind the Site

NAVIGATION

```
<? = $this->layout()->nav ?>
```

CONTENT

```
<? = $this->layout()->content ?>
```

SIDEBAR

```
<? = $this->layout()->sidebar ?>
```

FOOTER

```
<? = $this->partial('footer.phtml')?>
```

```
  </body>
</html>
```

TEST YOUR KNOWLEDGE : QUESTIONS

1 Front Controller plugins and Action Helpers share what common feature?

a. pre- and postDispatch() hooks

b. Introspection of the action controller

c. Ability to change routing behavior

d. Registration with a common broker

2 Which one of the following will NOT assign the values to the view object?

a.
```
<code>
$view->foo = 'bar';
$view->bar = 'baz';
</code>
```

b.
```
<code>
$view->assign(array(
    'foo' => 'bar',
    'bar' => 'baz',
));
</code>
```

c.
```
<code>
$view->assign('foo', 'bar');
$view->assign('bar', 'baz');
</code>
```

d.
```
<code>
$view->assign(
        array('foo' => 'bar'),
        array('bar' => 'baz')
);
</code>
```

TEST YOUR KNOWLEDGE : ANSWERS

★ = CORRECT

1

Front Controller plugins and Action Helpers share what common feature?

- ★ a. pre- and postDispatch() hooks
- b. Introspection of the action controller
- c. Ability to change routing behavior
- d. Registration with a common broker

2

Which one of the following will NOT assign the values to the view object?

a.
```
<code>
$view->foo = 'bar';
$view->bar = 'baz';
</code>
```

b.
```
<code>
$view->assign(array(
    'foo' => 'bar',
    'bar' => 'baz',
));
</code>
```

c.
```
<code>
$view->assign('foo', 'bar');
$view->assign('bar', 'baz');
</code>
```

★ d.
```
<code>
$view->assign(
       array('foo' => 'bar'),
       array('bar' => 'baz')
);
</code>
```

CERTIFICATION TOPIC : PERFORMANCE

Zend_Cache

Use / Purpose · *Drivers* · *Setting Options*

Auto-serialization · *Functions*

Zend_Memory

Modes · *Storage* · *Retrieval*

Implications of File

Zend_Loader

Performance Configuration

Zend_Config

Optimization

Opcode Cache ●··● *Profiling*

Zend Framework: Performance

For the exam, here’s what you should know already …

You should be able to explain how caching works and the benefits it provides.

You should know what kinds of frontends and backends Zend Framework employs, and the various caching options available.

You should be familiar with opcode caching and profiling.

You should know how to utilize the factory function for Cache and Memory.

You should know how the Memory function works within the framework, for storage and retrieval.

You should understand the trade-off relationship between performance and the Loader.

ZEND_CACHE

Zend_Cache provides a generic way of caching data. Caching in Zend Framework is operated by frontends while cache records are stored through backend adapters (File, Sqlite, Memcache...) through a flexible system of IDs and tags. These IDs and tags allow for easier deletion by type.

Zend_Cache_Core, the core of the module, is generic, flexible and configurable. Cache frontends that extend Zend_Cache_Core include: Output, File, Function and Class. The addition of a frontend turns on serialization, so that any type of data can be cached. This helps to boost performance. For example, the results of a resource-intensive query can be cached and retrieved later (unserialized) without requiring a connection to the database and running the query again.

Zend_Cache::factory() instantiates correct objects and ties them together.

Example: Using a Core frontend together with a File backend

```
<?php
require_once 'Zend/Cache.php';

$frontendOptions = array(
   'lifetime' => 7200, // cache lifetime of 2 hours
   'automatic_serialization' => true
);

$backendOptions = array(
    'cache_dir' => './tmp/' // Directory to put the cache files
);

// getting a Zend_Cache_Core object
$cache = Zend_Cache::factory('Core', 'File', $frontendOptions,
         $backendOptions);
```

A core strategy for caching is to demarcate the sections in which to cache output by adding some conditional logic, encapsulating the section within start() and end() methods. Also, be sure to make the cache identifier, which gets passed to save() and start(), unique; otherwise, unrelated cache records may wipe each other out or even be displayed in place of the other.

Multiple-Word Frontends and Backends:
Frontends and backends that are named using multiple words, such as 'ZendPlatform' should be separated by using a word separator, such as a space (' '), hyphen ('-'), or period ('.').

ZEND_CACHE

Theory Behind Caching

There are three key concepts behind caching: 1) using a **unique identifier** for manipulating records; 2) issuing a **lifetime directive** that defines the 'age' of the resource; 3) employ **conditional execution**, which means the cached resources can be selectively utilized, boosting performance.

Factory Method

Always use `Zend_Cache::factory()` to get frontend instances. Instantiating frontends and backends yourself will not work as expected, as illustrated in this code:

```
<?php
// Load the Zend_Cache factory
require 'Zend/Cache.php';

// Select the backend (for example 'File' or 'Sqlite'...)
$backendName = '[...]';

// Select the frontend (for example 'Core', 'Output', 'Page'...)
$frontendName = '[...]';

// Set an array of options for the chosen frontend
$frontendOptions = array([...]);

// Set an array of options for the chosen backend
$backendOptions = array([...]);

// Create the instance, with two optional arguments (last two)
$cache = Zend_Cache::factory($frontendName, $backendName,
         $frontendOptions, $backendOptions);
```

Cleaning the Cache

Use the `remove()` method to remove or invalidate a particular cache ID. To remove multiple IDs, use the clean() method. Example: Remove all Cache Records

```
<?php
// clean all records
$cache->clean(Zend_Cache::CLEANING_MODE_ALL);


// clean only outdated records
$cache->clean(Zend_Cache::CLEANING_MODE_OLD);
```

ZEND_CACHE

Frontends

Zend_Cache_Core

`Zend_Cache_Core` is the primary frontend, the core of the module. It is generic and extended by other classes. All frontends inherit from `Zend_Cache_Core` so its methods and options are also available in other frontends.

Available Frontend Options (*for detailed information, see the Programmers Guide*)

- `caching`
- `cache_id_prefix`
- `lifetime`
- `logging`
- `write_control`
- `automatic_serialization`
- `automatic_cleaning_factor`
- `ignore_user_abort`

Zend_Cache_Frontend_

`Zend_Cache_Frontend_Output` is a frontend that utilizes output buffering in PHP to capture all output between its `start()` and `end()` methods.

`Zend_Cache_Frontend_Function` catches the results of function calls, and has a single main method, `call()`, which takes a function name and parameters for the call in an array. Using the `call()` function is the same as using `call_user_func_array()` in PHP. This component caches both the return value of the function and its internal output.

Available Function Frontend Options (*for detailed information, see the Programmers Guide*)

- `cache_by_defaulting`
- `cached_functions`
- `logging`
- `non-cached_functions`

`Zend_Cache_Frontend_Class` differs from `Zend_Cache_Frontend_Function` in that it allows caching of object and static method calls.

`Zend_Cache_Frontend_File` is a frontend driven by the modification time of a 'master file' and is typically used to help solve configuration and template issues.

`Zend_Cache_Frontend_Page` is similar to `Zend_Cache_Frontend_Output` but is designed for a whole page, and cannot be used for only a single block.

ZEND_CACHE

BACKENDS

Zend_Cache_Backend_File
`Zend_Cache_Backend_File` stores cached records into files in a chosen directory.

Available Backend Options (*for detailed information, see the Reference Guide*)

- `cache_dir`
- `file_locking`
- `read_control`
- `read_control_type`
- `file_name_prefix`
- `cache_file_umask`
- `hashed_directory_level`
- `hashed_directory_umask`
- `metadatas_array_max_size`

Zend_Cache_Backend_Sqlite
`Zend_Cache_Backend_Sqlite` stores cache records into a SQLite database.

- `cache_db_complete_path` *`required`*
- `automatic_vacuum_factor`

Zend_Cache_Backend_Memcached
`Zend_Cache_Backend_Memcached` stores cached records into a 'memcached' server, where memcached is a high-performance, distributed memory object caching system. Using this backend requires a memcached daemon and the memcache PECL extension.

- `servers`
- `compression`

Zend_Cache_Backend_Apc
`Zend_Cache_Backend_Apc` stores cached records in shared memory through the Alternative PHP Cache (APC) extension (*required*).

Zend_Cache_Backend_ZendPlatform
`Zend_Cache_Backend_ZendPlatform` uses the Platform content caching API.

ZEND_MEMORY

Zend_Memory is used to manage data within a memory-limited environment. The Factory instantiates the memory manager object, which regulates memory through the use of cache and the swapping and loading of memory objects as required. Note: the memory manager uses the Zend_Cache backends for storage.

```
<?php
require_once 'Zend/Memory.php';

$backendOptions = array(
    'cache_dir' => './tmp/' // Directory where to put the swapped
                               memory blocks
);
$memoryManager = Zend_Memory::factory('File', $backendOptions);

$loadedFiles = array();

for ($count = 0; $count < 10000; $count++) {
    $f = fopen($fileNames[$count], 'rb');
    $data = fread($f, filesize($fileNames[$count]));
    $fclose($f);

    $loadedFiles[] = $memoryManager->create($data);
}
echo $loadedFiles[$index1]->value;

$loadedFiles[$index2]->value = $newValue;

$loadedFiles[$index3]->value[$charIndex] = '_';
```

ZEND_MEMORY

Theory Behind Zend Memory

There are four key concepts behind `Zend_Memory`: 1) Memory Manager, which generates memory objects upon request of the application, and returns the objects wrapped into a memory container object; 2) Memory Container, which contains either a virtual or actual `value` attribute, a string that contains the data value specified when the memory object is created; 3) Locked Memory , which contains data that becomes effectively 'locked' and is never swapped to the cache backend; 4) Movable Memory, basically the opposite of Locked Memory, containing movable objects that are transparently swapped and loaded, to and from, the backend as needed.

Memory Manager and Factory

Use the `Zend_Memory::factory($backendName [, $backendOptions])` method to create a new Memory Manager. As mentioned previously, the Memory Manager uses the `Zend_Cache` backends for storage. To prevent the Manager from swapping memory blocks, it is possible to use a special backend, `'none'`. This is used whenever memory is not limited or the overall size of the objects does not exceed the memory limit.

```php
<?php
$backendOptions = array(
    'cache_dir' => './tmp/' // Directory where to put the swapped
                               memory blocks
);

$memoryManager = Zend_Memory::factory('File', $backendOptions);
```

Create Movable Objects

Use the `Zend_Memory_Manager::create([$data])` method

```php
<?php
$memObject = $memoryManager->create($data);
```

Create Locked Objects

Use the `Zend_Memory_Manager::createLocked([$data])` method

```php
<?php
$memObject = $memoryManager->createLocked($data);
```

ZEND_MEMORY

Destroy Objects

Memory objects are automatically destroyed and removed from memory when they go out of scope.

Set Memory Limit

Use the `getMemoryLimit()` and `setMemoryLimit($newLimit)` methods to retrieve or set the memory limit setting. A negative value equates to 'no limit'.

```
<?php
$oldLimit = $memoryManager->getMemoryLimit();
// Get memory limit in bytes

$memoryManager->setMemoryLimit($newLimit);
// Set memory limit in bytes
```

Memory Container

Use the memory container (movable or locked) `'value'` property to operate with memory object data. An alternative way to access memory object data is to use the `getRef()` method. This method *must* be used for PHP versions before 5.2, and may have to be used in some other cases for performance reasons.

PERFORMANCE

Loading Files and Classes Dynamically

The `Zend_Loader` class includes methods to help you load files dynamically. It is best used when the file name to be loaded is variable (for example, based on user input). With fixed file or class names, there will be no advantage to using `Zend_Loader` over `require_once()`.

Configuration

Generally, developers will use one of the `Zend_Config` adapter classes, such as `Zend_Config_Ini` or `Zend_Config_Xml`, to execute the process, but it is important to note that configuration data available in a PHP array may simply be passed to the `Zend_Config` constructor in order to utilize a simple object-oriented interface. `Zend_Config_Ini` and `Zend_Config_Xml` are much slower than loading an array directly.

Opcode Caching

Opcode caching is an important boost to performance for applications written in an interpretive language like PHP. Instead of having to generate pages each time they are requested, the opcode caching mechanism preserves the generated code in cache so that it need only be generated a single time to service any number subsequent requests. This is not technically a part of ZF, but it is a best practice for production.

Profiling and Zend_Db_Profiler

Within ZF, database profiling is the only effective way to find the actual performance problems within an application. `Zend_Db_Profiler` allows for the profiling of queries, including information on which queries were processed by the adapter as well as elapsed time to run the queries, helping greatly in identifying bottlenecks.

TEST YOUR KNOWLEDGE : QUESTIONS

1

If $cache is an instance of Zend_Cache_Frontend_Function, which ONE of the following will cache this function call:

```
<php>
$result = multiply(5, 10);
</php>
```

```
a. $result = $cache->call('multiply', array(5, 10));

b. $result =  $cache->call('multiply', 5, 10);

c. $result =  $cache->multiply(array(5, 10));

d. $result =  $cache->multiply(5, 10);
```

2

Which ONE of the following will create a memory manager object?

```
a.  $memoryManager = new Zend_Memory('None');

b.  $memoryManager = new Zend_Memory_Backend_None();

c.  $memoryManager = Zend_Memory::factory('None');

d.  $memoryManager = Zend_Memory::getInstance('None');
```

TEST YOUR KNOWLEDGE : ANSWERS

★ = CORRECT

1

If $cache is an instance of Zend_Cache_Frontend_Function, which ONE of the following will cache this function call:

```
<php>
$result = multiply(5, 10);
</php>
```

★ a. `$result = $cache->call('multiply', array(5, 10));`

b. `$result =  $cache->call('multiply', 5, 10);`

c. `$result =  $cache->multiply(array(5, 10));`

d. `$result =  $cache->multiply(5, 10);`

2

Which ONE of the following will create a memory manager object?

a. `$memoryManager = new Zend_Memory('None');`

b. `$memoryManager = new Zend_Memory_Backend_None();`

★ c. `$memoryManager = Zend_Memory::factory('None');`

d. `$memoryManager = Zend_Memory::getInstance('None');`

CERTIFICATION TOPIC : SEARCH

Zend_Search_Lucene

Use / Purpose ●··● *Compatibility* ●··● *Storage Backend*

Search - Indexing

Atomic Items ●··● *Analyzers* ●··● *Documents*

Search - Querying

Language Concepts ●··● *Construct Methods* ●··● *Iteration*

Fetch Hit Properties ●··● *Pagination*

Search - Performance

Index Optimization ●··● *Batch Indexing* ●··● *Field Selectivity*

SEARCH
FOCUS

Zend Framework: Search

For the exam, here's what you should know already …

You should understand the purpose of a full-text search system within a web application, and the benefits that Zend_Search_Lucene, as the search component of ZF, provides. You should know the basics of using Zend_Search_Lucene, including compatibility and mode of operation.

You should be familiar with all basic aspects of indexing, parsing, documents and fields, including index optimization.

You should know query language concepts, object constructing methods, and how to work with results sets.

ZEND_SEARCH_LUCENE

`Zend_Search_Lucene` is a general purpose text search engine written entirely in PHP 5. Since it stores its index on the filesystem and does not require a database server, it can add search capabilities to almost any PHP-driven website. `Zend_Search_Lucene` supports the following features:

- Ranked searching - best results returned first
- Many powerful query types: phrase queries, wildcard queries, proximity queries, range queries and more
- Search by specific field (e.g., title, author, contents)

`Zend_Search_Lucene` was derived from the Apache Lucene project. For more information on Lucene, visit http://lucene.apache.org/java/docs/

File Formats

`Zend_Search_Lucene` index file formats are binary compatible with Java Lucene version 1.4 and greater.

Document and Field Objects

`Zend_Search_Lucene` operates with documents as atomic objects for indexing. A document is divided into named fields with searchable content and is represented by the `Zend_Search_Lucene_Document` class. Objects of this class contain instances of `Zend_Search_Lucene_Field` that represent the fields on the document.

It is important to note that *any* information can be added to the index. Application-specific information or metadata can be stored in the document fields, and later retrieved with the document during search. Each application must be set up to control the indexer, which means that data can be indexed from any source accessible to your application – a filesystem, database, HTML form, etc.

ZEND_SEARCH_LUCENE

Document and Field Objects (continued)

`Zend_Search_Lucene_Field` class provides several static methods to create fields with different characteristics:

```
<?php
$doc = new Zend_Search_Lucene_Document();
// Field not tokenized, but is indexed and stored within index.
// Stored fields can be retrieved from the index.
$doc->addField(Zend_Search_Lucene_Field::Keyword('doctype',
                                                 'autogenerated'));

// Field not tokenized nor indexed, but is stored in the index.
$doc->addField(Zend_Search_Lucene_Field::UnIndexed('created',
                                                    time()));

// Binary string value field not tokenized nor indexed,
// but is stored in the index.
$doc->addField(Zend_Search_Lucene_Field::Binary('icon',
                                                $iconData));

// Field is tokenized and indexed, and is stored in the index.
$doc->addField(Zend_Search_Lucene_Field::Text('annotate',
                                              'Doc annotate text'));

// Field is tokenized and indexed, but is not stored in the index.
$doc->addField(Zend_Search_Lucene_Field::UnStored('contents',
                                                  'Doc content'));
```

ZEND_SEARCH_LUCENE

Charset

`Zend_Search_Lucene` works with the UTF-8 charset internally. Index files store unicode data in Java's "modified UTF-8 encoding". `Zend_Search_Lucene` core completely supports this encoding except for "supplementary characters" - Unicode characters with codes greater than `0xFFFF`.

Actual input data encoding may be specified through the `Zend_Search_Lucene` API. In general, the data is automatically converted into UTF-8 encoding. However, some text transformations are made by text analyzers and the default text analyzer does not work with UTF-8 because of requirements for certain extensions to be loaded. In these cases, the date is translated into ASCII/TRANSLIT and then tokenized. Special UTF-8 analyzers, described later in the "Analyzer" section, may be used.

Storage

The abstract class `Zend_Search_Lucene_Storage_Directory` defines directory functionality.

The `Zend_Search_Lucene` constructor uses either a string or a `Zend_Search_Lucene_Storage_Directory` object as input.

The `Zend_Search_Lucene_Storage_Directory_Filesystem` class implements directory functionality for a file system. If a string is used as an input for the `Zend_Search_Lucene` constructor, then the index reader (`Zend_Search_Lucene` object) treats it as a file system path and instantiates the `Zend_Search_Lucene_Storage_Directory_Filesystem` object.

Directory implementations can be custom-defined by extending the `Zend_Search_Lucene_Storage_Directory` class.

ZEND_SEARCH_LUCENE

Building and Updating an Index

Index creation and updating capabilities are implemented within the Zend_Search_Lucene component, as well as the Java Lucene project – either can be used to create indexes that Zend_Search_Lucene can search. The same procedure is used to update an existing index. The only difference is that the open() method is called instead of the create() method. Ex: Indexing a file using the Zend_Search_Lucene indexing API

```
<?php
// Create index
$index = Zend_Search_Lucene::create('/data/my-index');
$doc = new Zend_Search_Lucene_Document();

// Store document URL to identify it in the search results
$doc->addField(Zend_Search_Lucene_Field::Text('url', $docUrl));

// Index document contents
$doc->addField(
 Zend_Search_Lucene_Field::UnStored(
  'contents',
  $docContent
 )
);

// Add document to the index
$index->addDocument($doc);
```

ZEND_SEARCH_LUCENE

Updating Documents

The Lucene index file format doesn't support updating documents. Documents should be removed and re-added to the index to effectively update them. `Zend_Search_Lucene::delete()` method operates with an internal index document id. It can be retrieved from a query hit by 'id' property:

```
<?php
$removePath = ...;
$docs = $index->find('path:' . $removePath);
foreach ($docs as $doc) {
    $index->delete($doc->id);
}
```

Index Optimization

A Lucene index consists of many segments - each segment is a completely independent set of data, which cannot be updated. A segment update needs full segment reorganization, and new documents are added to the index by creating new segments.

Increasing the number of segments reduces index quality, but index optimization restores it. Optimization essentially merges several segments into a new one. The process generates one new large segment and updates the segment list ('segments' file), but not the segments themselves.

A call to `Zend_Search_Lucene::optimize()` invokes full index optimization, merging all index segments into one new segment:

```
<?php
// Open existing index
$index = Zend_Search_Lucene::open('/data/my-index');

// Optimize index
$index->optimize();
```

Automatic index optimization is performed to keep indexes in a consistent state. It is an iterative process managed by several index options, which merges very small segments into larger ones, then merges these larger segments into even larger segments and so on.

ZEND_SEARCH_LUCENE

Searching Indices – Building Queries

There are two ways to search an index - the first method uses a query parser to construct a query from a string. The second is to programmatically create your own queries through the `Zend_Search_Lucene` API. Generally speaking, the query parser is designed for human-entered text, not for program-generated text (better suited to the API method). Untokenized fields are best added directly to queries and not through the query parser. If a field's values are generated programmatically by the application, then the query clauses for this field should also be constructed programmatically.

Analyzer

An analyzer, which is used by the query parser, is designed to convert human-entered text to terms. Program-generated values, like dates, keywords, etc., should be added with the query API.

In a query form, fields that are general text should use the query parser. All others, such as date ranges, keywords, etc., are better added directly through the query API. A field with a limited set of values that can be specified with a pull-down menu should not be added to a query string subsequently parsed; instead, it should be added as a `TermQuery` clause.

Boolean queries allow the programmer to logically combine two or more queries into new one, the best way to add additional criteria to a search defined by a query string.

Both ways use the same API method to search through the index:

```
<?php
require_once 'Zend/Search/Lucene.php';

$index = Zend_Search_Lucene::open('/data/my_index');

$index->find($query);
```

`Zend_Search_Lucene::find()` determines input type automatically and uses the query parser to construct an appropriate `Zend_Search_Lucene_Search_Query` object from an input of type string.

Note: the query parser uses the standard analyzer to tokenize separate parts of a query string; all transformations applied to indexed text are also applied to query strings.

The standard analyzer may transform the query string to lower case for case-insensitivity, remove stop-words, and stem among other modifications. The API method doesn't transform or filter input terms in any way, making it more suitable for computer generated or untokenized fields.

ZEND_SEARCH_LUCENE

Analyzer (continued)

Zend_Search_Lucene contains a set of UTF-8 compatible analyzers:
Zend_Search_Lucene_Analysis_Analyzer_Common_ +
Utf8 *or* Utf8Num *or* Utf8_CaseInsensitive *or* Utf8Num_CaseInsensitive

Search Results

The search result is an array of Zend_Search_Lucene_Search_QueryHit objects. Each of these has two properties: $hit->document is a document number within the index and $hit->score is a score of the hit in a search result. The results are ordered by score (descending from highest score).

The Zend_Search_Lucene_Search_QueryHit object also exposes each field of the Zend_Search_Lucene_Document found in the search as a property of the hit. Stored fields are always returned in UTF-8 encoding. Optionally, the original Zend_Search_Lucene_Document object can be returned from the Zend_Search_Lucene_Search_QueryHit. You can retrieve stored parts of the document by using the getDocument() method of the index object and then get them using getFieldValue(). Ex:

```php
<?php
require_once 'Zend/Search/Lucene.php';
$index = Zend_Search_Lucene::open('/data/my_index');
$hits = $index->find($query);
foreach ($hits as $hit) {
// return Zend_Search_Lucene_Document object for this hit
    echo $document = $hit->getDocument();
// return Zend_Search_Lucene_Field object
// from Zend_Search_Lucene_Document
    echo $document->getField('title');
// return the string value of Zend_Search_Lucene_Field object
    echo $document->getFieldValue('title');
// same as getFieldValue()
    echo $document->title;
}
```

ZEND_SEARCH_LUCENE

Search Results (continued)

The fields available from the `Zend_Search_Lucene_Document` object are determined at the time of indexing. The document fields are either indexed, or index and stored, in the document by the indexing application (Example: `LuceneIndexCreation.jar`). Note that the document identity ('path' in our example) is also stored in the index and must be retrieved from it.

Short Document Field Access

Short syntax automatically retrieves a document from an index (although it needs additional time and increases memory usage). Short index is not suitable for `'id'` and `'score'` document fields (if they are present) because of special `'id'` and `'score'` hit properties name conflicts.

```
$hit->title
// identical to $hit->getDocument()->title and also
// identical to $hit->getDocument()->getFieldValue('title')
```

Short syntax is very important for pagination implementation. Whole result set ids and scores should be stored somewhere (for example, using `Zend_Cache`) without actual document retrieval (only `'id'` and `'score'` properties are accessed). Then, any required parts of ids or scores should be used to get documents using the `$index->getDocument($id)` method.

ZEND_SEARCH_LUCENE

Limit Result Set

Zend_Search_Lucene makes it possible to limit result set size using getResultSetLimit() and setResultSetLimit(). The component limits the result set by the "first N" results, before ordering these results by score.

As the score computation uses up essential search time, a reduction in the number of results scored represents a corresponding increase in Search performance.

Results Scoring

Zend_Search_Lucene uses the same scoring algorithms as Java Lucene. All hits in the search result are ordered by score by default. Hits with greater score come first, and documents having higher scores should match the query more precisely than documents having lower scores. A hit's score can be retrieved by accessing the score property of the hit:

```
<?php
$currentResultSetLimit = Zend_Search_Lucene::getResultSetLimit();

Zend_Search_Lucene::setResultSetLimit($newLimit);
```

Results Sorting

By default, the search results are ordered by score. This default can be overridden by setting a sort field (or a list of fields), sort type and sort order parameters.

```
<?php
$hits = $index->find($query);

foreach ($hits as $hit) {
    echo $hit->id;
    echo $hit->score;
}
```

TEST YOUR KNOWLEDGE : QUESTIONS

1

Which method should be used to retrieve total number of documents stored in the index (including deleted documents)?

a. `$index->countDocuments();`
b. `$index->numDoc();`
c. `$index->docCount();`
d. `$index->count();`

2

How to get full list of indexed fields from the index?

a. `$index->getFields(true);`
b. `$index->getFields(false);`
c. `$index->getFields(Zend_Search_Lucene_Field::INDEXED);`
d. `$index->getFields();`

TEST YOUR KNOWLEDGE : ANSWERS

★ = CORRECT

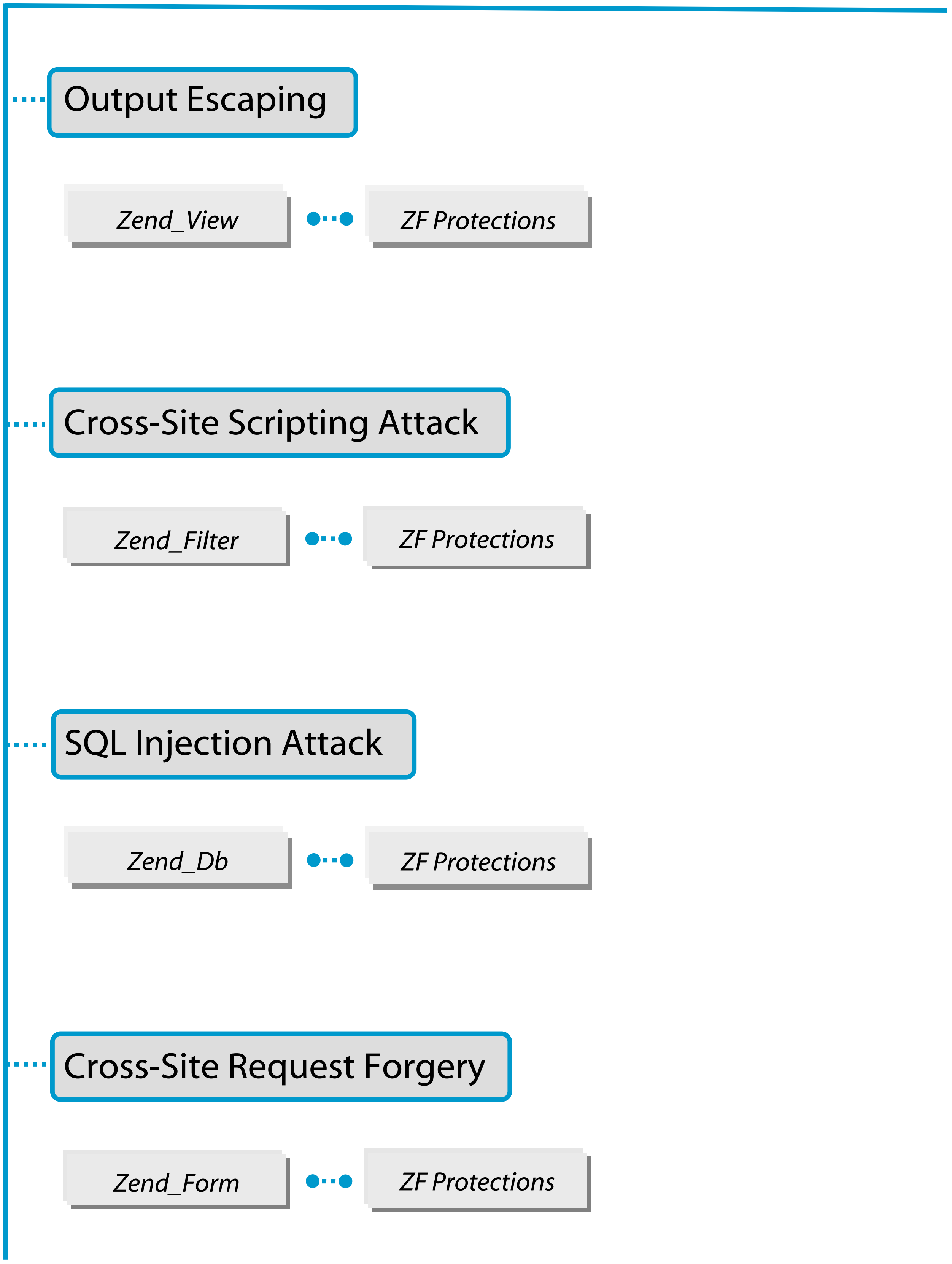
1

```
Which method should be used to retrieve total number of
documents stored in the index (including deleted documents)?
    a.  $index->countDocuments();
    b.  $index->numDoc();
    c.  $index->docCount();
★   d.  $index->count();
```

2

```
How to get full list of indexed fields from the index?
★  a.   $index->getFields(true);
   b.   $index->getFields(false);
   c.   $index->getFields(Zend_Search_Lucene_Field::INDEXED);
   d.   $index->getFields();
```

CERTIFICATION TOPIC : SECURITY



Secure Authentication

Zend_Auth

ZF Protections

Security Best Practices

ZF Protections

Zend Framework: Security

For the exam, here's what you should know already …

You will need to know the basics around security – the most common types of security attacks and breaches, and the most common measures to employ to help prevent or mitigate such attacks.

There are six sub-topic areas on which you will be tested:

- Output Escaping
- Cross-Site Scripting Attacks
- SQL Injection Attacks
- Cross-Site Request Forgery Attacks
- Secure Authentication
- Security Best Practices

OUTPUT ESCAPING

A critical function for view scripts to perform is to escape output properly, as it helps to fend off security attacks like cross-site scripting.

A best practice rule is always to escape variables when outputting them, unless you are using a function *or* method *or* helper that performs this step on its own.

```
<?php

// Bad View Script Practice:
echo $this->variable;

// Good View Script Practice:
echo $this->escape($this->variable);
```

- `Zend_View` uses the `escape()` method to performs output escaping
- By default, it uses the PHP function `htmlspecialchars()` for this process
- To escape using an alternate way, call the `setEscape()` method at the Controller level to instruct `Zend_View` on what escaping callback to use

```
<?php
// create a Zend_View instance
$view = new Zend_View();

// tell it to use htmlentities as the escaping callback
$view->setEscape('htmlentities');

// or tell it to use a static class method as the callback
$view->setEscape(array('SomeClass', 'methodName'));

// or even an instance method
$obj = new SomeClass();
$view->setEscape(array($obj, 'methodName'));

// and then render your view
echo $view->render(...);
```

CROSS-SITE SCRIPTING ATTACKS

Cross-site scripting attacks are an injection of HTML, CSS, or script code into a page. JavaScript is especially a threat; its primary cause is displaying data mis-interpreted by the browser.

Filters are commonly used to escape HTML elements, helping to avoid such attacks. If, for example, a form field is populated with untrustworthy data (and any data input from the outside is untrustworthy!), then this value should either not contain HTML (*removal*), or if it does, then have that HTML escaped.

- Calling the `filter()` method on any `Zend_Filter_*` object performs transformations upon input data (*Example for HTML & and " given below*)

```
<?php
require_once 'Zend/Filter/HtmlEntities.php';

$htmlEntities = new Zend_Filter_HtmlEntities();

echo $htmlEntities->filter('&'); // &amp;
echo $htmlEntities->filter('"'); // &quot;
```

SQL INJECTION ATTACKS

SQL injection attacks are those in which applications are manipulated to return data to a user that should not have the privilege to access/read such data.

This access is commonly accomplished by an attacker randomly attempting to exploit a flaw, known or not, in the code involving SQL queries which use PHP variables (requiring quotes). The developer must take into account that these variables may contain symbols that will result in incorrect SQL code – for example, the use of a quote in a person's name. The attack is accomplished by the attacker detecting such flaws, and manipulating them by, for example, specifying a value for a PHP variable through the use of an HTTP parameter or other similar mechanism.

Where is the error and consequent security flaw in this code?

```
$name = $_GET['Name'];
//$_GET['Name'] = "O'Reilly";
$sql = "SELECT * FROM bugs WHERE reported_by = '$name'";

echo $sql;
// SELECT * FROM bugs WHERE reported_by = 'O'Reilly'
```

Zend_Db has some built-in features that help to mitigate such attacks, although there is no substitute for good programming.

When inserting data from an array, the values are inserted as parameters by default, reducing the risk of some types of security issues. Consequently, you do not need to apply escaping or quoting to values in a data array.

CROSS-SITE REQUEST FORGERY ATTACKS

Cross-Site Request Forgery (XSRF) attacks employ an almost opposite approach to Cross-Site Scripting (XSS). XSRF attacks disguise themselves as a trusted user to attack a web site, whereas XSS attacks disguise themselves as a trusted web site to attack a user.

To protect against XSRF attacks originating from third party sites, you have options …

> First, prior to doing any kind of "dangerous" activity, such as changing
> a password or buying a car, require the user to re-authenticate themselves so there is a forced user interaction with the website (Ex: using `Zend_Auth`). By disallowing automated page and form submissions, the window that an attacker has to initiate the attack is severely limited.
>
> Another option available is the use of hashed identifiers in submitted forms (typically the most dangerous activity, like changing a password) can reduce the damage of a XSRF attack. Once a unique hash has been used to service a request, it is subsequently invalidated, and any request using that ID is not honored.
> `Zend_Form_Element_Hash`, used in conjunction with `Zend_Form::isValid()` helps to automate the hashing/validation mechanism.
>
> Filters (such as `Zend_Filter`) are commonly used to escape HTML elements. If, for example, a form field is populated with untrustworthy data (and any data input from the outside is untrustworthy!), then this value should either not contain HTML (*removal*), or if it does, then have that HTML escaped.
>
> (Note: there may be times when HTML output needs to be displayed – for example, when using a JavaScript-based HTML editor; in these cases, you would need to build either a validator or filter that employs a whitelist approach.

SECURE AUTHENTICATION

Authentication is the process of verifying a user's identity against some set of pre-established criteria.

`Zend_Auth` provides an API for conducting authentication, along with adapters designed for the most common uses.

Here are some things to keep in mind (*see the* Authorization *section for more detail*):

- `Zend_Auth` implements the Singleton pattern through its static `getInstance`() method.
  - Singleton pattern means only one instance of the class is available at any one time
  - The new operator and the clone keyword will not work with this class… use `Zend_Auth::getInstance()` instead
  - Use of Zend_Auth is not a substitute for proper encryption of communications
- The adapters authenticate against a particular service, like LDAP, RDBMS, etc.; while their behavior and options will vary, they share some common actions:
  - accept authenticating credentials
  - perform queries against the service
  - return results

SECURITY BEST PRACTICES

There are three golden rules to follow when providing security to web applications and their environments:

1. Always validate your input
2. Always filter your output
3. Never trust your users

The Zend Framework features (`Zend_Filter` and `Zend_Filter_Input`, `Zend_Validate`, `Zend_Auth`, etc.) that map to these rules have been covered individually in context with the types of attacks. Additional information can be found within other sections of this guide, such as Filtering and Validation, Authorization and Authentication.

TEST YOUR KNOWLEDGE : QUESTIONS

1

When using Zend_View with unescaped data, which of the following view script calls would escape your data, $data:

a. $this->filter($data)

b. $this->escape($data)

c. $this->htmlEntities($data)

d. $data->escape()

2

Zend_Auth will, regardless of the adapter used to process identities and credentials, will encrypt the information sent from the browser to the application using it.

a. TRUE

b. FALSE

TEST YOUR KNOWLEDGE : ANSWERS

★ = CORRECT

1

When using Zend_View with unescaped data, which of the following view script calls would escape your data, $data:

a. `$this->filter($data)`

★ b. `$this->escape($data)`

c. `$this->htmlEntities($data)`

d. `$data->escape()`

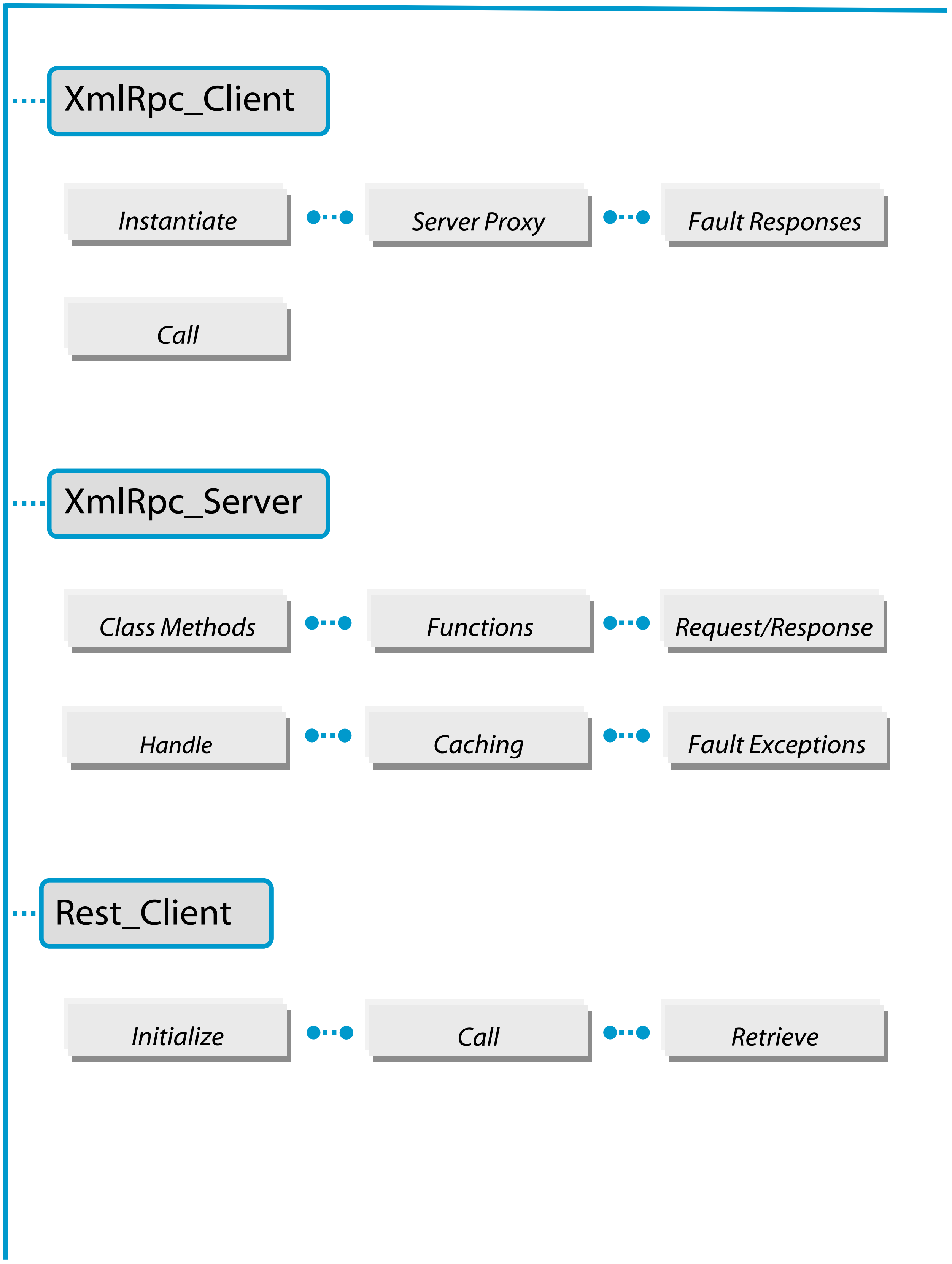
2

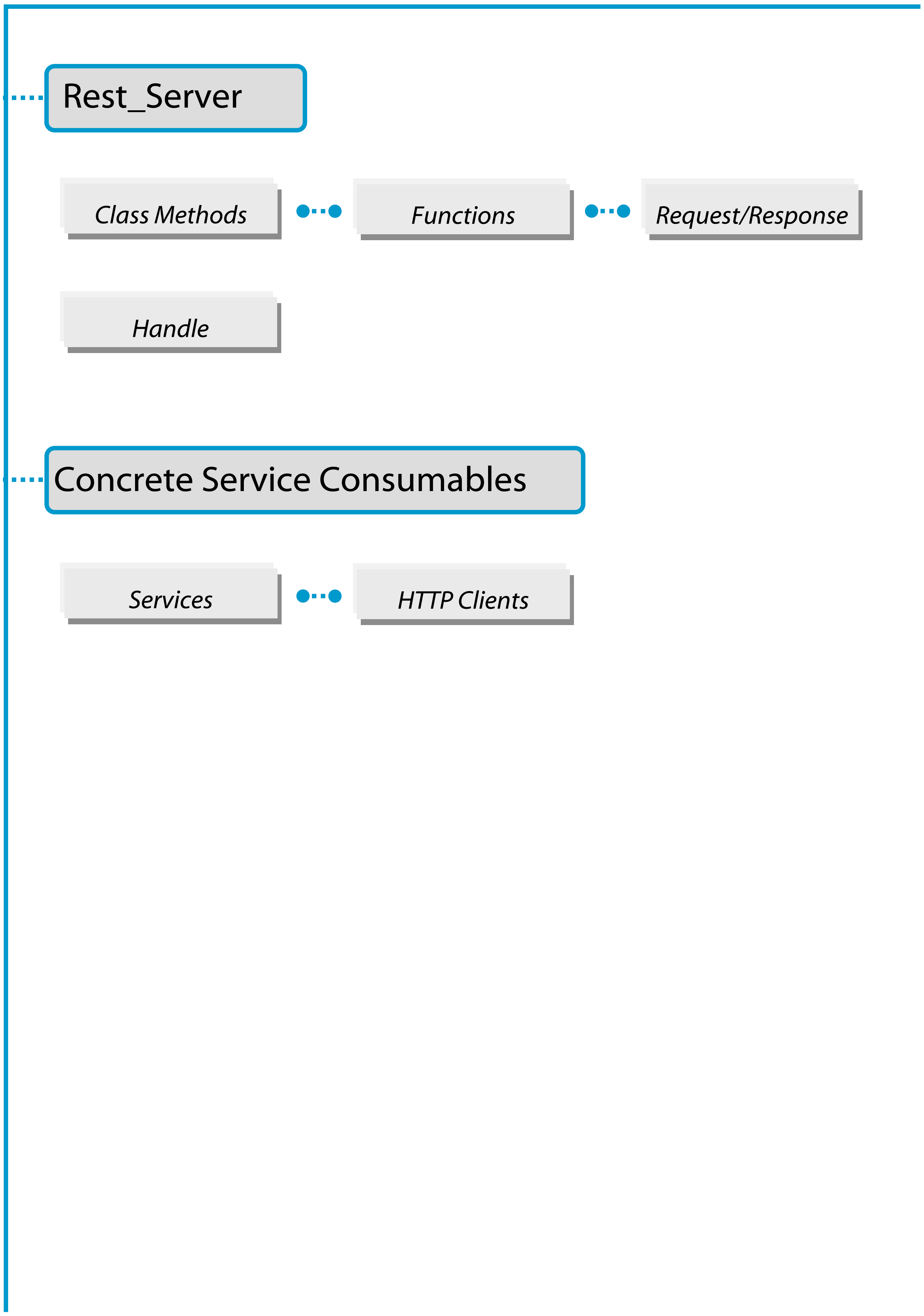
Zend_Auth will, regardless of the adapter used to process identities and credentials, will encrypt the information sent from the browser to the application using it.

a. TRUE

★ b. FALSE

CERTIFICATION TOPIC : WEB SERVICES




Rest_Server
Class Methods
Functions
Request/Response
Handle
Concrete Service Consumables
Services
HTTP Clients

Zend Framework: Web Services

For the exam, here's what you should know already …

You should be able to explain the function of the XmlRpc Clients and Servers, and provide code for related functions, methods, and classes.

You should know how to specify exceptions that may be used as fault responses.

You should be able to work with REST clients and services.

You should know the purpose of consumable services, and which are available to Zend Framework.

ZEND_XMLRPC_CLIENT

`Zend_XmlRpc_Client` provides support for consuming remote XML-RPC services as a client in its package. Its major features include automatic type conversion between PHP and XML-RPC, a server proxy object, and access to server introspection capabilities.

Method Calls

The constructor of `Zend_XmlRpc_Client` receives the URL of the remote XML-RPC server endpoint as its first parameter. The new instance returned can call any number of remote methods at that endpoint, by instantiating it and then using the `call()` instance method.

```php
<?php
require_once 'Zend/XmlRpc/Client.php';
$client = new Zend_XmlRpc_Client('http://www.zend.com/xmlrpc');
echo $client->call('test.sayHello');   // hello
```

The XML-RPC value returned from the remote method call will be automatically unmarshaled and cast to the equivalent PHP native type. In the example above, a PHP `string` is returned and immediately ready for use. The first parameter of the `call()` method receives the name of the remote method to call. Any required parameters can be sent by supplying a second, optional parameter to `call()` with an `array` of values to pass to the remote method.

```php
<?php
require_once 'Zend/XmlRpc/Client.php';
$client = new Zend_XmlRpc_Client('http://www.zend.com/xmlrpc');

$arg1 = 1.1;
$arg2 = 'foo';
$result = $client->call('test.sayHello', array($arg1, $arg2));
// $result is a native PHP type
```

If no parameters are required, this option may either be left out or an empty `array()` passed to it. Parameters can contain native PHP types, `Zend_XmlRpc_Value` objects, or a mix. The `call()` method will automatically convert the XML-RPC response and return its PHP native type. `getLastResponse()` makes a `Zend_XmlRpc_Response` object available for the return value.

Request to Response

The `call()` instance method of `Zend_XmlRpc_Client` builds a request object (`Zend_XmlRpc_Request`) and sends it to another method, `doRequest()`. This returns a response object (`Zend_XmlRpc_Response`). `doRequest()` can also be directly used.

ZEND_XMLRPC_CLIENT

Server Proxy Object:

Another way to call remote methods with the XML-RPC client is to use the server proxy, a PHP object that proxies a remote XML-RPC namespace, making it work as close to a native PHP object as possible. To instantiate a server proxy, call the `getProxy()` instance method of `Zend_XmlRpc_Client`, which returns an instance of `Zend_XmlRpc_Client_ServerProxy`. Any method call on the server proxy object will be forwarded to the remote, and parameters may be passed like any other PHP method.

```
<?php
require_once 'Zend/XmlRpc/Client.php';

$client = new Zend_XmlRpc_Client('http://www.zend.com/xmlrpc');

// Proxy the default namespace
$server = $client->getProxy();

//test.Hello(1,2)returns "hello"
$hello = $server->test->sayHello(1, 2);
```

The `getProxy()` method receives an optional argument specifying which namespace of the remote server to proxy. If it does not receive a namespace, the default namespace will be proxied.

Error Handling – HTTP Errors:

If an HTTP error occurs, such as the remote HTTP server returns a "`404 Not Found`", a `Zend_XmlRpc_Client_HttpException` will be thrown. Ex:

```
<?php
require_once 'Zend/XmlRpc/Client.php';

$client = new Zend_XmlRpc_Client('http://foo/404');

try {
    $client->call('bar', array($arg1, $arg2));

} catch (Zend_XmlRpc_Client_HttpException $e) {

    // $e->getCode() returns 404
    // $e->getMessage() returns "Not Found"
}
```

ZEND_XMLRPC_CLIENT

Error Handling – XML-RPC Faults:

An XML-RPC fault is analogous to a PHP exception. It is a special type returned from an XML-RPC method call that has both an error code and an error message. XML-RPC faults are handled differently depending on the context of how the `Zend_XmlRpc_Client` is used. When the `call()` method or the server proxy object is used, an XML-RPC fault will result in a `Zend_XmlRpc_Client_FaultException` being thrown. The code and message of the exception will map directly to their respective values in the original XML-RPC fault response.

```
<?php
require_once 'Zend/XmlRpc/Client.php';
$client = new Zend_XmlRpc_Client('http://www.zend.com/xmlrpc');

try {
    $client->call('badMethod');
} catch (Zend_XmlRpc_Client_FaultException $e) {

    // $e->getCode() returns 1
   Zend_Db contains a factory() method by which you may
instantiate a database adapter object
```

If `call()` is used to make the request, `Zend_XmlRpc_Client_FaultException` will be thrown on fault. A `Zend_XmlRpc_Response` object containing the fault will also be available by calling `getLastResponse()`.

If `doRequest()` is used to make the request, it will not throw the exception. Instead, it will return a `Zend_XmlRpc_Response` object containing the fault, which can be checked with the `isFault()` instance method of `Zend_XmlRpc_Response`.

When using the ServerProxy, faults are thrown as `Zend_XmlRpc_Client_FaultException` -- just as when using `call()`.

ZEND_XMLRPC_SERVER

Basic Usage:

Zend_XmlRpc_Server is intended as a fully-featured XML-RPC server, which implements all system methods, including the system.multicall() method, allowing for the 'boxcarring' of requests. Example of Zend_XmlRpc_Server 's most basic use:

```
<?php
require_once 'Zend/XmlRpc/Server.php';
require_once 'My/Service/Class.php';

$server = new Zend_XmlRpc_Server();
$server->setClass('My_Service_Class');
echo $server->handle();

;
```

Server Structure:

Zend_XmlRpc_Server is composed of a variety of components: he server itself, as well as request, response, and fault objects. To bootstrap Zend_XmlRpc_Server, attach one or more classes or functions to the server, via setClass() and addFunction(). Then, either pass a Zend_XmlRpc_Request object to Zend_XmlRpc_Server::handle(), or it will instantiate a Zend_XmlRpc_Request_Http object if none is provided -- thus grabbing the request from php://input.

Zend_XmlRpc_Server::handle() then attempts to dispatch to the appropriate handler based on the method requested, and returns either a Zend_XmlRpc_Response-based object or a Zend_XmlRpc_Server_Fault object. These objects both have __toString() methods that create valid XML-RPC XML responses, allowing them to be directly echoed.

ZEND_XMLRPC_SERVER

When attaching items to a server, functions and class methods must have full docblocks with, minimally, the parameter and return type annotations. Without this information, the servers will not work.

Attaching a Function:

This example illustrates the relatively simple process of attaching a function as a dispatchable XML-RPC method, also handling incoming calls.

```
<?php
require_once 'Zend/XmlRpc/Server.php';

/**
 * Return the MD5 sum of a value
 *
 * @param string $value Value to md5sum
 * @return string MD5 sum of value
 */
function md5Value($value)
{
    return md5($value);
}

$server = new Zend_XmlRpc_Server();
$server->addFunction('md5Value');
echo $server->handle();
```

Attaching a Class:

The code example below attaches a class' public methods as dispatchable XML-RPC methods.

```
<?php
require_once 'Zend/XmlRpc/Server.php';
require_once 'Services/Comb.php';

$server = new Zend_XmlRpc_Server();
$server->setClass('Services_Comb');
echo $server->handle();
```

Custom Request Objects:

Most of the time, you'll simply use the default request type included with `Zend_XmlRpc_Server`, `Zend_XmlRpc_Request_Http`. However, there may be times when you need XML-RPC to be available via the CLI, a GUI, or other environment, or want to log incoming requests. To do so, you may create a custom request object that extends `Zend_XmlRpc_Request`. The most important thing to remember is to ensure that the getMethod() and getParams() methods are implemented so that the XML-RPC server can retrieve that information in order to dispatch the request.

ZEND_XMLRPC_SERVER

Custom Responses:

Similar to request objects, `Zend_XmlRpc_Server` can return custom response objects; by default, a `Zend_XmlRpc_Response_Http` object is returned, which sends an appropriate Content-Type HTTP header for use with XML-RPC.

Possible uses of a custom object would be to log responses, or to send responses back to `STDOUT`. Be sure to use `Zend_XmlRpc_Server::setResponseClass()` prior to calling `handle()`.

Handling Exceptions:

`Zend_XmlRpc_Server` catches exceptions generated by a dispatched method, and generates an XML-RPC fault response when such an exception is caught.

By default, however, the exception messages and codes are not used in a fault response. This is an intentional decision to protect your code; exceptions expose information about the code or environment that can create a security risk.

Exception classes can be whitelisted to be used as fault responses. Simply utilize `Zend_XmlRpc_Server_Fault::attachFaultException()` to pass an exception class to whitelist.

```
<?php
Zend_XmlRpc_Server_Fault::attachFaultException('My Exception');
```

If you utilize an exception class that your other project exceptions inherit, a whole family of exceptions can be whitelisted. `Zend_XmlRpc_Server_Exceptions` are always whitelisted, for reporting specific internal errors (undefined methods, etc.).

Any exception not specifically whitelisted will generate a fault response with a code of `'404'` and a message of `'Unknown error'`.

Caching Server Definitions Between Requests:

Attaching many classes to an XML-RPC server instance can tie up resources; each class must introspect using the Reflection API (via `Zend_Server_Reflection`), which in turn generates a list of all possible method signatures to provide to the server class. To offset this performance hit, `Zend_XmlRpc_Server_Cache` can be used to cache the server definition between requests. When combined with `__autoload()`, this can greatly increase performance.

ZEND_XMLRPC_SERVER

Caching Server Definitions Between Requests

Attaching many classes to an XML-RPC server instance can tie up resources; each class must introspect using the Reflection API (via `Zend_Server_Reflection`), which in turn generates a list of all possible method signatures to provide to the server class. To offset this performance hit, `Zend_XmlRpc_Server_Cache` can be used to cache the server definition between requests. When combined with `__autoload()`, this can greatly increase performance. A sample use case follows:

```
function __autoload($class)
{
    Zend_Loader::loadClass($class);
}

$cacheFile = dirname(__FILE__) . '/xmlrpc.cache';
$server = new Zend_XmlRpc_Server();

if (!Zend_XmlRpc_Server_Cache::get($cacheFile, $server)) {
    require_once 'My/Services/Glue.php';
    require_once 'My/Services/Paste.php';
    require_once 'My/Services/Tape.php';

// glue. namespace
    $server->setClass('My_Services_Glue', 'glue');
// paste. namespace
    $server->setClass('My_Services_Paste', 'paste');
// tape. namespace
    $server->setClass('My_Services_Tape', 'tape');

    Zend_XmlRpc_Server_Cache::save($cacheFile, $server);
}

echo $server->handle();
```

The above example attempts to retrieve a server definition from `xmlrpc`.cache in the same directory as the script. If unsuccessful, it loads the service classes it needs, attaches them to the server instance, and then attempts to create a new cache file with the server definition.

ZEND_REST_CLIENT

REST Web Services use service-specific XML formats; consequently, the manner for accessing a REST web service is different for each service. REST web services typically use URL parameters (GET data) or path information for requesting data and POST data for sending data.

The Zend Framework provides both Client and Server capabilities that, when used together, allow for a much more "local" interface experience via virtual object property access. The Server component automatically exposes functions and classes using a meaningful and simple XML format. When accessing these services using the Client, the return data can be easily retrieved from the remote call.

Using the `Zend_Rest_Client` is very similar to using `SoapClient` objects . Simply call the REST service procedures as `Zend_Rest_Client` methods and specify the service's full address in the `Zend_Rest_Client` constructor.

```php
<?php
/**
 * Connect to framework.zend.com server and retrieve a greeting
 */
require_once 'Zend/Rest/Client.php';
$client = new Zend_Rest_Client('http://framework.zend.com/rest');
echo $client->sayHello('Dave', 'Day')->get(); // "Hello Dave,
Good Day"
```

`Zend_Rest_Client` attempts to make remote methods look as much like native methods as possible, only that method call must be followed with one of either `get()`, `post()`, `put()` or `delete()`. This call may be made via method chaining or in separate method calls.

```php
<?php
$client->sayHello('Dave', 'Day');
echo $client->get();
```

ZEND_REST_CLIENT

Responses

All requests made using `Zend_Rest_Client` return a `Zend_Rest_Client_Response` object. This object has many properties that make it easier to access the results. When the service is based on `Zend_Rest_Server`, `Zend_Rest_Client` can make several assumptions about the response, including response status (success or failure) and return type.

```
<?php
$result = $client->sayHello('Dave', 'Day')->get();

if ($result->isSuccess()) {
    echo $result; // "Hello Dave, Good Day"
}
```

In the example above, the request result as an object, used to call `isSuccess()`, and then using `__toString()`echoes the object to get the result.

`Zend_Rest_Client_Response` will allow any scalar value to be echoed. For complex types, you can use either array or object notation.

If the service is queried *not* using `Zend_Rest_Server`, the `Zend_Rest_Client_Response` object will behave more like a `SimpleXMLElement`. However, to make things easier, it will automatically query the XML using XPath if the property is not a direct descendant of the document root element. Additionally, if the property is accessed as a method, the PHP value for the object is returned, or an array of PHP value results.

ZEND_REST_SERVER

Zend_Rest_Server is intended as a fully-featured REST server. When attaching items to a server, functions and class methods must have full docblocks with, minimally, the parameter and return type annotations. Without this information, the servers will not work.

Zend_Rest_Server Usage - Classes:

```php
<?php
require_once 'Zend/Rest/Server.php';
require_once 'My/Service/Class.php';

$server = new Zend_Rest_Server();
$server->setClass('My_Service_Class');
$server->handle();
```

Zend_Rest_Server Usage - Functions:

```php
<?php
require_once 'Zend/Rest/Server.php';

/**
 * Say Hello
 *
 * @param string $who
 * @param string $when
 * @return string
 */
function sayHello($who, $when)
{
    return "Hello $who, Good $when";
}

$server = new Zend_Rest_Server();
$server->addFunction('sayHello');
$server->handle();
```

Calling a Zend_Rest_Server Service:

To call a Zend_Rest_Server service, supply a GET/POST method argument with a value that is the method you wish to call. You can then follow that up with any number of arguments using either the name of the argument (i.e. "who") or using arg following by the numeric position of the argument (i.e. "arg1"). Using the example above, sayHello can be called in two ways:

ZEND_REST_SERVER

Returning Custom XML Responses:

To return custom XML, simply return a DOMDocument, DOMElement or SimpleXMLElement object. The response from the service will be returned without modification to the client.

```
<?php
require_once 'Zend/Rest/Server.php';

/**
 * Say Hello
 *
 * @param string $who
 * @param string $when
 * @return SimpleXMLElement
 */
function sayHello($who, $when)
{
    $xml ='<?xml version="1.0" encoding="ISO-8859-1"?>
<mysite>
    <value>Hey $who! Hope you're having a good $when</value>
    <code>200</code>
</mysite>';

    $xml = simplexml_load_string($xml);
    return $xml;
}

$server = new Zend_Rest_Server();
$server->addFunction('sayHello);

$server->handle();
```

ZEND_SERVICE

Zend_Service_Abstract is an abstract class which serves as a foundation for web service implementations (also look at Zend_Rest_Client to support generic, XML-based REST services).

While Zend_Service is extensible, Zend also provides support for popular web services. Examples of web service support packaged with Zend Framework include:

- Akismet
- Amazon
- Audioscrobbler
- Del.icio.us
- Flickr
- Simpy
- SlideShare
- StrikeIron
- Yahoo!

HTTP Clients:

For those Zend services that utilize HTTP requests, change the HTTP client of Zend_Rest_Client to change which HTTP client the service uses.

```
<?php
$myHttpClient = new My_Http_Client();
Zend_Service_Akismet::setHttpClient($myHttpClient);
```

When making more than one request with a service, configure the HTTP client to keep connections alive and speed the requests.

```
<?php
Zend_Service_Akismet::getHttpClient()->setConfig(array(
        'keepalive' => true
));
```

TEST YOUR KNOWLEDGE : QUESTIONS

1

XML-RPC fault responses are reported by Zend_XmlRpc_Client by:

a. Raising an exception

b. Triggering an error

c. Using the client's isFault() method

d. Checking for a fault Message in the response

2

Zend_Rest_Client expects a REST service that returns what type of content?

a. Plain text

b. JSON

c. HTML

d. XML

Test Your Knowledge : Answers

★ = CORRECT

1

XML-RPC fault responses are reported by Zend_XmlRpc_Client by:

★ a. Raising an exception

b. Triggering an error

c. Using the client's isFault() method

d. Checking for a fault Message in the response

2

Zend_Rest_Client expects a REST service that returns what type of content?

a. Plain text

b. JSON

c. HTML

★ d. XML