PSR-3: Logger Interface

This document describes a common interface for logging libraries.

The main goal is to allow 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`.

1. Specification

1.1 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.

1.2 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 => $val) {
        // check that the value can be cast to string
        if (!is_array($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);
```

1.3 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.

1.4 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.

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.

3. `Psr\Log\LoggerInterface`

```php
<?php

namespace Psr\Log;

/**
 * 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 $context
     * @return void
     */
    public function emergency($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 $context
     * @return void
     */
    public function alert($message, array $context = array());

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

    /**
```

```
     * Runtime errors that do not require immediate action but should typically
     * be logged and monitored.
     *
     * @param string $message
     * @param array $context
     * @return void
     */
    public function error($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 $context
     * @return void
     */
    public function warning($message, array $context = array());

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

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

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

    /**
     * Logs with an arbitrary level.
     *
     * @param mixed $level
     * @param string $message
     * @param array $context
```

```
     * @return void
     */
    public function log($level, $message, array $context = array());
}
```

4. `Psr\Log\LoggerAwareInterface`

```php
<?php

namespace Psr\Log;

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

5. `Psr\Log\LogLevel`

```php
<?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';
}
```