香 港 科 技 大 學
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

HKUST PHP Developer Guidelines

(Draft)

Table of Content

1. Introduction

This document serves to provide the general guidelines for developers implementing system / website adopting PHP as the developing tool.

2. PHP Version to be adopted

As indicated by PHP.net, the average active support life span of PHP version is around 3 years only [#1], with reference to the Common Vulnerabilities and Exposures (CVE) provided by the National Vulnerability Database, the number of vulnerabilities related to PHP is considered high[#2] when compare to other programming languages, as such, it is highly recommended that the latest version of PHP available should be adopted for new project as far as possible to ensure a longer active support time of security patches.

For systems already in maintenance stage, regular review (e.g. yearly) should be conducted to ensure the version in use is still in active support status. In addition, related security patches should be applied to the production environment following the minimum security guidelines set by ITSC[#3].

3. Project Folder Structure

It is a good practice to have a standard folder structure for different developers can understand the project structure easily. Following is a recommended php project folder structure.

```
- project_folder/
   - config/
   - css/
   - images/
   - includes/
   - js/
   - lib/
   - index.php
```

Folder	**Usages**
config	Contains environment dependent resources (DB connection config, URL, application properties)
css	css files
images	Images files
includes	Contains .php file which is commonly called
js	javascript files, javascript libraries
lib	php library

4. Coding Standards and Guide

In addition to keeping the version of PHP up-to-date, it is equally important to adopt a good coding style as majority of security breaches actually are related to insecure coding style. The following sections, which is based on the recommendations from the PHP Standards Recommendations (PSR)[#4], provide some good reference and best practices for PHP coding. Development teams should adopt the practices as far as possible if within project budgets and timelines.

Guidelines are detailed in separate sections below:

1. Basic Coding Standard (PSR-1)

2. Coding Style Guide (PSR-2)

3. Logger Interface

4. Caching Interface

4.1 Basic Coding Standard (PSR-1)

This section of the standard comprises what should be considered the standard coding elements that are required to ensure a high level of technical interoperability between shared PHP code.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

4.1.1 Overview

- Files MUST use only <?php and <?= tags.
- Files MUST use only **UTF-8 without BOM** for PHP code.
- Files SHOULD *either* declare symbols (classes, functions, constants, etc.) *or* cause side-effects (e.g. generate output, change .ini settings, etc.) but SHOULD NOT do both.
- **Class names** MUST be declared in **StudlyCaps**.
- **Class constants** MUST be declared in **all upper case with underscore separators**.
- **Method names** MUST be declared in **camelCase**.

4.1.2 Files

a. PHP Tags

PHP code MUST use the long <?php ?> tags or the short-echo <?= ?> tags; it MUST NOT use the other tag variations.

b. Character Encoding

PHP code MUST use only **UTF-8 without BOM**.

c. Side Effects

A file SHOULD declare new symbols (classes, functions, constants, etc.) and cause no other side effects, or it SHOULD execute logic with side effects, but SHOULD NOT do both.

The phrase "side effects" means execution of logic not directly related to declaring classes, functions, constants, etc., *merely from including the file*.

"Side effects" include but are not limited to: generating output, explicit use of **require** or **include**, connecting to external services, modifying ini settings, emitting errors or exceptions, modifying global or static variables, reading from or writing to a file, and so on.

The following is an example of a file with both declarations and side effects; i.e, an example of what to avoid:

```
<?php
// side effect: change ini settings
ini_set('error_reporting', E_ALL);

// side effect: loads a file
include "file.php";

// side effect: generates output
echo "<html>\n";

// declaration
function foo()
{
    // function body
}
```

The following example is of a file that contains declarations without side effects; i.e., an example of what to emulate:

```
<?php
// declaration
function foo()
{
    // function body
}

// conditional declaration is *not* a side effect
if (! function_exists('bar')) {
    function bar()
    {
        // function body
    }
}
```

4.1.3 Namespace and Class Names

This means each class is in a file by itself, and is in a namespace of at least one level: a top-level vendor name.

Class names MUST be declared in **StudlyCaps**.
Code written for PHP 5.3 and after MUST use formal namespaces.

For example:

```php
<?php
// PHP 5.3 and later:
namespace Vendor\Model;

class Foo
{
}
```

Code written for 5.2.x and before SHOULD use the pseudo-namespacing convention of `Vendor_` prefixes on class names.

```php
<?php
// PHP 5.2.x and earlier:
class Vendor_Model_Foo
{
}
```

4.1.4 Class Constants, Properties, and Methods

The term "class" refers to all classes, interfaces, and traits.

a. Constants

Class constants MUST be declared in all upper case with underscore separators. For example:

```php
<?php
namespace Vendor\Model;

class Foo
{
    const VERSION = '1.0';
    const DATE_APPROVED = '2012-06-01';
}
```

b. Properties

This guide intentionally avoids any recommendation regarding the use of **$StudlyCaps**, **$camelCase**, or $**under_score**property names.

Whatever naming convention is used SHOULD be applied consistently within a reasonable scope. That scope may be vendor-level, package-level, class-level, or method-level.

c. Methods

Method names MUST be declared in **camelCase().**

4.2 Coding Style Guide (PSR-2)

This guide extends and expands on PSR-1, the basic coding standard.

The intent of this guide is to reduce cognitive friction when scanning code from different authors. It does so by enumerating a shared set of rules and expectations about how to format PHP code.

The style rules herein are derived from commonalities among the various member projects. When various authors collaborate across multiple projects, it helps to have one set of guidelines to be used among all those projects. Thus, the benefit of this guide is not in the rules themselves, but in the sharing of those rules.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

4.2.1 Overview

- Code MUST follow a "coding style guide" PSR [PSR-1].
- Code MUST use 4 spaces for indenting, not tabs.
- There MUST NOT be a hard limit on line length; the soft limit MUST be 120 characters; lines SHOULD be 80 characters or less.
- There MUST be one blank line after the namespace declaration, and there MUST be one blank line after the block of usedeclarations.
- Opening braces for classes MUST go on the next line, and closing braces MUST go on the next line after the body.
- Opening braces for methods MUST go on the next line, and closing braces MUST go on the next line after the body.
- Visibility MUST be declared on all properties and methods; abstract and final MUST be declared before the visibility; static MUST be declared after the visibility.
- Control structure keywords MUST have one space after them; method and function calls MUST NOT.
- Opening braces for control structures MUST go on the same line, and closing braces MUST go on the next line after the body.
- Opening parentheses for control structures MUST NOT have a space after them, and closing parentheses for control structures MUST NOT have a space before.

The example below encompasses some of the rules below as a quick overview:

```
<?php
namespace Vendor\Package;

use FooInterface;
use BarClass as Bar;
use OtherVendor\OtherPackage\BazClass;

class Foo extends Bar implements FooInterface
{
    public function sampleMethod($a, $b = null)
    {
        if ($a === $b) {
            bar();
        } elseif ($a > $b) {
            $foo->bar($arg1);
        } else {
            BazClass::bar($arg2, $arg3);
        }
    }

    final public static function bar()
    {
        // method body
    }
}
```

4.2.2 General

a. Basic Coding Standard

Code MUST follow all rules outlined in PSR-1.

b. Files

All PHP files MUST use the Unix LF (linefeed) line ending.

All PHP files MUST end with a single blank line.

The closing ?> tag MUST be omitted from files containing only PHP.

c. Lines

There MUST NOT be a hard limit on line length.

The soft limit on line length MUST be 120 characters; automated style checkers MUST warn but MUST NOT error at the soft limit.

Lines SHOULD NOT be longer than 80 characters; lines longer than that SHOULD be split into multiple subsequent lines of no more than 80 characters each.

There **MUST NOT be trailing whitespace** at the end of non-blank lines.

Blank lines MAY be added to improve readability and to indicate related blocks of code.

There MUST NOT be more than one statement per line.

d. Indenting

Code **MUST use an indent of 4 spaces**, and **MUST NOT use tabs** for indenting.

N.b.: Using only spaces, and not mixing spaces with tabs, helps to avoid problems with diffs, patches, history, and annotations. The use of spaces also makes it easy to insert fine-grained sub-indentation for inter-line alignment.

e. Keywords and True/False/Null

PHP keywords MUST be in **lower case**.
The PHP constants **true**, **false**, and **null** MUST be in **lower case**.

4.2.3 Namespace and Use Declarations

When present, there MUST be one blank line after the namespace declaration.
When present, all use declarations MUST go after the namespace declaration.
There MUST be one use keyword per declaration.
There MUST be one blank line after the use block.

For example:

```php
<?php
namespace Vendor\Package;

use FooClass;
use BarClass as Bar;
use OtherVendor\OtherPackage\BazClass;

// ... additional PHP code ...
```

4.2.4 Classes, Properties, and Methods

The term "class" refers to all classes, interfaces, and traits.

a. Extends and Implements

The extends and implements keywords MUST be declared on the same line as the class name.

The opening brace for the class MUST go on its own line; the closing brace for the class MUST go on the next line after the body.

```php
<?php
namespace Vendor\Package;

use FooClass;
use BarClass as Bar;
use OtherVendor\OtherPackage\BazClass;

class ClassName extends ParentClass implements \ArrayAccess, \Countable
{
    // constants, properties, methods
}
```

Lists of implements MAY be split across multiple lines, where each subsequent line is indented once. When doing so, the first item in the list MUST be on the next line, and there MUST be only one interface per line.

```php
<?php
namespace Vendor\Package;

use FooClass;
use BarClass as Bar;
use OtherVendor\OtherPackage\BazClass;

class ClassName extends ParentClass implements
    \ArrayAccess,
    \Countable,
    \Serializable
{
    // constants, properties, methods
}
```

b. Properties

Visibility MUST be declared on all properties.

The var keyword MUST NOT be used to declare a property.
There MUST NOT be more than one property declared per statement.

Property names SHOULD NOT be prefixed with a single underscore to indicate protected or private visibility.

A property declaration looks like the following.

```php
<?php
namespace Vendor\Package;

```

```
class ClassName
{
    public $foo = null;
}
```

c. Methods

Visibility MUST be declared on all methods.

Method names SHOULD NOT be prefixed with a single underscore to indicate protected or private visibility.

Method names MUST NOT be declared with a space after the method name. The opening brace MUST go on its own line, and the closing brace MUST go on the next line following the body. There MUST NOT be a space after the opening parenthesis, and there MUST NOT be a space before the closing parenthesis.

A method declaration looks like the following. Note the placement of parentheses, commas, spaces, and braces:

```
<?php
namespace Vendor\Package;

class ClassName
{
    public function fooBarBaz($arg1, &$arg2, $arg3 = [])
    {
        // method body
    }
}
```

d. Method Arguments

In the argument list, there MUST NOT be a space before each comma, and there MUST be one space after each comma.

Method arguments with default values MUST go at the end of the argument list.

```
<?php
namespace Vendor\Package;

class ClassName
{
    public function foo($arg1, &$arg2, $arg3 = [])
    {
        // method body
    }
}
```

Argument lists MAY be split across multiple lines, where each subsequent line is indented once. When doing so, the first item in the list MUST be on the next line, and there MUST be only one argument per line.

When the argument list is split across multiple lines, the closing parenthesis and opening brace MUST be placed together on their own line with one space between them.

```php
<?php
namespace Vendor\Package;

class ClassName
{
    public function aVeryLongMethodName(
        ClassTypeHint $arg1,
        &$arg2,
        array $arg3 = []
    ) {
        // method body
    }
}
```

e. abstract, final, and static

When present, the abstract and final declarations MUST precede the visibility declaration.
When present, the static declaration MUST come after the visibility declaration.

```php
<?php
namespace Vendor\Package;

abstract class ClassName
{
    protected static $foo;

    abstract protected function zim();

    final public static function bar()
    {
        // method body
    }
}
```

f. Method and Function Calls

When making a method or function call, there MUST NOT be a space between the method or function name and the opening parenthesis, there MUST NOT be a space after the opening parenthesis, and there MUST NOT be a space before the closing parenthesis. In the argument list, there MUST NOT be a space before each comma, and there MUST be one space after each comma.

```php
<?php
bar();
$foo->bar($arg1);
Foo::bar($arg2, $arg3);
```

Argument lists MAY be split across multiple lines, where each subsequent line is indented once. When doing so, the first item in the list MUST be on the next line, and there MUST be only one argument per line.

```php
<?php
$foo->bar(
    $longArgument,
    $longerArgument,
    $muchLongerArgument
);
```

4.2.5 Control Structures

The general style rules for control structures are as follows:

- There MUST be one space after the control structure keyword
- There MUST NOT be a space after the opening parenthesis
- There MUST NOT be a space before the closing parenthesis
- There MUST be one space between the closing parenthesis and the opening brace
- The structure body MUST be indented once
- The closing brace MUST be on the next line after the body

The body of each structure MUST be enclosed by braces. This standardizes how the structures look, and reduces the likelihood of introducing errors as new lines get added to the body.

a. if, elseif, else

An if structure looks like the following. Note the placement of parentheses, spaces, and braces; and that else and elseifare on the same line as the closing brace from the earlier body.

```php
<?php
if ($expr1) {
    // if body
} elseif ($expr2) {
    // elseif body
} else {
    // else body;
}
```

The keyword elseif SHOULD be used instead of else if so that all control keywords look like single words.

b. switch, case

A switch structure looks like the following. Note the placement of parentheses, spaces, and braces. The case statement MUST be indented once from switch, and the break keyword (or other terminating keyword) MUST be indented at the same level as the case body. There MUST be a comment such as // no break when fall-through is intentional in a non-empty case body.

```php
<?php
switch ($expr) {
    case 0:
        echo 'First case, with a break';
        break;
    case 1:
        echo 'Second case, which falls through';
        // no break
    case 2:
    case 3:
    case 4:
        echo 'Third case, return instead of break';
        return;
    default:
        echo 'Default case';
        break;
}
```

c. while, do while

A while statement looks like the following. Note the placement of parentheses, spaces, and braces.

```php
<?php
while ($expr) {
    // structure body
}
```

Similarly, a do while statement looks like the following. Note the placement of parentheses, spaces, and braces.

```php
<?php
do {
    // structure body;
} while ($expr);
```

d. for

A for statement looks like the following. Note the placement of parentheses, spaces, and braces.

```php
<?php
for ($i = 0; $i < 10; $i++) {
    // for body
}
```

e. foreach

A `foreach` statement looks like the following. Note the placement of parentheses, spaces, and braces.

```php
<?php
foreach ($iterable as $key => $value) {
    // foreach body
}
```

f. try, catch

A try catch block looks like the following. Note the placement of parentheses, spaces, and braces.

```php
<?php
try {
    // try body
} catch (FirstExceptionType $e) {
    // catch body
} catch (OtherExceptionType $e) {
    // catch body
}
```

4.2.6 Closures

Closures MUST be declared with a space after the function keyword, and a space before and after the use keyword.

The opening brace MUST go on the same line, and the closing brace MUST go on the next line following the body.

There MUST NOT be a space after the opening parenthesis of the argument list or variable list, and there MUST NOT be a space before the closing parenthesis of the argument list or variable list.

In the argument list and variable list, there MUST NOT be a space before each comma, and there MUST be one space after each comma.

Closure arguments with default values MUST go at the end of the argument list.

A closure declaration looks like the following. Note the placement of parentheses, commas, spaces, and braces:

```php
<?php
$closureWithArgs = function ($arg1, $arg2) {
    // body
};

$closureWithArgsAndVars = function ($arg1, $arg2) use ($var1, $var2) {
    // body
};
```

Argument lists and variable lists MAY be split across multiple lines, where each subsequent line is indented once. When doing so, the first item in the list MUST be on the next line, and there MUST be only one argument or variable per line.

When the ending list (whether of arguments or variables) is split across multiple lines, the closing parenthesis and opening brace MUST be placed together on their own line with one space between them.

The following are examples of closures with and without argument lists and variable lists split across multiple lines.

```php
<?php
$longArgs_noVars = function (
    $longArgument,
    $longerArgument,
    $muchLongerArgument
) {
    // body
};

$noArgs_longVars = function () use (
    $longVar1,
    $longerVar2,
    $muchLongerVar3
) {
    // body
};

$longArgs_longVars = function (
    $longArgument,
    $longerArgument,
    $muchLongerArgument
) use (
    $longVar1,
    $longerVar2,
    $muchLongerVar3
) {
    // body
};

$longArgs_shortVars = function (
    $longArgument,
    $longerArgument,
    $muchLongerArgument
) use ($var1) {
    // body
};

$shortArgs_longVars = function ($arg) use (
    $longVar1,
    $longerVar2,
    $muchLongerVar3
) {
    // body
};
```

Note that the formatting rules also apply when the closure is used directly in a function or method call as an argument.

```
<?php
$foo->bar(
    $arg1,
    function ($arg2) use ($var1) {
        // body
    },
    $arg3
);
```

4.3 Logger Interface

To improve traceability, errors leading to system/ function failures should always be logged. Pre-building logging mechanism which can be turned on when needed to facilitate debugging is also a good practice.

This section describes a commonly adopted interface for logging libraries. Other methods addressing the necessary level of logging needs can also be applied if deems more appropriate.

The main objective is to enable libraries to receive a Psr\Log\LoggerInterface object and write logs to it in a simple and universal way. Frameworks and CMSs that have custom needs MAY extend the interface for their own purpose, but SHOULD remain compatible with this document. This ensures that the third-party libraries an application uses can write to the centralized application logs.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

The word implementor in this document is to be interpreted as someone implementing the LoggerInterface in a log-related library or framework. Users of loggers are referred to as user.

4.3.1 Specification

a. Basics

- The LoggerInterface exposes eight methods to write logs to the eight RFC 5424 levels (debug, info, notice, warning, error, critical, alert, emergency).

- A ninth method, log, accepts a log level as the first argument. Calling this method with one of the log level constants MUST have the same result as calling the level-specific method. Calling this method with a level not defined by this specification MUST throw a Psr\Log\InvalidArgumentException if the implementation does not know about the level. Users SHOULD NOT use a custom level without knowing for sure the current implementation supports it.

b. Message

- Every method accepts a string as the message, or an object with a __toString() method. Implementors MAY have special handling for the passed objects. If that is not the case, implementors MUST cast it to a string.

- The message MAY contain placeholders which implementors MAY replace with values from the context array.

Placeholder names MUST correspond to keys in the context array.

Placeholder names MUST be delimited with a single opening brace { and a single closing brace }. There MUST NOT be any whitespace between the delimiters and the placeholder name.
Placeholder names SHOULD be composed only of the characters A-Z, a-z, 0-9, underscore _, and period .. The use of other characters is reserved for future modifications of the placeholders specification.

Implementors MAY use placeholders to implement various escaping strategies and translate logs for display. Users SHOULD NOT pre-escape placeholder values since they can not know in which context the data will be displayed.

The following is an example implementation of placeholder interpolation provided for reference purposes only:

```php
<?php

/**
 * Interpolates context values into the message placeholders.
 */
function interpolate($message, array $context = array())
{
    // build a replacement array with braces around the context keys
    $replace = array();
    foreach ($context as $key => $val) {
        // check that the value can be casted to string
        if (!is_array($val) && (!is_object($val) || method_exists($val, '__toString')))
{
            $replace['{' . $key . '}'] = $val;
        }
    }

    // interpolate replacement values into the message and return
    return strtr($message, $replace);
}

// a message with brace-delimited placeholder names
$message = "User {username} created";

// a context array of placeholder names => replacement values
$context = array('username' => 'bolivar');

// echoes "User bolivar created"
echo interpolate($message, $context);
```

c. Context

- Every method accepts an array as context data. This is meant to hold any extraneous information that does not fit well in a string. The array can contain anything. Implementors MUST ensure they treat context data with as much lenience as possible. A given value in the context MUST NOT throw an exception nor raise any php error, warning or notice.

- If an Exception object is passed in the context data, it MUST be in the 'exception' key. Logging exceptions is a common pattern and this allows implementors to extract a stack trace from the exception when the log backend supports it. Implementors MUST still verify that the 'exception' key is actually an Exception before using it as such, as it MAY contain anything.

d. Helper classes and interfaces

- The Psr\Log\AbstractLogger class lets you implement the LoggerInterface very easily by extending it and implementing the generic log method. The other eight methods are forwarding the message and context to it.
- Similarly, using the Psr\Log\LoggerTrait only requires you to implement the generic log method. Note that since traits can not implement interfaces, in this case you still have to implement LoggerInterface.
- The Psr\Log\NullLogger is provided together with the interface. It MAY be used by users of the interface to provide a fall-back "black hole" implementation if no logger is given to them. However, conditional logging may be a better approach if context data creation is expensive.
- The Psr\Log\LoggerAwareInterface only contains a setLogger(LoggerInterface $logger) method and can be used by frameworks to auto-wire arbitrary instances with a logger.
- The Psr\Log\LoggerAwareTrait trait can be used to implement the equivalent interface easily in any class. It gives you access to $this->logger.
- The Psr\Log\LogLevel class holds constants for the eight log levels.

4.3.2 Package

The interfaces and classes described as well as relevant exception classes and a test suite to verify your implementation are provided as part of the psr/log package.

4.3.3 Psr\Log\LoggerInterface

```
<?php

namespace Psr\Log;
```

```php
/**
 * Describes a logger instance
 *
 * The message MUST be a string or object implementing __toString().
 *
 * The message MAY contain placeholders in the form: {foo} where foo
 * will be replaced by the context data in key "foo".
 *
 * The context array can contain arbitrary data, the only assumption that
 * can be made by implementors is that if an Exception instance is given
 * to produce a stack trace, it MUST be in a key named "exception".
 *
 * See https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-3-logger-
interface.md
 * for the full interface specification.
 */
interface LoggerInterface
{
    /**
     * System is unusable.
     *
     * @param string $message
     * @param array $context
     * @return null
     */
    public function emergency($message, array $context = array());

    /**
     * Action must be taken immediately.
     *
     * Example: Entire website down, database unavailable, etc. This should
     * trigger the SMS alerts and wake you up.
     *
     * @param string $message
     * @param array $context
     * @return null
     */
    public function alert($message, array $context = array());

    /**
     * Critical conditions.
     *
     * Example: Application component unavailable, unexpected exception.
     *
     * @param string $message
     * @param array $context
     * @return null
     */
    public function critical($message, array $context = array());

    /**
     * Runtime errors that do not require immediate action but should typically
     * be logged and monitored.
     *
     * @param string $message
     * @param array $context
     * @return null
     */
    public function error($message, array $context = array());
    /**
     * Exceptional occurrences that are not errors.
     *
     * Example: Use of deprecated APIs, poor use of an API, undesirable things
     * that are not necessarily wrong.
     *
     * @param string $message
     * @param array $context
     * @return null
     */
    public function warning($message, array $context = array());

    /**
     * Normal but significant events.
     *
     * @param string $message
     * @param array $context
     * @return null
     */
    public function notice($message, array $context = array());

    /**
     * Interesting events.
     *
     * Example: User logs in, SQL logs.
     *
     * @param string $message
     * @param array $context
     * @return null
     */
    public function info($message, array $context = array());

    /**
     * Detailed debug information.
     *
     * @param string $message
     * @param array $context
     * @return null
     */
    public function debug($message, array $context = array());

    /**
     * Logs with an arbitrary level.
     *
     * @param mixed $level
     * @param string $message
     * @param array $context
     * @return null
     */
    public function log($level, $message, array $context = array());
}
```

4.3.4 Psr\Log\LoggerAwareInterface

```
<?php

namespace Psr\Log;

/**
```

```
 * Describes a logger-aware instance
 */
interface LoggerAwareInterface
{
    /**
     * Sets a logger instance on the object
     *
     * @param LoggerInterface $logger
     * @return null
     */
    public function setLogger(LoggerInterface $logger);
}
```

4.3.5 Psr\Log\LogLevel

```
<?php

namespace Psr\Log;

/**
 * Describes log levels
 */
class LogLevel
{
    const EMERGENCY = 'emergency';
    const ALERT     = 'alert';
    const CRITICAL  = 'critical';
    const ERROR     = 'error';
    const WARNING   = 'warning';
    const NOTICE    = 'notice';
    const INFO      = 'info';
    const DEBUG     = 'debug';
}
```

4.3.6 Scenarios may require logging

a. ERROR: $this->logger->error

- All **expected error**, e.g. API fail to call. Error message content should include which API is fail to call, when to call, etc.
- NEVER log sensitive information, e.g. personal data (phone, id number, etc) in ERROR log!

b. INFO: $this->logger->info

- Only information for notice should log in INFO, e.g. system start or shutdown, schedule job start or the job is done.
- NEVER log sensitive information, e.g. personal data (phone, id number, etc) in ERROR log!

c. DEBUG $this->logger->debug

- Since DEBUG log is for developer's reference, it can be logged whenever debug message is need. But NEVER set debug level in production environment.

4.3.7 Implementation Examples:

```php
<?php

use Psr\Log\LoggerInterface;

class Foo
{
    private $logger;

    public function __construct(LoggerInterface $logger = null)
    {
        $this->logger = $logger;
    }

    public function doSomething()
    {
        if ($this->logger) {
            $this->logger->info('Doing work');
        }

        // do something useful
    }
}
```

Reference:

https://github.com/php-fig/log

https://github.com/php-fig/log/blob/master/Psr/Log/LoggerInterface.php

4.4 Caching Interface

Caching is a common way to improve the performance of any project, making caching libraries one of the most common features of many frameworks and libraries. This has lead to a situation where many libraries roll their own caching libraries, with various levels of functionality. These differences are causing developers to have to learn multiple systems which may or may not provide the functionality they need. In addition, the developers of caching libraries themselves face a choice between only supporting a limited number of frameworks or creating a large number of adapter classes.

A common interface for caching systems will solve these problems. Library and framework developers can count on the caching systems working the way they're expecting, while the developers of caching systems will only have to implement a single set of interfaces rather than a whole assortment of adapters.

Note: Caching Interface is not always required. For smaller scale projects, applying caching may even induce overhead and undesirable effects. ***Discussion with project team or project owner is necessary to determine whether cache is needed or not****.*

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

4.4.1 Goal

The goal of this PSR is to allow developers to create cache-aware libraries that can be integrated into existing frameworks and systems without the need for custom development.

4.4.2 Definitions

- **Calling Library** - The library or code that actually needs the cache services. This library will utilize caching services that implement this standard's interfaces, but will otherwise have no knowledge of the implementation of those caching services.

- **Implementing Library** - This library is responsible for implementing this standard in order to provide caching services to any Calling Library. The Implementing Library MUST provide classes which implement the Cache\CacheItemPoolInterface and Cache\CacheItemInterface interfaces. Implementing Libraries MUST support at minimum TTL functionality as described below with whole-second granularity.

- **TTL** - The Time To Live (TTL) of an item is the amount of time between when that item is stored and it is considered stale. The TTL is normally defined by an integer representing time in seconds, or a DateInterval object.

- **Expiration** - The actual time when an item is set to go stale. This it typically calculated by adding the TTL to the time when an object is stored, but may also be explicitly set with DateTime object.

An item with a 300 second TTL stored at 1:30:00 will have an expiration of 1:35:00.

Implementing Libraries MAY expire an item before its requested Expiration Time, but MUST treat an item as expired once its Expiration Time is reached. If a calling library asks for an item to be saved but does not specify an expiration time, or specifies a null expiration time or TTL, an Implementing Library MAY use a configured default duration. If no default duration has been set, the Implementing Library MUST interpret that as a request to cache the item forever, or for as long as the underlying implementation supports.

- **Key** - A string of at least one character that uniquely identifies a cached item. Implementing libraries MUST support keys consisting of the characters A-Z, a-z, 0-9, _, and . in any order in UTF-8 encoding and a length of up to 64 characters. Implementing libraries MAY support additional characters and encodings or longer lengths, but must support at least that minimum. Libraries are responsible for their own escaping of key strings as appropriate, but MUST be able to return the original unmodified key string. The following characters are reserved for future extensions and MUST NOT be supported by implementing libraries: {}()/\@:

- **Hit** - A cache hit occurs when a Calling Library requests an Item by key and a matching value is found for that key, and that value has not expired, and the value is not invalid for some other reason. Calling Libraries SHOULD make sure to verify isHit() on all get() calls.

- **Miss** - A cache miss is the opposite of a cache hit. A cache miss occurs when a Calling Library requests an item by key and that value not found for that key, or the value was found but has expired, or the value is invalid for some other reason. An expired value MUST always be considered a cache miss.

- **Deferred** - A deferred cache save indicates that a cache item may not be persisted immediately by the pool. A Pool object MAY delay persisting a deferred cache item in order to take advantage of bulk-set operations supported by some storage engines. A Pool MUST ensure that any deferred cache items are eventually persisted and data is not lost, and MAY persist them before a Calling Library requests that they be persisted. When a Calling Library invokes the commit() method all outstanding deferred items MUST be persisted. An Implementing Library MAY use whatever logic is appropriate to determine when to persist deferred items, such as an object destructor, persisting all on save(), a timeout or max-items check or any other appropriate logic. Requests for a cache item that has been deferred MUST return the deferred but not-yet-persisted item.

4.4.3 Data

Implementing libraries MUST support all serializable PHP data types, including:

- **Strings** - Character strings of arbitrary size in any PHP-compatible encoding.
- **Integers** - All integers of any size supported by PHP, up to 64-bit signed.
- **Floats** - All signed floating point values.
- **Boolean** - True and False.
- **Null** - The actual null value.
- **Arrays** - Indexed, associative and multidimensional arrays of arbitrary depth.
- **Object** - Any object that supports lossless serialization and deserialization such that $o == unserialize(serialize($o)). Objects MAY leverage PHP's Serializable interface, __sleep() or __wakeup() magic methods, or similar language functionality if appropriate.

All data passed into the Implementing Library MUST be returned exactly as passed. That includes the variable type. That is, it is an error to return (string) 5 if (int) 5 was the value saved. Implementing Libraries MAY use PHP's serialize()/unserialize() functions internally but are not required to do so. Compatibility with them is simply used as a baseline for acceptable object values.

If it is not possible to return the exact saved value for any reason, implementing libraries MUST respond with a cache miss rather than corrupted data.

4.4.4 Key Concepts

a. Pool

The Pool represents a collection of items in a caching system. The pool is a logical Repository of all items it contains. All cacheable items are retrieved from the Pool as an Item object, and all interaction with the whole universe of cached objects happens through the Pool.

b. Items

An Item represents a single key/value pair within a Pool. The key is the primary unique identifier for an Item and MUST be immutable. The Value MAY be changed at any time.

c. Error handling

While caching is often an important part of application performance, it should never be a critical part of application functionality. Thus, an error in a cache system SHOULD NOT result in application failure. For that reason, Implementing Libraries MUST NOT throw exceptions other than those defined by the interface, and SHOULD trap any errors or exceptions triggered by an underlying data store and not allow them to bubble.

An Implementing Library SHOULD log such errors or otherwise report them to an administrator as appropriate.

If a Calling Library requests that one or more Items be deleted, or that a pool be cleared, it MUST NOT be considered an error condition if the specified key does not exist. The post-condition is the same (the key does not exist, or the pool is empty), thus there is no error condition.

4.4.5 Interfaces

d. CacheItemInterface

CacheItemInterface defines an item inside a cache system. Each Item object MUST be associated with a specific key, which can be set according to the implementing system and is typically passed by the Cache\CacheItemPoolInterface object.

The Cache\CacheItemInterface object encapsulates the storage and retrieval of cache items. Each Cache\CacheItemInterface is generated by a Cache\CacheItemPoolInterface object, which is responsible for any required setup as well as associating the object with a unique Key. Cache\CacheItemInterface objects MUST be able to store and retrieve any type of PHP value defined in the Data section of this document.

Calling Libraries MUST NOT instantiate Item objects themselves. They may only be requested from a Pool object via the getItem() method. Calling Libraries SHOULD NOT assume that an Item created by one Implementing Library is compatible with a Pool from another Implementing Library.

```php
<?php

namespace Psr\Cache;

/**
 * CacheItemInterface defines an interface for interacting with objects inside a cache.
 */
interface CacheItemInterface
{
    /**
     * Returns the key for the current cache item.
     *
     * The key is loaded by the Implementing Library, but should be available to
     * the higher level callers when needed.
     *
     * @return string
     *   The key string for this cache item.
     */
    public function getKey();

     /**
     * Retrieves the value of the item from the cache associated with this object's key.
     *
     * The value returned must be identical to the value originally stored by set().
     *
     * If isHit() returns false, this method MUST return null. Note that null
     * is a legitimate cached value, so the isHit() method SHOULD be used to
     * differentiate between "null value was found" and "no value was found."
     *
     * @return mixed
     *   The value corresponding to this cache item's key, or null if not found.
     */
    public function get();

    /**
     * Confirms if the cache item lookup resulted in a cache hit.
     *
     * Note: This method MUST NOT have a race condition between calling isHit()
     * and calling get().
     *
     * @return bool
     *   True if the request resulted in a cache hit. False otherwise.
     */
    public function isHit();

    /**
     * Sets the value represented by this cache item.
     *
     * The $value argument may be any item that can be serialized by PHP,
     * although the method of serialization is left up to the Implementing
     * Library.
     *
     * @param mixed $value
     *   The serializable value to be stored.
     *
     * @return static
     *   The invoked object.
     */
    public function set($value);

    /**
     * Sets the expiration time for this cache item.
     *
     * @param \DateTimeInterface|null $expiration
     *   The point in time after which the item MUST be considered expired.
     *   If null is passed explicitly, a default value MAY be used. If none is set,
     *   the value should be stored permanently or for as long as the
     *   implementation allows.
     *
     * @return static
     *   The called object.
     */
    public function expiresAt($expiration);

    /**
     * Sets the expiration time for this cache item.
     *
     * @param int|\DateInterval|null $time
     *   The period of time from the present after which the item MUST be considered
     *   expired. An integer parameter is understood to be the time in seconds until
     *   expiration. If null is passed explicitly, a default value MAY be used.
     *   If none is set, the value should be stored permanently or for as long as the
     *   implementation allows.
     *
     * @return static
     *   The called object.
     */
    public function expiresAfter($time);

}
```

e. CacheItemPoolInterface

The primary purpose of Cache\CacheItemPoolInterface is to accept a key from the Calling Library and return the associated Cache\CacheItemInterface object. It is also the primary point of interaction with the entire cache collection. All configuration and initialization of the Pool is left up to an Implementing Library.

```
<?php

namespace Psr\Cache;

/**
 * CacheItemPoolInterface generates CacheItemInterface objects.
 */
interface CacheItemPoolInterface
{
    /**
     * Returns a Cache Item representing the specified key.
     *
     * This method must always return a CacheItemInterface object, even in case of
     * a cache miss. It MUST NOT return null.
     *
     * @param string $key
     *   The key for which to return the corresponding Cache Item.
     *
     * @throws InvalidArgumentException
     *   If the $key string is not a legal value a \Psr\Cache\InvalidArgumentException
     *   MUST be thrown.
     *
     * @return CacheItemInterface
     *   The corresponding Cache Item.
     */
    public function getItem($key);

    /**
     * Returns a traversable set of cache items.
     *
     * @param string[] $keys
     *   An indexed array of keys of items to retrieve.
     *
     * @throws InvalidArgumentException
     *   If any of the keys in $keys are not a legal value a
\Psr\Cache\InvalidArgumentException
     *   MUST be thrown.
     *
     * @return array|\Traversable
     *   A traversable collection of Cache Items keyed by the cache keys of
     *   each item. A Cache item will be returned for each key, even if that
     *   key is not found. However, if no keys are specified then an empty
     *   traversable MUST be returned instead.
     */
    public function getItems(array $keys = array());
    /**
     * Confirms if the cache contains specified cache item.
     *
     * Note: This method MAY avoid retrieving the cached value for performance reasons.
     * This could result in a race condition with CacheItemInterface::get(). To avoid
     * such situation use CacheItemInterface::isHit() instead.
     *
     * @param string $key
     *   The key for which to check existence.
     *
     * @throws InvalidArgumentException
     *   If the $key string is not a legal value a \Psr\Cache\InvalidArgumentException
     *   MUST be thrown.
     *
     * @return bool
     *   True if item exists in the cache, false otherwise.
     */
    public function hasItem($key);

    /**
     * Deletes all items in the pool.
     *
     * @return bool
     *   True if the pool was successfully cleared. False if there was an error.
     */
    public function clear();

    /**
     * Removes the item from the pool.
     *
     * @param string $key
     *   The key to delete.
     *
     * @throws InvalidArgumentException
     *   If the $key string is not a legal value a \Psr\Cache\InvalidArgumentException
     *   MUST be thrown.
     *
     * @return bool
     *   True if the item was successfully removed. False if there was an error.
     */
    public function deleteItem($key);

    /**
     * Removes multiple items from the pool.
     *
     * @param string[] $keys
     *   An array of keys that should be removed from the pool.

     * @throws InvalidArgumentException
     *   If any of the keys in $keys are not a legal value a
\Psr\Cache\InvalidArgumentException
     *   MUST be thrown.
     *
     * @return bool
     *   True if the items were successfully removed. False if there was an error.
     */
    public function deleteItems(array $keys);

    /**
     * Persists a cache item immediately.
     *
     * @param CacheItemInterface $item
     *   The cache item to save.
     *
     * @return bool
     *   True if the item was successfully persisted. False if there was an error.
     */
    public function save(CacheItemInterface $item);

    /**
     * Sets a cache item to be persisted later.
     *
     * @param CacheItemInterface $item
     *   The cache item to save.
     *
     * @return bool
     *   False if the item could not be queued or if a commit was attempted and failed.
True otherwise.
     */
    public function saveDeferred(CacheItemInterface $item);

    /**
     * Persists any deferred cache items.
     *
     * @return bool
     *   True if all not-yet-saved items were successfully saved or there were none.
False otherwise.
     */
    public function commit();
}
```

f. CacheException

This exception interface is intended for use when critical errors occur, including but not limited to *cache setup* such as connecting to a cache server or invalid credentials supplied.

Any exception thrown by an Implementing Library MUST implement this interface.

```
<?php

namespace Psr\Cache;

/**
 * Exception interface for all exceptions thrown by an Implementing Library.
 */
interface CacheException
{
}
```

g. InvalidArgumentException

```
<?php

namespace Psr\Cache;

/**
 * Exception interface for invalid cache arguments.
 *
 * Any time an invalid argument is passed into a method it must throw an
 * exception class which implements Psr\Cache\InvalidArgumentException.
 */
interface InvalidArgumentException extends CacheException
{
}
```

Reference

#1 http://php.net/supported-versions.php

#2 http://www.cvedetails.com/product/128/PHP-PHP.html?vendor_id=74

#3 http://itsc.ust.hk/it-policies-guidelines/minimum-security-standard/

#4 http://www.php-fig.org/psr/