PHP

PSRs - FIG

Ñ

Carlos Valencia R.

Contenido

Acerca de

Apreciado amigo/a,

Este documento reune las recomendaciones FIG-PSRs en este momento de Julio del 2021, puede encontrar los PSRs Aceptados (PSR - 1, PSR - 3, PSR - 4, PSR - 6, PSR - 7, PSR - 11, PSR - 12, PSR - 13, PSR - 14, PSR - 15, PSR - 16, PSR - 17 y PSR - 18) y, los considerados como Borrador: (PSR - 5, PSR - 19 y PSR - 20), es una traducción al Español, he utilizado el excelente servicio de traductor que ofrece el Sr. Google; mi nivel de Inglés es muy limitado, le pido indulgencia por los errores que pueda encontrar, lo he repasado pero...

Espero que le pueda servir y gracias por su atención.

En Barcelona, Julio del 2021
Carlos Valencia R.

Foto de portada: Antoni Gaudí, La Pedrera - Casa Milà, chimeneas.
Foto procedente de: Pigsels - Fotos de alta resolución libres de derechos.

Introduccción

PSR (PHP Standard Recommendation) es una recomendación de especificación estándar de PHP publicada por PHP Framework Interop Groupy sirve para la estandarización de conceptos de programación en PHP.

El objetivo es permitir la interoperabilidad de los componentes y proporcionar una base técnica común para la implementación de conceptos probados para prácticas óptimas de programación y prueba. El PHP-FIG está formado por varios fundadores de Frameworks PHP.

PHP-FIG, PHP Framework Interop Group, nació en 2009 de la unión de los Frameworks y proyectos PHP más importantes (como Symfony, Drupal, Laravel, etc.) Su propósito inicial era estandarizar soluciones para las necesidades más comunes de las aplicaciones PHP.

Por ejemplo, gracias a los estándares PSR-0 y PSR-4, todas las aplicaciones PHP cargan sus clases de la misma manera. Gracias a los estándares PSR-1 y PSR-2, muchas aplicaciones usan el mismo formato para escribir PHP.

En muchos documentos de seguimiento de normas se utilizan varias palabras para indicar los requisitos de la especificación. Estas palabras están a menudo en mayúsculas. Este documento define que deben interpretarse como se describe en RFC 2119.

Al final del documento puede encontrar un extracto del RFC 2119 con las palabras mencionadas y su interpretación.

PSR 1- Basic Coding Standard (Estándar básico de codificación)

En el documentos se utilizan varias palabras para indicar los requisitos de la especificación. Estas palabras están a menudo en mayúsculas y deben interpretarse como se describe en la RFC 2119. Puede encontrar un extracto del RFC 2119 con las palabras mencionadas y su interpretación.

Esta sección comprende lo que deben considerarse los elementos de codificación estándar requeridos para garantizar un alto nivel de interoperabilidad técnica entre el código PHP compartido.

Información general

Puede consultar el original en: https://www.php-fig.org/psr/psr-1

Esta sección comprende lo que deben considerarse los elementos de codificación estándar requeridos para garantizar un alto nivel de interoperabilidad técnica entre el código PHP compartido.

- Los archivos DEBEN usar solo las etiquetas `<?php` y `<?=` .
- Los archivos DEBEN usar solo UTF-8 sin BOM (Byte Order Mask) para el código PHP.
- Un archivo DEBE declarar símbolos (clases, funciones, constantes, etc.) o lógica secundaría (generar informes, cambiar la configuración, etc...) pero NO DEBE hacer ambas cosas.
- Los ***namespace*** y las clases DEBEN seguir el PSR de "***autoloading***" PSR: [PSR-0, PSR-4].
- Los nombres de las clases DEBEN declararse en notación `StudlyCaps` (la primera letra de cada palabra en mayúsculas).
- Las constantes de clase DEBEN declararse en mayúsculas con separadores de subrayado.
- Los nombres de métodos DEBEN declararse en `camelCase` (la primera letra de cada palabra en mayúscula a excepción de la primera palabra)

Archivos

Etiquetas PHP

DEBE usar las etiquetas largas `<?php ?>` o las etiquetas cortas `<?= ?>` .

Codificación de caracteres

DEBE usar solo UTF-8 sin BOM.

Efectos secundarios

Un archivo DEBE declarar símbolos (clases, funciones, constantes, etc.) o lógica secundaría (generar informes, cambiar la configuración, etc...) pero NO DEBE hacer ambas cosas.

La frase "***efectos secundarios***" se refiere a la ejecución de lógica que no está directamente relacionada con la declaración de clases, funciones, constantes, etc.,

Los "***efectos secundarios***" incluyen, entre otros: generar resultados, uso explícito de `require` o `include`, conectarse a servicios externos, modificar la configuración de ini, emitir errores o excepciones, modificar variables globales o estáticas, leer o escribir en un archivo y casos similares.

Sigue un ejemplo de declaraciones y efectos secundarios; es decir, un ejemplo de lo que **se debe evitar**:

```
<?php
```

```
//efecto secundario: cambia la configuración de ini
ini_set('error_reporting', E_ALL);

//efecto secundario: carga un archivo
include "file.php";

//efecto secundario: genera salida
echo "<html>\n";

// declaración
function foo()
{
 //cuerpo de la función
}
```

El siguiente no contiene declaraciones con efectos secundarios; es decir, un ejemplo en la forma propuesta:

```
<?php

//declaración
function foo()
{
 //contenido
}

//la declaración condicional NO es un efecto secundario
if (! function_exists('bar')) {
    function bar()
    {
     //contenido
    }
}
```

Namespace y nombres de clases

Los ***namespace*** y las clases DEBEN seguir el PSR de "***autoloading***" PSR: [PSR-0, PSR-4].

Esto significa que cada clase está en un único archivo y en un ***namespace*** de al menos un nivel.

Los nombres de las clases DEBEN declararse en notación `StudlyCaps` (la primera letra de cada palabra en mayúsculas).

El código escrito para PHP 5.3 y posteriores DEBE usar ***namespace*** formales.

Por ejemplo:

```
<?php

// PHP 5.3 and later:
namespace Vendor\Model;

class Foo
{
}
```

El código escrito para versiones 5.2.x y anteriores DEBERÍA usar la convención de ***pseudo-namespacing*** de los prefijos de `Vendor_` en los nombres de clases.

```
<?php

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

Constantes de clase, propiedades y métodos

El término "*clase*" se refiere a todas las *clases*, *interfaces* y *traits*.

Constantes

Las constantes de clase DEBEN declararse en mayúsculas con separadores de subrayado.

Por ejemplo:

```
<?php

namespace Vendor\Model;

class Foo
{
    const VERSION = '1.0';
    const FECHA_VERSION = '2021-06-16';
}
```

Propiedades

Se evita intencionalmente cualquier recomendación con respecto a los nombres de propiedades.

DEBE aplicar la convención de nomenclatura que decida `$StudlyCaps`, `$camelCase` o `$under_score` de manera coherente dentro de un alcance razonable. Ese alcance puede ser a *vendor-level* (nivel de proveedor), *package-level* (nivel de paquete), *class-level* (nivel de clase) o *method-level* (nivel de método).

Métodos

Los nombres de métodos DEBEN declararse en `camelCase` (la primera letra de cada palabra en mayúscula a excepción de la primera palabra)

PSR 3 - Logger Interface (Interfaz de registros)

Información general

Puede consultar el original en: https://www.php-fig.org/psr/psr-3

Describe una interfaz común para las librerías de registro de acciones que ocurren durante la ejecución de la aplicación.

El objetivo principal es permitir que las librerías reciban un objeto `Psr\Log\LoggerInterface` y escriban registros en él de una manera simple y universal.

Los Frameworks y CMS que tienen necesidades personalizadas PUEDEN extender la interfaz para sus propios fines, pero DEBEN seguir siendo compatibles con este documento. Esto asegura que las bibliotecas de terceros que usa una aplicación puedan escribir en los registros de la aplicación centralizada.

La palabra `implementor` en este documento debe interpretarse como alguien que implementa `LoggerInterface` en una librería o Framework relacionado con el registro. Los usuarios de registradores se denominan `user`.

Especificación

Conceptos básicos

`LoggerInterface` utiliza ocho métodos basados en la RFC 5424 son: ***debug***, ***info***, ***notice***, ***warning***, ***error***, ***critical***, ***alert*** y ***emergency***.

Un noveno método : `log`, que acepta en su primer argumento un nivel de registro.

Llamar a este método `log` con una de las constantes de nivel de registro DEBE tener el mismo resultado que llamar al método específico.

Llamar a este método `log` con un nivel no definido por esta especificación DEBE lanzar una `Psr\Log\InvalidArgumentException` si la implementación no conoce el nivel.

Los usuarios NO DEBEN usar un nivel personalizado sin saber con certeza que la implementación actual lo admite.

Mensaje

Cada método acepta una string como mensaje o un objeto con un método `__toString()`.

El implementador PUEDE manejar de forma especial los objetos recibidos. Si no es el caso, el implementador DEBE convertirlo en una string.

El mensaje PUEDE contener marcadores de posición que las implementaciones PUEDEN reemplazar con valores del array de contexto.

Los nombres de los marcadores de posición DEBEN corresponder a las claves del array de contexto.

Los nombres de los marcadores de posición DEBEN estar delimitados por una sola llave de apertura `{` y una sola llave de cierre `}` . NO DEBE haber ningún espacio en blanco entre los delimitadores y el nombre del marcador de posición.

Los nombres de los marcadores de posición DEBEN estar compuestos únicamente por los caracteres `A-Z`, `a-z`, `0-9`, guión bajo `_` y el punto `.` . El uso de otros caracteres está reservado para futuras modificaciones de la especificación.

Las implementaciones PUEDEN usar marcadores de posición para incorporar varias estrategias de salida y trasladar la información para su visualización. Los usuarios NO DEBEN escapar previamente los valores de marcador de posición, ya que no pueden saber en qué contexto se mostrarán los datos.

A continuación, se muestra un ejemplo, que se proporciona solo con fines de referencia, de implementación de la interpolación de marcadores de posición:

```php
<?php

/**
* Interpola valores de contexto en los marcadores de posición del mensaje.
*/
function interpolate($mensaje, array $contexto = array())
{
  // construye un array de reemplazo con llaves alrededor de las claves de contexto
  $reemplazo = array();
  foreach ($contexto as $key => $val)
  {
   // verifica que el valor se pueda convertir en string
   if (!is_array($val) && (!is_object($val) || method_exists($val, '__toString')))
    {
      $reemplazo['{' . $key . '}'] = $val;
    }
  }
    // interpolar valores de reemplazo en el mensaje y retornar
    return strtr($mensaje, $reemplazo);
}

//un mensaje con nombres de marcador de posición delimitado por llaves
$mensaje = "Usuario {username} creado";

//un array de contexto de nombres => valores de reemplazo
$contexto = array('username' => 'Antoni Gaudí');

//imprime "Usuario Antoni Gaudí creado"
echo interpolate($mensaje, $contexto);
```

Contexto

Cada método acepta un array como datos de contexto. Esto está destinado a contener cualquier información extraña que no encaje bien en una string.

El array puede contener cualquier cosa, el implementador DEBE asegurarse de que tratan los datos de contexto con la mayor indulgencia posible.

Un valor dado en el contexto NO DEBE lanzar una excepción ni generar ningún error, advertencia o aviso de php.

Si se pasa un objeto `Exception` en los datos de contexto, DEBE estar en la clave '`exception`'.

El registro de excepciones es un patrón común y esto permite a los implementadores extraer un seguimiento de la pila de la `exception` cuando el backend del registro lo admite.

Los implementadores DEBEN verificar que la clave de '`exception` ' sea en realidad una `Exception` antes de usarla como tal, ya que PUEDE contener otra información.

Helper classes e interfaces

La clase `Psr\Log\AbstractLogger` le permite implementar `LoggerInterface` muy fácilmente extendiéndola e implementando el método de registro genérico. Los otros ocho métodos le envían el mensaje y el contexto.

De manera similar, el uso de `Psr\Log\LoggerTrait` solo requiere que implemente el método de registro genérico. Tenga en cuenta que, dado que los ***traits*** no pueden implementar interfaces, en este caso aún debe implementar `LoggerInterface`.

`Psr\Log\NullLogger` se proporciona junto con la interfaz. PUEDE ser utilizado por los usuarios de la interfaz para proporcionar una implementación alternativa de "***black hole***" si no se les proporciona un registrador. Sin embargo, el registro condicional puede ser un mejor enfoque si la creación de datos de contexto es costosa.

`Psr\Log\LoggerAwareInterface` solo contiene el método `setLogger (LoggerInterface $logger)` y puede ser utilizado por Frameworks para enlazar automáticamente instancias arbitrarias con un registrador.

El ***trait*** `Psr\Log\LoggerAwareTrait` se puede utilizar para implementar la interfaz equivalente fácilmente en cualquier clase. Da acceso a `$this-> logger`.

La clase `Psr\Log\LogLevel` contiene constantes para los ocho niveles de registro.

Paquete

Las interfaces y clases descritas, así como las clases de excepción relevantes y un conjunto de pruebas para verificar su implementación, se proporcionan como parte del paquete psr/log.

Psr\Log\LoggerInterface

```php
<?php

namespace Psr\Log;

/**
 * Describe un logger instance.
 *
 * El mensaje DEBE ser una string u objeto que implemente __toString().
 *
 * El mensaje PUEDE contener marcadores de posición en la forma: {foo} donde foo
 * será reemplazado por los datos de contexto en la clave "foo".
 *
 * El array de contexto puede contener datos arbitrarios, el único supuesto que
 * pueden hacer los implementadores es que si se proporciona una instancia de Exception
 * para producir un seguimiento de pila, DEBE estar en una clave llamada "exception".
 *
 * Consulte https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-3-logger-interface.md
 * para la especificación completa de la interfaz.
 */
interface LoggerInterface
{
    /**
     * El sistema no se puede utilizar.
     *
     * @param string $message
     * @param array $context
     *
     * @return void
     */
    public function emergency($message, array $context = array());

    /**
     * Se deben tomar medidas de inmediato.
     *
     * Ejemplo: todo el sitio web a caído, la base de datos no está disponible, etc.
     * Esto debería activa las alertas por SMS y avisar.
     *
     * @param string $message
     * @param array $context
     *
     * @return void
     */
    public function alert($message, array $context = array());

    /**
     * Condiciones críticas.

     *
     * Ejemplo: Un componente de la aplicación no disponible, excepción inesperada.
     *
     * @param string $message
     * @param array $context
     *
     * @return void
     */
    public function critical($message, array $context = array());

    /**
     * Errores en tiempo de ejecución que no requieren una acción inmediata, pero que
     * normalmente deberían ser registrados y monitoreados.
     *
     * @param string $message
     * @param array $context
     *
```

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

    /**
     * Incidencias excepcionales que no son errores.
     *
     * Ejemplo: uso de API obsoletas, mal uso de una API, cosas indeseables
     * que no son necesariamente incorrectas.
     *
     * @param string $message
     * @param array $context
     *
     * @return void
     */
    public function warning($message, array $context = array());

    /**
     * Eventos normales pero significativos.
     *
     * @param string $message
     * @param array $context
     *
     * @return void
     */
    public function notice($message, array $context = array());

    /**
     * Eventos interesantes.
     *
     * Ejemplo: Inicio de sesión de usuarios, registros SQL.
     *
     * @param string $message
     * @param array $context
     *
     * @return void
     */
    public function info($message, array $context = array());

    /**
     * Información detallada de depuración.
     *
     * @param string $message
     * @param array $context
     *
     * @return void
     */
    public function debug($message, array $context = array());

    /**
     * Registros con un nivel arbitrario.
     *
     * @param mixed $level es:Psr\Log\LogLevel::DEBUG, Psr\Log\LogLevel::INFO, etc.,
     * @param string $message
     * @param array $context
     *
     * @return void
     */
    public function log($level, $message, array $context = array());
}
```

Psr\Log\LoggerAwareInterface

```
<?php

namespace Psr\Log;

/**
 * Describe una logger-aware (con reconocimiento de registrador).
 * /
 */
```

```
interface LoggerAwareInterface
{
    /**
     * Establece una logger instance (instancia de registrador) en el objeto.
     *
     *
     * @param LoggerInterface $logger
     *
     * @return void
     */
    public function setLogger(LoggerInterface $logger);
}
```

Psr\Log\LogLevel

```
<?php

namespace Psr\Log;

/**
 * Describe los niveles de registro.
 * /
 */
class LogLevel
{
    //Es el error más grave e indica que todo el sistema está inutilizable.
    const EMERGENCY = 'emergency';

    // Se deben tomar medidas inmediatamente. Caída completa de la web, base de datos
    // no disponible etc... En estos casos, se suelen enviar mensajes de aviso por email.
    const ALERT     = 'alert';

    // Situaciones importantes donde se generan excepciones no esperadas o no hay disponible
    // un componente.
    const CRITICAL  = 'critical';

    // Errores de ejecución que permiten continuar con la ejecución de la aplicación pero
    // que deben ser monitorizados.
    const ERROR     = 'error';

    // Incidencias excepcionales que no llegan a ser error.
    const WARNING   = 'warning';

    // Eventos normales pero significativos.
    const NOTICE    = 'notice';

    // Eventos interesantes como el inicio de sesión de usuarios.
    const INFO      = 'info';

    // Información de debug de la aplicación. No usado en entornos de producción.
    const DEBUG     = 'debug';
}
```

PSR 4 - Autoloader (Cargador automático)

Información general

Puede consultar el original en: https://www.php-fig.org/psr/psr-4

Este PSR describe una especificación para clases de autoloading (carga automática) a partir de rutas de archivo.

Es totalmente interoperable y se puede utilizar además de cualquier otra especificación de carga automática, incluida PSR-0.

Este PSR también describe dónde colocar los archivos que se cargarán automáticamente de acuerdo con la especificación.

Especificación

El término "***clase***" se refiere a clases, interfaces, traits y otras estructuras similares.

Un nombre de clase completamente cualificado tiene la siguiente forma:

```
\<NamespaceName>(\<SubNamespaceNames>)*\<ClassName>
```

1. El nombre de clase completamente cualificado completo DEBE tener un nombre de ***namespace*** de nivel superior, también conocido como "***vendor namespace***".
2. El nombre de clase completamente cualificado PUEDE tener uno o más nombres de ***sub-namespace***.
3. El nombre de clase completamente cualificado DEBE terminar con un nombre de clase.
4. Los guiones bajos no tienen un significado especial en ninguna parte del nombre de clase completamente cualificado.
5. Los caracteres alfabéticos en el nombre de clase totalmente cualificados PUEDEN ser cualquier combinación de minúsculas y mayúsculas.
6. Todos los nombres de clases DEBEN ser referenciados de manera ***case-sensitive*** (sensible a mayúsculas y minúsculas).

Al cargar un archivo que corresponde a un nombre de clase completamente cualificado ...

1. Una serie contigua de uno o más ***namespace*** y ***sub-namespace*** iniciales, sin incluir el separador inicial, en el nombre de clase completamente cualificado (un "***namespace prefix***") corresponde al menos a un "***directorio base***".
2. Los nombres de los ***sub-namespace*** contiguos después del "***namespace prefix***" corresponden a un subdirectorio dentro de un "***directorio base***", en el que los separadores de ***namespace*** representan separadores de directorios. El nombre del subdirectorio DEBE coincidir con el nombre de los ***sub-namespace***.
3. El nombre de clase final se corresponde con un nombre de archivo que termina en ***.php***. El nombre del archivo DEBE coincidir con el nombre de la clase.
4. La implementación del ***autoloader*** NO DEBE generar excepciones, NO DEBE generar errores de ningún nivel y NO DEBE devolver un valor.

Ejemplos

La siguiente tabla muestra la ruta de archivo correspondiente para un nombre de clase completamente cualificado, un *namespace* de prefijo y un directorio base:

NOMBRE DE CLASE COMPLETAMENTE CUALIFICADO	NAMESPACE PREFIX	DIRECTORIO BASE	RUTA DEL ARCHIVO RESULTANTE
`\Acme\Log\Writer\File_Writer`	`Acme\Log\Writer`	`./acme-log-writer/lib/`	`./acme-log-writer/lib/File_Writer.php`
`\Aura\Web\Response\Status`	`Aura\Web`	`/path/to/aura-web/src/`	`/path/to/aura-web/src/Response/Status.php`
`\Symfony\Core\Request`	`Symfony\Core`	`./vendor/Symfony/Core/`	`./vendor/Symfony/Core/Request.php`
`\Zend\Acl`	`Zend`	`/usr/includes/Zend/`	`/usr/includes/Zend/Acl.php`

Puede encontrar seguidamente un ejemplo de implementaciones de *autoloader* que cumplen con la especificación.

Implementaciones de ejemplo de PSR-4

Las implementaciones del ejemplo NO DEBEN considerarse parte de la especificación y PUEDEN cambiar en cualquier momento.

Ejemplo original en: https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-4-autoloader-examples.md

Los siguientes ejemplos ilustran el código compatible con PSR-4:

Ejemplo de *Closure* (Cierre)

```
<?php
/**
 * Un ejemplo de implementación específica de un proyecto.
 *
 * Después de registrar esta función de autoload (carga automática) con SPL,
 * la siguiente línea haría que la función intentara cargar la clase \Foo\Bar\Baz\Qux
 * desde /path/to/project/src/Baz/Qux.php:
 *
 *      new \Foo\Bar\Baz\Qux;
 *
 * @param string $class El nombre de clase completamente cualificado.
 *
 * @return void
 */
spl_autoload_register(function ($class) {

    // prefijo del namespace específico del proyecto
    $prefix = 'Foo\\Bar\\';

    // directorio base para el prefijo del namespace
    $base_dir = __DIR__ . '/src/';

    // ¿la clase usa el prefijo del namespace?
    $len = strlen($prefix);
    if (strncmp($prefix, $class, $len) !== 0) {
        // no, pasar al siguiente registered autoloader
        return;
    }

    // obtener el relative class name
    $relative_class = substr($class, $len);

    // reemplaza el prefijo del namespace con el directorio base, reemplaza los separadores
    // del namespace con separadores de directorio en el nombre de la clase relativa, añadir .php
    $file = $base_dir . str_replace('\\', '/', $relative_class) . '.php';

    // si el archivo existe, requerirlo
```

```
    if (file_exists($file)) {
        require $file;
    }
});
```

Ejemplo de clase

La siguiente es una implementación de clase de ejemplo para manejar múltiples *namespace*:

```
<?php
namespace Example;

/**
 * Un ejemplo de implementación de propósito general que incluye la funcionalidad
 * opcional de  permitir múltiples directorios base para un solo prefijo de namespace.
 *
 * Dado un paquete foo-bar de clases en el sistema de archivos en las siguientes
 * rutas ...
 *
 *     /path/to/packages/foo-bar/
 *         src/
 *             Baz.php             # Foo\Bar\Baz
 *             Qux/
 *                 Quux.php        # Foo\Bar\Qux\Quux
 *         tests/
 *             BazTest.php         # Foo\Bar\BazTest
 *             Qux/
 *                 QuuxTest.php    # Foo\Bar\Qux\QuuxTest
 *
 * ... agregue la ruta a los archivos de clase para el prefijo de namespace \Foo\Bar\
 * como sigue:
 *
 *      <?php
 *      // crear una instancia del cargador
 *      $loader = new \Example\Psr4AutoloaderClass;
 *
 *      // registra el autoloader
 *      $loader->register();
 *
 *      // registra los directorios base para el prefijo del namespace
 *      $loader->addNamespace('Foo\Bar', '/path/to/packages/foo-bar/src');
 *      $loader->addNamespace('Foo\Bar', '/path/to/packages/foo-bar/tests');
 *
 * La siguiente línea haría que el autoloader intentara cargar la clase
 * \Foo\Bar\Qux\Quux desde /path/to/packages/foo-bar/tests/Qux/Quux.php:
 *
 *      <?php
 *      new \Foo\Bar\Qux\Quux;
 *
 * La siguiente línea haría que el autoloader intentara cargar la clase
 * \Foo\Bar\Qux\QuuxTest desde /path/to/packages/foo-bar/tests/Qux/QuuxTest.php:
 *
 *      <?php
 *      new \Foo\Bar\Qux\QuuxTest;
 */
class Psr4AutoloaderClass
{
    /**
     * Un array asociativo donde la clave es un prefijo de namespace y el valor
     * es un array de directorios base para clases en ese namespace.
     *
     * @var array
     */
    protected $prefixes = array();

    /**
     * Registrar el autoloader con SPL.
     *
     * @return void
     */
    public function register()
    {
        spl_autoload_register(array($this, 'loadClass'));
```

```
    }

    /**
     * Agrega un directorio base para un namespace prefix.
     *
     * @param string $prefix El namespace prefix.
     * @param string $base_dir Directorio base para archivos de clase en el namespace.
     * @param bool $prepend Si es true, antepone el directorio base a la pila en lugar de agregarlo;
     *     esto hace que se busque primero.
     *
     * @return void
     */
    public function addNamespace($prefix, $base_dir, $prepend = false)
    {
        // normaliza el  namespace prefix
        $prefix = trim($prefix, '\\') . '\\';

        // normaliza el directorio base con un separador final
        $base_dir = rtrim($base_dir, DIRECTORY_SEPARATOR) . '/';

        // inicializar el array de namespace prefix
        if (isset($this->prefixes[$prefix]) === false) {
            $this->prefixes[$prefix] = array();
        }

        // conserva el directorio base para el namespace prefix
        if ($prepend) {
            array_unshift($this->prefixes[$prefix], $base_dir);
        } else {
            array_push($this->prefixes[$prefix], $base_dir);
        }
    }

    /**
     * Carga el archivo de clase para un nombre de clase determinado.
     *
     * @param string $class El nombre de clase completamente cualificado.
     * @return mixed El nombre del archivo mapeado en caso de éxito, o booleano false si falla.
     */
    public function loadClass($class)
    {
        // el actual namespace prefix
        $prefix = $class;

        // buscar hacia atrás a través de los namespace de los nombre de clase
        //completamente cualificados para encontrar un nombre de archivo mapeado
        while (false !== $pos = strrpos($prefix, '\\')) {

            // conserva el separador de namespace final en el prefijo
            $prefix = substr($class, 0, $pos + 1);

            // el resto es el nombre de la clase relativa
            $relative_class = substr($class, $pos + 1);

            // intenta cargar un archivo mapeado para el prefijo y la clase relativa
            $mapped_file = $this->loadMappedFile($prefix, $relative_class);
            if ($mapped_file) {
                return $mapped_file;
            }

            // elimina el separador de namespace final para la próxima iteración
            // de strrpos()
            $prefix = rtrim($prefix, '\\');
        }

        // no ha encontrado un archivo mapeado
        return false;
    }

    /**
     * Carga el archivo mapeado para un namespace prefix y una clase relativa.
     *
     * @param string $prefix El namespace prefix.
     * @param string $relative_class  Enl nombre de clase relativa
     * @return mixed Boolean false si no se puede cargar el archivo mapeado, o el
     *     nombre del archivo cargado.
     */
    protected function loadMappedFile($prefix, $relative_class)
```

```
    {
        // ¿Hay directorios base para este namespace prefix?
        if (isset($this->prefixes[$prefix]) === false) {
            return false;
        }

       // busca en los directorios base el namespace prefix
       foreach ($this->prefixes[$prefix] as $base_dir) {

            // reemplaza el namespace prefix con el directorio base,
            // reemplaza los separadores de namespace con separadores de directorio
            // en el nombre de la clase relativa, anexar .php
            $file = $base_dir
                  . str_replace('\\', '/', $relative_class)
                  . '.php';

            // si el archivo mapeado existe,  lo requiere
            if ($this->requireFile($file)) {
                // si, hemos terminado
                return $file;
            }
        }

      // no encontrado
       return false;
    }

    /**
     * Si existe un archivo, lo requiere del sistema de archivos.
     *
     * @param string $file El archivo requerido.
     * @return bool True si el archivo existe, false si no.
     */
    protected function requireFile($file)
    {
        if (file_exists($file)) {
            require $file;
            return true;
        }
        return false;
    }
}
```

Unit Tests

El siguiente ejemplo es una forma de realizar pruebas unitarias de la clase *autolader* anterior:

```
<?php
namespace Example\Tests;

class MockPsr4AutoloaderClass extends Psr4AutoloaderClass
{
    protected $files = array();

    public function setFiles(array $files)
    {
        $this->files = $files;
    }

    protected function requireFile($file)
    {
        return in_array($file, $this->files);
    }
}

class Psr4AutoloaderClassTest extends \PHPUnit_Framework_TestCase
{
    protected $loader;

    protected function setUp()
    {
        $this->loader = new MockPsr4AutoloaderClass;
```

```
        $this->loader->setFiles(array(
            '/vendor/foo.bar/src/ClassName.php',
            '/vendor/foo.bar/src/DoomClassName.php',
            '/vendor/foo.bar/tests/ClassNameTest.php',
            '/vendor/foo.bardoom/src/ClassName.php',
            '/vendor/foo.bar.baz.dib/src/ClassName.php',
            '/vendor/foo.bar.baz.dib.zim.gir/src/ClassName.php',
        ));

        $this->loader->addNamespace(
            'Foo\Bar',
            '/vendor/foo.bar/src'
        );

        $this->loader->addNamespace(
            'Foo\Bar',
            '/vendor/foo.bar/tests'
        );

        $this->loader->addNamespace(
            'Foo\BarDoom',
            '/vendor/foo.bardoom/src'
        );

        $this->loader->addNamespace(
            'Foo\Bar\Baz\Dib',
            '/vendor/foo.bar.baz.dib/src'
        );

        $this->loader->addNamespace(
            'Foo\Bar\Baz\Dib\Zim\Gir',
            '/vendor/foo.bar.baz.dib.zim.gir/src'
        );
    }

    public function testExistingFile()
    {
        $actual = $this->loader->loadClass('Foo\Bar\ClassName');
        $expect = '/vendor/foo.bar/src/ClassName.php';
        $this->assertSame($expect, $actual);

        $actual = $this->loader->loadClass('Foo\Bar\ClassNameTest');
        $expect = '/vendor/foo.bar/tests/ClassNameTest.php';
        $this->assertSame($expect, $actual);
    }

    public function testMissingFile()
    {
        $actual = $this->loader->loadClass('No_Vendor\No_Package\NoClass');
        $this->assertFalse($actual);
    }

    public function testDeepFile()
    {
        $actual = $this->loader->loadClass('Foo\Bar\Baz\Dib\Zim\Gir\ClassName');
        $expect = '/vendor/foo.bar.baz.dib.zim.gir/src/ClassName.php';
        $this->assertSame($expect, $actual);
    }

    public function testConfusion()
    {
        $actual = $this->loader->loadClass('Foo\Bar\DoomClassName');
        $expect = '/vendor/foo.bar/src/DoomClassName.php';
        $this->assertSame($expect, $actual);

        $actual = $this->loader->loadClass('Foo\BarDoom\ClassName');
        $expect = '/vendor/foo.bardoom/src/ClassName.php';
        $this->assertSame($expect, $actual);
    }
}
```

PSR 5 - Estándar PHPDoc (Borrador)

Información general

PHPDoc define un estándar oficial para comentar código php.

El estándar ofrece un método que anima a los programadores a definir y comentar los aspectos del código que normalmente se ignoran.

Permite que los generadores de documentos externos como phpDocumentor puedan crear la documentación API en formato claro y fácil de entender.

Permite que algunos IDEs como Zend Studio, NetBeans, Aptana Studio y PhpStorm interpreten los tipos de variables y otras ambigüedades en el lenguaje de programación.

Puede consultar el original en: https://github.com/php-fig/fig-standards/blob/master/proposed/phpdoc.md

El propósito principal de este PSR es proporcionar una definición completa y formal del estándar ***PHPDoc***.

Este PSR se desvía de su predecesor, el estándar ***PHPDoc*** de facto asociado con ***phpDocumentor*** 1.x, para proporcionar soporte para funciones más nuevas en el lenguaje PHP y para abordar algunas de las deficiencias de su predecesor.

Este documento NO DEBE:

Describir un estándar para implementar anotaciones a través de ***PHPDoc***. Aunque ofrece una versatilidad que permite crear un PSR posterior basado en las prácticas actuales.

Describir las mejores prácticas o recomendaciones para estándares de codificación sobre la aplicación del estándar ***PHPDoc***. Este documento se limita a una especificación formal de sintaxis e intención.

Convenciones utilizadas en este documento

Las descritas en la RFC 2119

Al final del documento puede encontrar un extracto del RFC 2119 con las palabras mencionadas y su interpretación.

Definiciones

PHPDoc

Es una sección de documentación que proporciona información sobre aspectos de un "***Elemento estructural***".
Es importante tener en cuenta que ***PHPDoc*** y ***DocBlock*** son dos entidades separadas.

DocBlock

Es la combinación de un ***DocComment***, que es un tipo de comentario, y una entidad ***PHPDoc***.
Es la entidad ***PHPDoc*** la que contiene la sintaxis descrita en esta especificación (como la descripción y las etiquetas).

Elemento estructural

Es una colección de construcciones de programación que PUEDEN estar precedidas por un ***DocBlock***.
La colección contiene las siguientes construcciones:

- require(_once)
- include(_once)
- class
- interface
- trait
- function (incluidos los métodos)
- property
- constant
- variables, tanto de ámbito local como global.

Se RECOMIENDA anteponer un *Elemento estructural* con un *DocBlock* donde esté definido y no con cada uso.

Es una práctica común que el *DocBlock* preceda a un *Elemento estructural*, pero también PUEDE estar separado por un número indeterminado de líneas vacías.

Ejemplo:

```
/** @var int $int Esto es un contador. * /
$int = 0;

// no debería haber ningún docblock aquí
$int++;
```

o

```
/**
 * Esta clase actúa como un ejemplo sobre dónde colocar un DocBlock.
 */
class Foo
{
    /** @var string|null $title contiene un título para Foo * /
    protected $title = null;

    /**
     * Establece un título de una sola línea.
     *
     * @param string $title Un texto para el título.
     *
     * @return void
     */
    public function setTitle($title)
    {
        // no debería haber ningún docblock aquí
        $this->title = $title;
    }
}
```

NO SE RECOMIENDA utilizar definiciones compuestas para *Constantes* o *Propiedades*, ya que el manejo de *DocBlock* en estas situaciones puede conducir a resultados inesperados.

Si se utiliza una declaración compuesta, cada elemento DEBE tener un *DocBlock* anterior.

Ejemplo:

```
  class Foo
  {
    protected
      /**
       * @var string Debe contener una descripción
       */
      $name,
      /**
       * @var string Debe contener una descripción
       */
      $description;
  }
```

Un ejemplo de uso que cae más allá del alcance de esta norma es documentar la variable en un *foreach* explícitamente; varios IDE utilizan esta información para ayudar a su función de autocompletar.

Esta norma no cubre este caso específico, ya que cada declaración se considera una declaración de "*Flujo de control*" en lugar de un *Elemento estructural*.

```
/** @var \Sqlite3 $sqlite */
foreach ($connections as $sqlite) {
  // no debería haber ningún docblock aquí
    $sqlite->open('/my/database/path');
```

```
    <...>
}
```

DocComment es un tipo especial de comentario que:

- DEBE comenzar con la secuencia de caracteres `/**` seguida de un carácter de espacio en blanco.
- DEBE terminar con `*/`
- Tener cero o más líneas en el medio.

En el caso de que un *DocComment* abarque varias líneas, cada línea DEBE comenzar con un asterisco (`*`) que DEBE estar alineado con el primer asterisco de la cláusula inicial.

Ejemplo de una sola línea:

```
/** <...> */
```

Ejemplo con multiples líneas:

```
 /**
  * <...>
  */
```

DocBlock

Es un *DocComment* que contiene una única estructura *PHPDoc* y representa la representación básica en el código fuente.

Tag (etiqueta)

Es una pieza única de metainformación con respecto a un *Elemento estructural* o un componente del mismo.

Type (Tipo)

Es la determinación de qué tipo de datos está asociado con un elemento. Esto se usa comúnmente para determinar los valores exactos de argumentos, constantes, propiedades y más.

Consulte el Apéndice A para obtener información más detallada sobre los *type* (tipos).

Vendor-name (namespace superior)

FQSEN

Es una abreviatura de *Fully Qualified Structural Element Name* (Nombre de elemento estructural totalmente calificado).

FQCN

Es una abreviatura de *Fully Qualified Class Name* (Nombre de clase totalmente cualificado)

Esta notación amplía el *FQCN - Fully Qualified Class Name* (Nombre de clase totalmente cualificado) y agrega una notación para identificar miembros de `class/interface/trait` y volver a aplicar los principios del *FQCN* a interfaces, traits, funciones y constantes globales.

Se pueden utilizar las siguientes notaciones por tipo de *Elemento estructural*:

- Namespace: `\My\Space`
- Function: `\My\Space\myFunction()`
- Constant: `\My\Space\MY_CONSTANT`

- Class: `\My\Space\MyClass`
- Interface: `\My\Space\MyInterface`
- Trait: `\My\Space\MyTrait`
- Method: `\My\Space\MyClass::myMethod()`
- Property: `\My\Space\MyClass::$my_property`
- Class Constant: `\My\Space\MyClass::MY_CONSTANT`

Un *FQSEN Fully Qualified Structural Element Name* (Nombre de elemento estructural totalmente calificado) tiene la siguiente definición ABNF:

```
FQSEN    = fqnn/fqcn/constant/method/property  function

fqnn     = "\" [name] *("\" [name])

fqcn     = fqnn "\" name

constant = (fqnn "\" / fqcn "::") name

method   = fqcn "::" name "()"

property = fqcn "::$" name

function = fqnn "\" name "()"

name     = (ALPHA / "_") *(ALPHA / DIGIT / "_")
```

Principios básicos

- Un *PHPDoc* DEBE estar siempre contenido en un *DocComment*; la combinación de estos dos se llama *DocBlock*.
- Un *DocBlock* DEBE preceder directamente a un *Elemento estructural*.

El formato PHPDoc

El formato *PHPDoc* tiene la siguiente definición ABNF:

```
PHPDoc             = [summary] [description] [tags]

summary            = *CHAR (2*CRLF)

description        = 1*(CHAR / inline-tag) 1*CRLF ; cualquier cantidad de caracteres; con etiquetas en
                         línea en el interior

tags               = *(tag 1*CRLF)

inline-tag         = "{" tag "}"

tag                = "@" tag-name [":" tag-specialization] [tag-details]

tag-name           = (ALPHA / "\") *(ALPHA / DIGIT / "\" / "-" / "_")

tag-specialization = 1*(ALPHA / DIGIT / "-")

tag-details        = *SP (SP tag-description / tag-signature )

tag-description    = 1*(CHAR / CRLF)

tag-signature      = "(" *tag-argument ")"

tag-argument       = *SP 1*CHAR [","] *SP
```

En el un capítulo posterior se incluyen ejemplos de uso.

Summary (Resumen)

Un *summary* DEBE contener un resumen del *Elemento estructural* que define el propósito. Se RECOMIENDA que los *summary* abarquen una sola línea o como máximo dos, pero no más.

Un *summary* DEBE terminar con dos saltos de línea secuenciales, a menos que sea el único contenido en *PHPDoc*.

Si se proporciona una *description*, DEBE ir precedida de un *summary*. De lo contrario, la *description* se considerará el *summary*, hasta que se llegue al final del *summary*.

Dado que un *summary* es comparable al título de un capítulo, es beneficioso utilizar el menor formato posible. Como tal, contrariamente a *description* (consulte el capítulo siguiente), no se hace ninguna recomendación para admitir un lenguaje de marcado. Se deja explícitamente a la aplicación de implementación si quiere admitir esto o no.

Description (Descripción)

description es OPCIONAL pero DEBE incluirse cuando el *Elemento estructural*, al que precede este *DocBlock*, contiene más operaciones, u operaciones más complejas, que las que se pueden describir en el *summary*.

Se RECOMIENDA que cualquier aplicación que analice la *description* admita el lenguaje de marcado Markdown para este campo, de modo que el autor pueda proporcionar formato y una forma clara de representar ejemplos de código.

Los usos comunes de *description* son (entre otros):

- Para proporcionar más detalles sobre lo que hace el método que en *summary*.
- Especificar de qué elementos secundarios se compone un array de entrada o salida, o un objeto.
- Proporcionar un conjunto de casos o escenarios de uso común en los que se puede aplicar el *Elemento estructural*.

Tag (Etiqueta)

Las *tag* proporcionan una forma para que los autores proporcionen metadatos concisos sobre el *Elemento estructural* subsiguiente.

Cada *tag* comienza en una nueva línea, seguida de un signo de arroba (`@`) y un nombre de *tag*, seguido de un espacio en blanco y metadatos (incluida una *description*).

Si se proporcionan metadatos, PUEDEN abarcar varias líneas y PODRÍA seguir un formato estricto y, como tal, proporcionar parámetros, según lo dicte el tipo de *tag*. El tipo de *tag* se puede derivar de su nombre.

Por ejemplo:

```
@param string $argumento Este es un parámetro.
```

La *tag* anterior consta de un nombre ('`param`') y metadatos ('`string $argumento Este es un parámetro.`') Donde los metadatos se dividen en un *type* ('`string`'), nombre de variable ('`$argumento`') y *description* ('` Este es un parámetro.` ').

La descripción de una *tag* DEBE admitir Markdown como lenguaje de formato. Debido a la naturaleza de Markdown, es legal comenzar la descripción de la etiqueta en la misma línea o en la siguiente e interpretarla de la misma manera.

Por tanto, estas *tag* son semánticamente idénticas:

```
/**
 * @var string Esta es una descripción.
 * @var string Esta es una
 *    descripción.
 * @var string
 *    Esta es una descripción.
 */
```

Una variación de esto es donde, en lugar de una *description*, se usa una *tag-signature*; en la mayoría de los casos, la *tag* será de hecho una *annotation*". La *tag-signature* puede proporcionar a la *annotation* parámetros relacionados con su funcionamiento.

Si hay una *tag-signature*, NO DEBE haber una *description* presente en la misma *tag*.

Los metadatos proporcionados por las *tag* podrían resultar en un cambio del comportamiento real en tiempo de ejecución del *Elemento estructural* subsiguiente, en cuyo caso el término *annotation* se usa comúnmente en lugar de "*tag*".

Tag Name (Nombre de etiqueta)

Los *Tag name* indican qué tipo de información está representada por esa *tag* o, en el caso de las *annotation* , qué comportamiento debe inyectarse en el *Elemento estructural* siguiente.

En apoyo de las *annotation*, se permite introducir un conjunto de *tag* diseñadas específicamente para una aplicación individual o un subconjunto de aplicaciones (y por lo tanto no están cubiertas por esta especificación).

Estas *tag*, o *annotation* , DEBEN proporcionar un espacio de nombres ya sea prefijando el nombre de la *tag* con un namespace estilo PHP, o con un solo nombre de *vendor-name* seguido de un guión.

Ejemplo de un nombre de *tag* con el prefijo de namespace estilo php (la barra de prefijo es OPCIONAL):

```
@\Doctrine\Orm\Mapping\Entity()
```

Nota: El estándar *PHPDoc* NO hace suposiciones sobre el significado de una *tag* a menos que se especifique en este documento o en adiciones o extensiones posteriores.

Esto significa que PUEDE usar un alias de namespace siempre que se proporcione un elemento de namespace de prefijo.

Por lo tanto, lo siguiente también es legal:

```
@Mapping\Entity()
```

Su propia librería o aplicación puede buscar alias de namespace y hacer un *FQCN* a partir de ellos; esto no tiene ningún impacto en este estándar.

Importante: Las herramientas que utilizan el estándar *PHPDoc* PUEDEN interpretar los namespace que están registrados con esa aplicación y aplicar un comportamiento personalizado.

Ejemplo de un nombre de *tag* con el prefijo del *vendor-name* y un guión:

```
@phpdoc-event transformer.transform.pre
```

Los nombres de *tag* que no tienen el *vendor-name* o un namespace DEBEN describirse en el catálogo de etiquetas PSR y/o en cualquier anexo oficial.

Tag Specialization (Especialización en etiquetas)

Para proporcionar un método mediante el cual proporcionar matices a las *tag* definidas en este estándar, pero sin expandir el conjunto base, se PUEDE proporcionar una especialización después del nombre de la *tag* agregando dos puntos seguidos de una string que proporcione una descripción más matizada de la *tag*.

La lista de especializaciones de *tag* admitidas no se mantiene en el catálogo de etiquetas PSR, ya que puede cambiar con el tiempo.

El PSR puede contener una serie de recomendaciones para nombre de *tag*, pero los proyectos son libres de elegir sus propias especializaciones de *tag*.

Importante: Las herramientas que utilizan el estándar *PHPDoc* PUEDEN interpretar las especializaciones de *tag* que están registradas o comprendidas por esa aplicación y aplican un comportamiento personalizado, pero se espera que implementen el nombre de *tag* anterior como se define en el PSR.

Por ejemplo:

```
@see:unit-test \Mapping\EntityTest::testGetId
```

La *tag* anterior consta de un nombre ('`see`') y una especialización de *tag* ('`unit-test`') y, por lo tanto, define una relación con la prueba unitaria para el método de procedimiento.

Tag Signature (Firma de etiqueta)

Las *tag-signature* se utilizan comúnmente para *annotation* para proporcionar metadatos adicionales específicos de la *tag* actual.

Los metadatos proporcionados pueden influir en el comportamiento de la *annotation* propietaria y, como tal, influir en el comportamiento del *Elemento estructural* subsiguiente.

El contenido de una *tag-signature* debe ser determinado por el *type* de la *tag* (como se describe en *tag-name*) y queda fuera del alcance de esta especificación. Sin embargo, una *tag-signature* NO DEBE ir seguida de una *description* u otra forma de metadatos.

Ejemplos

Los siguientes ejemplos sirven para ilustrar el uso básico de *DocBlock*; Se recomienda leer la lista de *tag* .

Un ejemplo completo podría verse así:

```
/**
 * Este es un summary.
 *
 * Esta es una description. Puede abarcar varias líneas
 * o contener ejemplos de código usando el marcado Markdown
 *
 * @see Markdown
 *
 * @param int        $parameter1 Una description de parámetro.
 * @param \Exception $e          Otra description de parámetro.
 *
 * @\Doctrine\Orm\Mapper\Entity()
 *
 * @return string
 */
function test($parameter1, $e)
{
    ...
}
```

También se permite omitir la *description*:

```
/**
 * Este es un summary.
 *
 * @see Markdown
 *
 * @param int        $parameter1 Una description de parámetro.
 * @param \Exception $parameter2 Otra description de parámetro.
 *
 * @\Doctrine\Orm\Mapper\Entity()
 *
 * @return string
 */
function test($parameter1, $parameter2)
{
    ...
}
```

O incluso omita la sección de *tag* (aunque no se recomienda, ya que le falta información sobre los parámetros y el valor de retorno):

```
/**
 * Este es un summary.
 */
function test($parameter1, $parameter2)
{
    ...
}
```

Un *DocBlock* también puede abarcar una sola línea:

```
/** @var \ArrayObject $array */
public $array = null;
```

Apéndice A - Types ABNF

Un *type* tiene la siguiente definición ABNF:

```
type-expression = type *("|" type) *("&" type)

type  = class-name / keyword / array

array = (type / array-expression) "[]"

array-expression = "(" type-expression ")"

class-name = ["\"] label *("\" label)

label = (ALPHA / %x7F-FF) *(ALPHA / DIGIT / %x7F-FF)

keyword = "array" / "bool" / "callable" / "false" / "float" / "int" / "iterable" / "mixed" / "null" /
          "object" / "resource" / "self" / "static" / "string" / "true" / "void" / "$this"
```

Detalles

Cuando se utiliza un *type*, el usuario esperará un valor, o un conjunto de valores, como se detalla a continuación.

Cuando el *type* consta de varios *type*, estos DEBEN separarse con el signo de barra vertical (`|`) para la unión o el ampersand (`&`) para la intersección. Cualquier intérprete que admita esta especificación DEBE reconocer esto y dividir el *type* antes de evaluarlo.

Ejemplo de *type* de unión:

```
@return int|null
```

Ejemplo de *type* de intersección:

```
@var \MyClass&\PHPUnit\Framework\MockObject\MockObject $myMockObject
```

Arrays

El valor representado por *type* puede ser un array. El *type* DEBE definirse siguiendo el formato de una de las siguientes opciones:

Sin especificar :

No se da una definición del contenido del array representado.

Ejemplo:

```
@return array
```

Especificado Que contiene un solo type :

La definición de ***type*** informa al lector del ***type*** de cada valor del array. Entonces, solo se espera un ***type*** para cada valor en un array dado.

Ejemplo:

```
@return int[]
```

Tenga en cuenta que ***mixed*** también es un ***type*** único y con esta palabra clave es posible indicar que cada valor del array contiene cualquier ***type*** posible.

Especificado como que contiene varios type :

La definición de ***type*** informa al lector del type de cada valor del array. Cada valor puede ser de cualquiera de los ***type*** dados.

Ejemplo:

```
 @return (int|string)[]
```

Class Name

Un ***class-name*** es válido según el contexto en el que se menciona este ***type***. Por lo tanto, puede ser un nombre de clase totalmente calificado (***FQCN***) o un nombre local si está presente en un namespace.

El elemento al que se aplica este ***type*** es una instancia de esa clase o una instancia de una clase que es un (sub) hijo de la clase dada.

Debido a la naturaleza anterior, se RECOMIENDA que las aplicaciones que recopilan y dan forma a esta información muestren una lista de clases secundarias con cada representación de la clase. Esto haría obvio para el usuario qué clases son aceptables como ***type***.

Keyword (Palabra clave)

Una ***keyword*** define el propósito de este type. No todos los elementos están determinados por una clase, pero aún así son dignos de clasificación para ayudar al desarrollador a comprender el código cubierto por ***DocBlock***.

Nota: La mayoría de estas ***keyword*** están permitidas como nombres de clases en PHP y pueden ser difíciles de distinguir de las clases reales. Como tal, las ***keyword*** DEBEN estar en minúsculas, ya que la mayoría de los nombres de clases comienzan con un primer carácter en mayúscula, y NO DEBE usar clases con estos nombres en su código.

Hay más razones para no nombrar clases con los nombres de estas ***keyword***, pero eso queda fuera del alcance de esta especificación.

Este PSR reconoce las siguientes ***keyword***:

bool - el elemento al que se aplica este ***type*** solo tiene estado TRUE o FALSE.

int - el elemento al que se aplica este ***type*** es un número entero o entero.

float - el elemento al que se aplica este ***type*** es un número continuo o real.

string - el elemento al que se aplica este ***type*** es una string de caracteres binarios.

object - el elemento al que se aplica este ***type*** es la instancia de una clase indeterminada.

array - el elemento al que se aplica este *type* es un array de valores.

iterable - el elemento al que se aplica este *type* es un array u objeto Traversable según la definición de PHP.

resource - el elemento al que se aplica este *type* es un recurso según la definición de PHP.

mixed - el elemento al que se aplica este *type* puede ser de cualquier *type* como se especifica aquí. No se sabe en el momento de la compilación qué *type* se utilizará.

void - este *type* se usa comúnmente solo cuando se define el tipo de retorno de un método o función, lo que indica "*no se devuelve nada*", por lo que el usuario no debe confiar en ningún valor devuelto.

Ejemplo 1:

```
/**
 * @return void
 */
function saludar()
{
    echo '¿Como estan ustedes?';
}
```

En el ejemplo anterior, no se especifica ninguna declaración de retorno y, por lo tanto, no se determina el valor de retorno.

Ejemplo 2:

```
/**
 * @param bool $silencio  si es true no hay mensaje
 *
 * @return void
 */
function saludar($silencio)
{
    if ($silencio) {
        return;
    }
    echo '¿Como estan ustedes?';
}
```

En este ejemplo, la función contiene una declaración de retorno sin un valor dado.

Debido a que no hay un valor real especificado, esto también se califica como *type* `void`.

null - el elemento al que se aplica este *type* es un valor `NULL` o, en términos técnicos, no existe.

Una gran diferencia en comparación con `void` es que este *type* se usa en cualquier situación en la que el elemento descrito puede contener en un momento dado un valor `NULL` explícito.

Ejemplo 1:

```
/**
 * @return null
 */
function foo()
{
    echo '¿Como estan ustedes?';
    return null;
}
```

Este *type* se usa comúnmente junto con otro *type* para indicar que es posible que no se devuelve nada.

Ejemplo 2:

```
/**
 * @param bool $ create_new Cuando es true, devuelve una nueva stdClass.
 *
```

```
 * @return stdClass|null
 */
function foo($create_new)
{
    if ($create_new) {
        return new stdClass();
    }
    return null;
}
```

callable - el elemento al que se aplica este *type* es un puntero a una llamada de función. Esto puede ser cualquier tipo de invocable según la definición de PHP.

false o true - el elemento al que se aplica este *type* tendrá el valor `TRUE` o `FALSE`. No se devolverá ningún otro valor.

self - el elemento al que se aplica este *type* es de la misma clase en la que está contenido originalmente el elemento documentado.

Ejemplo:

> El método c está contenido en la clase A. *DocBlock* establece que su valor de retorno es de *type* self. Como tal, el método c devuelve una instancia de la clase A.
>
> Esto puede llevar a situaciones confusas cuando se trata de herencia.

Ejemplo (la situación de ejemplo anterior todavía se aplica):

> La clase B extiende la clase A y no redefine el método c. Como tal, es posible invocar el método c de la clase B.
>
> En esta situación, puede surgir ambigüedad ya que *self* podría interpretarse como clase A o B. En estos casos, self DEBE interpretarse como una instancia de la clase donde está escrito el *DocBlock* que contiene el *type self*.

En los ejemplos anteriores, *self* siempre DEBE referirse a la clase A, ya que se define con el método c en la clase A.

Debido a la naturaleza anterior, se RECOMIENDA que las aplicaciones que recopilan y dan forma a esta información muestren una lista de clases secundarias con cada representación de la clase. Esto haría obvio para el usuario qué clases son aceptables como *type*.

static - el elemento al que se aplica este *type* es de la misma clase en la que está contenido el elemento documentado o, cuando se encuentra en una subclase, es del *type* de esa subclase en lugar de la clase original.

Esta *keyword* se comporta de la misma manera que la *keyword* para el *static binding* (no el método estático, la propiedad ni el modificador de variable) como lo define PHP.

$this - el elemento al que se aplica este *type* es la misma instancia exacta que la clase actual en el contexto dado. Como tal, este *type* es una versión más estricta de *static*, porque la instancia devuelta no solo debe ser de la misma clase sino también de la misma instancia.

Este *type* se utiliza a menudo como valor de retorno para métodos que implementan el patrón de diseño de Fluent Interface.

PSR 6 - Caching Interface (Interfaz de almacenamiento en caché)

Información general

Puede consultar el original en: https://www.php-fig.org/psr/psr-6

El almacenamiento en caché es una forma común de mejorar el rendimiento de cualquier proyecto, lo que hace que las librerías de almacenamiento en caché sean una de las características más comunes de muchos Frameworks y librerías.

Esto ha llevado a una situación en la que muchas librerías implementan sus propias librerías de almacenamiento en caché, con varios niveles de funcionalidad.

Estas diferencias hacen que los desarrolladores tengan que aprender a utilizar varios sistemas que pueden proporcionar o no la funcionalidad que necesitan. Además, los desarrolladores de librerías de almacenamiento en caché se enfrentan a la elección entre admitir solo un número limitado de Frameworks o crear un gran número de clases de adaptadoras.

Una interfaz común para los sistemas de almacenamiento en caché resolverá estos problemas.

Los desarrolladores de librerías y Frameworks pueden contar con que los sistemas de almacenamiento en caché funcionen de la manera que esperan, mientras que los desarrolladores de sistemas de almacenamiento en caché solo tendrán que implementar un único conjunto de interfaces en lugar de una variedad completa de adaptadores.

Objetivo

El objetivo de este PSR es permitir a los desarrolladores crear librerías con memoria caché que se puedan integrar en Frameworks y sistemas existentes sin la necesidad de un desarrollo personalizado.

Definiciones

Key (Clave)

La ***Key*** es un string de al menos un carácter que identifica de forma exclusiva un ***Item*** almacenado en caché.

La implementación DEBE admitir claves que consten de los caracteres `A-Z`, `a-z`, `0-9`, `_` y `.` en cualquier orden en codificación UTF-8 y una longitud de hasta 64 caracteres, la implementación PUEDE admitir caracteres y codificaciones adicionales o longitudes más largas, pero debe admitir al menos ese mínimo.

Las librerías son responsables de su propio escape de cadenas de claves según corresponda, pero DEBEN poder devolver el string de claves original sin modificar.

Los siguientes caracteres están reservados para futuras extensiones y NO DEBEN ser compatibles con las ***Implementing Library*** : `{}` `()` `/` `\` `@`.

Item (Elemento)

Un ***Item*** representa un único par ***Key/value*** (clave/valor) dentro de un ***Pool***. La ***Key*** es el identificador único principal de un ***Item*** y DEBE ser inmutable. El valor PUEDE cambiarse en cualquier momento.

Pool (Grupo)

Pool representa una colección de ***Items*** en un sistema de almacenamiento en caché.

El ***Pool*** es un repositorio lógico de todos los ***Items*** que contiene.

Todos los elementos almacenables en caché se recuperan del ***Pool*** como un objeto ***Item***, y toda la interacción con todo el universo de objetos almacenados en caché ocurre a través del ***Pool***.

Calling Library (Librería de Llamada)

Es la librería o el código que realmente necesita los servicios de caché.

Esta librería utilizará los servicios de almacenamiento en caché que implementan las interfaces de este estándar, pero no tendrá conocimiento de la implementación de esos servicios.

Implementing Library (Librería de Implementación)

Esta librería es la responsable de implementar este estándar para proporcionar servicios de almacenamiento en caché a cualquier *Calling Library* .

La *Implementing Library* DEBE proporcionar clases que implementen las interfaces `Cache\CacheItemPoolInterface` y `Cache\CacheItemInterface`. La implementación DEBE admitir una funcionalidad *TTL* mínima como se describe a continuación con *whole-second granularity*.

TTL - The Time To Live (Tiempo de Vida)

El *TTL* (tiempo de vida de un *Item*) es la cantidad de tiempo entre el momento en que el *Item* se almacena y se considera obsoleto.

El *TTL* normalmente se define mediante un número entero que representa el tiempo en segundos o un objeto `DateInterval`.

Expiration (Caducidad)

Es la hora real en la que un *Item* está configurado para quedar obsoleto.

Por lo general, esto se calcula agregando el *TTL* a la hora en que se almacena un objeto, pero también se puede establecer explícitamente con el objeto `DateTime`.

Un *Item* con un *TTL* de 300 segundos almacenado a la 1:30:00 tendrá una *Expiration* de 1:35:00.

La implementación PUEDE caducar un elemento antes de su tiempo de *Expiration*, pero DEBE tratarlo como caducado una vez alcanzado su tiempo de *Expiration*.

Si una *Calling Library* solicita que se guarde un *Item* pero no especifica un tiempo de *Expiration*, o especifica un *TTL* o valor de *Expiration* `null`, la implementación PUEDE usar una duración predeterminada. Si no se ha establecido una duración predeterminada, la implementación DEBE interpretarlo como una solicitud de almacenamiento en caché perpetuo, o mientras la implementación subyacente lo admita.

Hit (Acierto de caché)

Se produce un *Hit* de caché cuando una *Calling Library* solicita un *Item* por clave y se encuentra un valor coincidente para esa clave, y ese valor no ha caducado y el valor no es inválido por algún otro motivo. Las *Calling Library* DEBEN asegurarse de verificar `isHit()` en todas las llamadas `get()`.

Miss (Error de caché)

Un *miss* de caché es lo opuesto a un *Hit* de caché. Se produce un *miss* cuando una *Calling Library* solicita un *Item* por clave y ese valor no se encuentra para esa clave, o el valor se encontró pero ha expirado, o el valor no es válido por alguna otra razón. Un valor que ha expirado siempre DEBE considerarse un *Miss* de caché.

Deferred (Diferido)

Un *Deferred* indica que un *Item* de caché puede no ser guardado inmediatamente por el *Pool*.

Un objeto *Pool* PUEDE retrasar la persistencia de un *Item* de caché *Deferred* para aprovechar las operaciones de conjuntos masivos, *bulk-set*, compatibles con algunos motores de almacenamiento.

Un *Pool* DEBE garantizar que los elementos de caché *Deferred* se conservan eventualmente y que los datos no se pierden, y PUEDE conservarlos antes de que una *Calling Library* solicite que se conserven.

Cuando una *Calling Library* invoca el método `commit()`, todos los *Items Deferred* pendientes DEBEN conservarse.

Una *Implementing Library* PUEDE usar cualquier lógica apropiada para determinar cuándo conservar los *Items Deferred* , como un destructor de objetos, conservar todos con `save()`, un tiempo de espera o una verificación de *Items* máximos o cualquier otra lógica apropiada. Las solicitudes de un *Item* de caché *Deferred* DEBEN devolver el *Item Deferred* pero aún no persistente.

Datos

La *implementing library* DEBE admitir todos los tipos de datos PHP serializables, incluidos:

String - Cadenas de caracteres de tamaño arbitrario en cualquier codificación compatible con PHP.

Integer - Todos los enteros de cualquier tamaño admitidos por PHP, hasta 64 bits con signo.

Float - Todos los valores de coma flotante.

Boolean - True y False.

Null - El valor null.

Array - Matrices indexadas, asociativas y multidimensionales de profundidad arbitraria.

Object - Cualquier objeto que admita serialización y deserialización sin pérdida de modo que `o == unserialize(serialize($o))`. Los objetos PUEDEN aprovechar la interfaz serializable de PHP, los métodos mágicos `__sleep ()` o `__wakeup ()`, o una funcionalidad del lenguaje similar si corresponde.

Todos los datos pasados a la *Implementing Library* DEBEN devolverse exactamente como se pasaron. Eso incluye el tipo de variable. Es decir, es un error devolver `(string) 5` si `(int) 5` fue el valor guardado.

La *implementing library* PUEDE usar las funciones `serialize()/unserialize()` de PHP internamente, pero no es necesario que lo haga, la compatibilidad con ellas se utiliza simplemente como base para valores de objeto aceptables.

Si no es posible, por cualquier motivo, devolver exactamente el valor guardado, la *implementing library* DEBE responder con un error de caché en lugar de los datos dañados.

Error handling (Manejo de errores)

Si bien el almacenamiento en caché suele ser una parte importante del rendimiento de la aplicación, nunca debería ser una parte crítica de la funcionalidad de la aplicación.

Por lo tanto, un error en un sistema de caché NO DEBE resultar en fallos de la aplicación.

Por esa razón, la *Implementing Library* NO DEBE generar excepciones distintas de las definidas por la interfaz, y DEBE atrapar cualquier error o excepción desencadenada por un almacén de datos subyacente y no permitir que se propaguen.

Una *Implementing Library* DEBE registrar dichos errores o informarlos a un administrador, según corresponda.

Si una *Calling Library* solicita que se eliminen uno o más *Items*, o que se borre un grupo, NO DEBE considerarse una condición de error si la *Key* especificada no existe. La condición posterior es la misma (la *Key* no existe o la *Pool* está vacía), por lo que no hay condición de error.

Interfaces

CacheItemInterface

CacheItemInterface define un *Item* dentro de un sistema de caché.

Cada objeto *Item* DEBE estar asociado con una *Key* específica, que se puede configurar de acuerdo con el sistema de implementación y generalmente es pasada por el objeto `Cache\CacheItemPoolInterface`.

El objeto `Cache\CacheItemInterface` encapsula el almacenamiento y la recuperación de *Items* de la caché.

Cada `Cache\CacheItemInterface` es generado por un objeto caché `CacheItemPoolInterface`, que es responsable de cualquier configuración requerida, así como de asociar el objeto con una *Key* única.

Los objetos `Cache\CacheItemInterface` DEBEN poder almacenar y recuperar cualquier tipo de valor PHP definido en la sección de Datos de este documento.

Las *Calling Library* NO DEBEN crear una instancia de los objetos *Item* en sí mismas. Solo se pueden solicitar desde un objeto *Pool* a través del método *getItem()*.

Las *Calling Library* NO DEBEN asumir que un *Item* creado por una *Implementing Library* es compatible con un *Pool* de otra *Implementing Library* .

```
<?php

namespace Psr\Cache;

/**
 * CacheItemInterface define una interfaz para interactuar con objetos dentro de una caché.
 */
interface CacheItemInterface
{
    /**
     * Devuelve la Key del Item de caché actual.
     *
     * La Key la carga la Implementing Library, pero debería estar disponible para
     * las llamadas de un nivel superior cuando sea necesario.
     *
     * @return string La string de Keys para este Item de caché.
     */
    public function getKey();

    /**
     * Recupera el valor del Item del caché asociado con la Key de ese objeto.
     *
     * El valor devuelto debe ser idéntico al valor almacenado originalmente por set().
     *
     * Si isHit() devuelve false, este método DEBE devolver null. Tenga en cuenta que null
     * es un valor legítimo almacenado en caché, por lo que el método isHit() DEBE usarse para
     * diferenciar entre "se encontró un valor null" y "no se encontró ningún valor".
     *
     * @return mixed The value corresponding to this cache Item's Key, or null if not found.
     */
    public function get();

    /**
     * Confirma si la búsqueda de Items de caché resultó en un caché Hit (acierto).
     *
     * Nota: Este método NO DEBE tener una condición de carrera entre las llamadas a isHit()
     * y a get().
     *
     *
     * @return bool True si la solicitud resultó en un Hit,false en caso contrario.
     */
    public function isHit();

    /**
     * Establece el valor representado por este Item de caché.
     *
     * El argumento $value puede ser cualquier Item que PHP pueda serializar,
     * aunque el método de serialización se deja en manos de la Implementing
     * Library.
     *
     * @param mixed $value El valor serializable que se almacenará.
     *
     * @return static El objeto invocado.
     */
    public function set($value);

    /**
     * Establece el Expiration time (tiempo de vencimiento) de este Item de caché.
     *
     * @param \DateTimeInterface|null $Expiration El momento después del cual el Item DEBE
```

```
     *     considerarse vencido. Si se pasa null explícitamente, PUEDE usarse un valor
     *     predeterminado. Si no se establece ninguno, el valor debe almacenarse de forma
     *     permanente o durante el tiempo que la implementación permita.
     *
     * @return static El objeto llamado.
     */
    public function expiresAt($Expiration);

    /**
     * Establece el Expiration time (tiempo de vencimiento) de este Item de caché.
     *
     * @param int|\DateInterval|null $time El período de tiempo desde el presente después
     *     del cual el Item DEBE ser considerado expired (caducado). Un integer con el tiempo
     *     en segundos hasta el vencimiento. Si se pasa null explícitamente, PUEDE usarse un valor
     *     predeterminado. Si no se establece ninguno, el valor debe almacenarse de forma
     *     permanente o durante el tiempo que la implementación permita.
     *



     * @return static El objeto llamado.
     */
    public function expiresAfter($time);
}
```

CacheItemPoolInterface

El propósito principal de `Cache\CacheItemPoolInterface` es aceptar una *Key* de la *Calling Library* y devolver el objeto `Cache\CacheItemInterface asociado`. Es el punto principal de interacción con toda la colección de caché.

Toda la configuración e inicialización del *Pool* se deja en manos de una *Implementing Library* .

```
<?php

namespace Psr\Cache;

/**
 * CacheItemPoolInterface genera objetos CacheItemInterface.
 */
interface CacheItemPoolInterface
{
    /**
     * Devuelve un Item de caché que representa la Key especificada.
     *
     * Este método siempre debe devolver un objeto CacheItemInterface, incluso en caso de
     * un Miss (fallo de caché). NO DEBE devolver un valor null.
     *
     * @param string $Key La Key para la que devolver el Item correspondiente.
     *
     * @throws InvalidArgumentException Si la string $Key no tiene un valor legal, una
     *     \Psr\Cache\InvalidArgumentException DEBE ser arrojada.
     *
     * @return CacheItemInterface El Item de caché.
     */
    public function getItem($Key);

    /**
     * Devuelve un conjunto de Items de caché traversables.
     *
     * @param string[] $Keys Un array indexado de Keys de Items para recuperar.
     *
     * @throws InvalidArgumentException Si alguna de las Keys en $Keys no tiene un valor legal, una
     *     \Psr\Cache\InvalidArgumentException DEBE ser arrojada.
     *
     * @return array|\Traversable Una colección traversable de Items de caché codificados
     *     por las Keys de cada elemento. Se devolverá un Item por cada Key, incluso si esa Key
     *     no se encuentra. Sin embargo, si no se especifican Keys, DEBE devolverse un traversable vacío.
     */
    public function getItems(array $Keys = array());

    /**
     * Confirma si la caché contiene un Item de caché especificado.
     *
```

```
     * Nota: Este método PUEDE evitar la recuperación del valor en caché por razones de rendimiento.
     * Esto podría resultar en una condición de carrera con CacheItemInterface::get(). Para evitar
     * tal situación use CacheItemInterface::isHit() en su lugar.
     *
     *
     * @param string $Key La Key para la cual verificar la existencia.
     *
     * @throws InvalidArgumentException Si la string $Key no tiene un valor legal, una
     *     \Psr\Cache\InvalidArgumentException DEBE ser arrojada.
     *
     * @return bool True si el Item existe en la caché, false en caso contrario.
     */
    public function hasItem($Key);

    /**
     * Elimina todos los Items del Pool.
     *
     * @return bool True si el Pool se borró correctamente. False si hubo un error.
     */
    public function clear();

    /**
     * Elimina el Item del Pool.
     *
     * @param string $Key La Key para borrar.
     *
     * @throws InvalidArgumentException Si la string $Key no tiene un valor legal, una
     *     \Psr\Cache\InvalidArgumentException DEBE ser arrojada.
     *
     * @return bool True si el Item se eliminó correctamente. False si hubo un error.
     */
    public function deleteItem($Key);

    /**
     * Elimina varios Items del Pool.
     *
     * @param string[] $Keys Un array de Keys que deben eliminarse del Pool.
     *
     * @throws InvalidArgumentException Si alguna de las Keys en $Keys no tiene un valor legal, una
     *     \Psr\Cache\InvalidArgumentException DEBE ser arrojada.
     *
     * @return bool True si los Items se eliminaron correctamente. False si hubo un error.
     */
    public function deleteItems(array $Keys);

    /**
     * Guarda un Item de caché inmediatamente.
     *
     * @param CacheItemInterface $Item El Item de caché para guardar.
     *
     * @return bool True si el Item se guardo correctamente. False si hubo un error.
     */
    public function save(CacheItemInterface $Item);

    /**
     * Establece un Item de caché para que se guarde más tarde.
     *
     * @param CacheItemInterface $Item El Item de caché para guardar.
     *
     * @return bool False si el Item no se pudo poner en cola o si se intentó guardar y falló.
     *     True en caso contrario.
     */
    public function saveDeferred (CacheItemInterface $Item);

    /**
     * Guarda cualquier Item de caché Deferred (diferido).
     *
     * @return bool True si todos los Items aún no guardados se guardaron correctamente o
     *     si no había ninguno. False en caso contrario.
     */
    public function commit();
}
```

CacheException

Esta interfaz de excepción está diseñada para usarse cuando ocurren errores críticos, que incluyen, entre otros, errores en la configuración de la caché, la conexión a un servidor de caché o las credenciales no válidas.

Cualquier excepción lanzada por una `Implementing Library` DEBE implementar esta interfaz.

```php
<?php

namespace Psr\Cache;

/**
 * Interfaz de excepción para todas las excepciones lanzadas por una Implementing Library.
 */
interface CacheException
{
}
```

InvalidArgumentException

```php
<?php

namespace Psr\Cache;

/**
 * Interfaz de excepción para argumentos de caché no válidos.
 *
 * Cada vez que se pasa un argumento inválido a un método, debe lanzar una
 * clase de excepción que implemente Psr\Cache\InvalidArgumentException.
 */
interface InvalidArgumentException extends CacheException
{
}
```

Desde *psr/cache versión 2.0*, las interfaces anteriores se han actualizado para agregar sugerencias de tipo de argumento.

Desde *psr/cache versión 3.0*, las interfaces anteriores se han actualizado para agregar sugerencias de tipo de retorno. Las referencias a `array|\Traversable` se han reemplazado por `iterable`.

PSR 7 - HTTP Message Interface (Interfaz de mensajes HTTP)

Información general

Puede consultar el original en: https://www.php-fig.org/psr/psr-7

Este documento describe interfaces comunes para representar mensajes HTTP como se describe en RFC 7230 y RFC 7231, y URIs para usar con mensajes HTTP como se describe en RFC 3986.

Los mensajes HTTP son la base del desarrollo web. Los navegadores web y los clientes HTTP como *cURL* crean mensajes de solicitud HTTP que se envían a un servidor web, que proporciona un mensaje de respuesta HTTP. El código del lado del servidor recibe un mensaje de solicitud HTTP y devuelve un mensaje de respuesta HTTP.

Los mensajes HTTP generalmente se extraen del usuario final, pero como desarrolladores, generalmente necesitamos saber cómo están estructurados y cómo acceder a ellos o manipularlos para realizar nuestras tareas, ya sea que se trate de una solicitud a una API HTTP. o manejar una solicitud entrante.

Cada mensaje de solicitud HTTP tiene una forma específica:

```
POST /path HTTP/1.1
Host: example.com

foo=bar&baz=bat
```

La primera línea de una solicitud es la *request line* (línea de solicitud) y contiene, en orden, el método de solicitud HTTP, el destino de la solicitud (generalmente un URI absoluto o una ruta en el servidor web) y la versión del protocolo HTTP. A esto le siguen uno o más encabezados HTTP, una línea vacía y el cuerpo del mensaje.

Los mensajes de respuesta HTTP tienen una estructura similar:

```
HTTP/1.1 200 OK
Content-Type: text/plain

Este es el cuerpo de respuesta
```

La primera línea es la *status line* (línea de estado) y contiene, en orden, la versión del protocolo HTTP, el código de estado HTTP y una *frase de motivo*, una descripción legible por humanos del código de estado.

Al igual que el mensaje de solicitud, este es seguido por uno o más encabezados HTTP, una línea vacía y el cuerpo del mensaje.

Las interfaces descritas en este documento son abstracciones en torno a los mensajes HTTP y los elementos que los componen.

Referencias

- RFC 2119
- RFC 3986
- RFC 7230
- RFC 7231

Especificación

Mensajes

Un mensaje HTTP es una solicitud de un cliente a un servidor o una respuesta de un servidor a un cliente.

Esta especificación define interfaces para los mensajes HTTP `Psr\Http\ Message\RequestInterface` y `Psr\Http\Message\ResponseInterface` respectivamente.

Tanto `Psr\Http\Message\RequestInterface` como `Psr\Http\Message\ResponseInterface` extienden `Psr\Http\Message\MessageInterface`.

Mientras que `Psr\Http\Message\MessageInterface` PUEDE implementarse directamente, los implementadores DEBEN implementar `Psr\Http\Message\RequestInterface` y `Psr\Http\Message\ResponseInterface`.

De aquí en adelante, el espacio de nombres `Psr\Http\Message` se omitirá cuando se haga referencia a estas interfaces.

HTTP Headers (Cabeceras HTTP)

Lo nombres de campo del encabezado son insensible a mayúsculas.

Los mensajes HTTP incluyen nombres de campo que son insensible a mayúsculas.

Los encabezados se recuperan por nombre de las clases que implementan `MessageInterface` insensible a mayúsculas.

Por ejemplo, recuperar el encabezado `foo` devolverá el mismo resultado que recuperar el encabezado `FoO`. De manera similar, configurar el encabezado `Foo` sobrescribirá cualquier valor de encabezado `foo` establecido previamente.

```
$message = $message->withHeader('foo', 'bar');

echo $message->getHeaderLine('foo');
// Salida: bar

echo $message->getHeaderLine('FOO');
// Salida: bar

$message = $message->withHeader('fOO', 'baz');

echo $message->getHeaderLine('foo');
// Salida: baz
```

A pesar de que los encabezados pueden recuperarse insensible a mayúsculas, la implementación DEBE conservar el valor original, en particular cuando se recupera con `getHeaders()`.

Las aplicaciones HTTP no conformes pueden depender de un valor determinado, por lo que es útil que un usuario pueda dictar el valor de los encabezados HTTP al crear una solicitud o respuesta.

Encabezados con varios valores

Para acomodar encabezados con múltiples valores y aún así brindar la conveniencia de trabajar con encabezados como `strings`, los encabezados se pueden recuperar de una instancia de `MessageInterface` como un array o una string.

Utilice el método `getHeaderLine()` para recuperar un valor de encabezado como una string que contiene todos los valores de encabezado de un encabezado con nombre insensible a mayúsculas y concatenados con una coma.

Utilice `getHeader()` para recuperar un array de todos los valores de encabezado para un encabezado particular con nombre insensible a mayúsculas.

```
$message = $message
    ->withHeader('foo', 'bar')
    ->withAddedHeader('foo', 'baz');

$header = $message->getHeaderLine('foo');
// $header contiene: 'bar,baz'

$header = $message->getHeader('foo');
// ['bar', 'baz']
```

Nota: No todos los valores de encabezado se pueden concatenar con una coma (por ejemplo, `Set-Cookie`). Cuando se trabaja con dichos encabezados, los consumidores de clases basadas en `MessageInterface` DEBEN confiar en el método `getHeader()` para recuperar dichos encabezados de valores múltiples.

Host header (Encabezado de host)

En las solicitudes, el *Host header* normalmente refleja el componente de host del URI, así como el `host` utilizado al establecer la conexión TCP. Sin embargo, la especificación HTTP permite que el *Host header* difiera en cada uno de los dos.

Durante la construcción, las implementaciones DEBEN intentar establecer el *Host header* a partir de un URI proporcionado si no se proporciona un *Host header*.

`RequestInterface::withUri()` reemplazará, de forma predeterminada, el *Host header* de la solicitud devuelta con un *Host header* que coincida con el componente de `host` de la `UriInterface` pasada.

Puede optar por conservar el estado original del *Host header* pasando `true` para el segundo argumento (`$preserveHost`). Cuando este argumento se establece en `true`, la solicitud devuelta no actualizará el *Host header* del mensaje devuelto, a menos que el mensaje no contenga un *Host header*.

Esta tabla ilustra lo que `getHeaderLine('Host')` devolverá para una solicitud devuelta por `withUri()` con el argumento `$preserveHost` establecido en `true` para varias solicitudes iniciales y URIs.

REQUEST HOST HEADER 1	REQUEST HOST COMPONENT 2	URI HOST COMPONENT 3	RESULT
	foo.com		foo.com
	foo.com	bar.com	foo.com
foo.com		bar.com	foo.com
foo.com	bar.com	baz.com	foo.com

1 Valor del `Host` header antes de la operación.

2 Componente de host del URI compuesto en la solicitud antes de la operación.

3 Componente de host del URI que se inyecta a través de `withUri()`.

Streams

Los mensajes HTTP constan de una línea de inicio, encabezados y un cuerpo.

El cuerpo de un mensaje HTTP puede ser muy pequeño o muy grande.

Intentar representar el cuerpo de un mensaje como una string puede consumir fácilmente más memoria de la deseada porque el cuerpo debe almacenarse completamente en memoria. Intentar almacenar el cuerpo de una solicitud o respuesta en la memoria impediría que el uso de esa implementación pueda funcionar con cuerpos de mensajes grandes.

`StreamInterface` se utiliza para ocultar los detalles de implementación cuando se lee o se escribe un stream de datos.

Para situaciones en las que una `string` sería una implementación de mensaje apropiada, se pueden usar secuencias integradas como `php://memory` y `php://temp`.

`StreamInterface` expone varios métodos que permiten que los streams se lean, escriban y recorran de manera eficaz.

Los streams exponen sus capacidades usando tres métodos: `isReadable()`, `isWritable()` e `isSeekable()`. Estos métodos se pueden utilizar para determinar si una transmisión es capaz de cumplir con sus requisitos.

Cada instancia de transmisión tendrá varias capacidades: puede ser de solo lectura, de solo escritura o de lectura y escritura.

También puede permitir el acceso aleatorio arbitrario (buscando hacia adelante o hacia atrás a cualquier ubicación), o solo el acceso secuencial (por ejemplo, en el caso de un flujo basado en `socket`, `pipe` o `callback`).

Finalmente, `StreamInterface` define un método `__toString ()` para simplificar la recuperación o emisión de todo el contenido del cuerpo a la vez.

A diferencia de las interfaces de solicitud y respuesta, `StreamInterface` no modela la inmutabilidad.

En situaciones en las que se envuelve un stream PHP real, la inmutabilidad es imposible de aplicar, ya que cualquier código que interactúe con el recurso puede cambiar potencialmente su estado (incluida la posición del cursor, el contenido y más).

Nuestra recomendación es que las implementaciones utilicen stream de solo lectura para solicitudes del lado del servidor y respuestas del lado del cliente.

Los consumidores deben ser conscientes del hecho de que la instancia del stream puede ser mutable y, como tal, podría alterar el estado del mensaje; en caso de duda, cree una nueva instancia de transmisión y adjúntela a un mensaje para imponer el estado.

Solicitar destinos y URI

Según RFC 7230, los mensajes de solicitud contienen un ***request-target*** (objetivo de solicitud) como el segundo segmento de la línea de solicitud. El destino de la solicitud puede ser una de las siguientes formas:

- ***origin-form***, que consta de la ruta y, si está presente, la `string` de consulta; esto a menudo se denomina URL relativa. Los mensajes que se transmiten a través de TCP suelen tener formato de origen; Los datos de ***scheme*** and ***authority data*** generalmente solo están presentes a través de variables CGI.

- ***absolute-form*** , que consta del ***scheme***, ***authority*** ("`[user-info@]host[:port]`", donde los elementos entre paréntesis son opcionales), ***path*** (si está presente), ***query string*** (si está presente) y ***fragment*** (si está presente). Esto a menudo se conoce como un URI absoluto y es la única forma para especificar un URI como se detalla en RFC 3986. Este forma se usa comúnmente cuando se realizan solicitudes a proxies HTTP.

- ***authority-form*** , que consta únicamente de ***authority***. Por lo general, esto se usa solo en solicitudes CONNECT, para establecer una conexión entre un cliente HTTP y un servidor proxy.

- ***asterisk-form***, que consta únicamente de la string `*`, y que se utiliza con el método OPTIONS para determinar las capacidades generales de un servidor web.

Aparte de estos destinos de solicitud, a menudo existe una ***effective URL*** (URL efectiva) que está separada del destino de la solicitud.

La ***effective URL*** no se transmite dentro de un mensaje HTTP, pero se utiliza para determinar el protocolo (http/https), el puerto y el nombre de `host` para realizar la solicitud.

La ***effective URL*** está representada por `UriInterface`. `UriInterface` modela HTTP y HTTPS URIs como se especifica en RFC 3986 (el caso de uso principal).

La interfaz proporciona métodos para interactuar con las distintas partes del URI, lo que evitará la necesidad de analizar repetidamente el URI. También especifica un método `__toString()` para convertir el URI en su representación como string.

Al recuperar el ***request-target*** con `getRequestTarget()`, por defecto este método usará el objeto URI y extraerá todos los componentes necesarios para construir el ***origin-form***. El ***origin-form*** es, con mucho, el destino de solicitud más común.

Si un usuario final desea utilizar una de las otras tres formas, o desea anular explícitamente el objetivo de la solicitud, es posible hacerlo con `withRequestTarget()`.

Llamar a este método no afecta al URI, ya que se devuelve desde `getUri()`.

Por ejemplo, un usuario puede querer realizar una solicitud ***asterisk-form*** a un servidor:

```
$request = $request
    ->withMethod('OPTIONS')
    ->withRequestTarget('*')
    ->withUri(new Uri('https://example.org/'));
```

Este ejemplo puede resultar, en última instancia, en una solicitud HTTP con este aspecto:

```
OPTIONS * HTTP/1.1
```

Pero el cliente HTTP podrá usar la URL efectiva (de `getUri()`) para determinar el protocolo, el nombre de `host` y el puerto TCP.

Un cliente HTTP DEBE ignorar los valores de `Uri::getPath()` y `Uri::getQuery()` y, en su lugar, utilizar el valor devuelto por `getRequestTarget()`, que por defecto concatena estos dos valores.

Los clientes que elijan no implementar 1 o más de las 4 formas de ***request-target***, DEBEN usar `getRequestTarget()`. Estos clientes DEBEN rechazar los ***request-target*** que no admiten y NO DEBEN recurrir a los valores de `getUri()`.

`RequestInterface` proporciona métodos para recuperar el ***request-target*** o crear una nueva instancia con el ***request-target*** proporcionado. De forma predeterminada, si no se compone específicamente ningún ***request-target*** en la instancia, `getRequestTarget()` devolverá la forma de origen del URI compuesto (o `"/"` si no se compone ningún URI).

`withRequestTarget ($requestTarget)` crea una nueva instancia con el *request-target* especificado y, por lo tanto, permite a los desarrolladores crear mensajes de solicitud que representan las otras tres formas de *request-target* (*absolute-form*, *authority-form*, y *asterisk-form*). Cuando se usa, la instancia de URI compuesta aún puede ser útil, particularmente en clientes, donde se puede usar para crear la conexión con el servidor.

Server-side Requests (Solicitudes del lado del servidor)

`RequestInterface` proporciona la representación general de un mensaje de solicitud HTTP.

Sin embargo, las solicitudes del lado del servidor necesitan un tratamiento adicional, debido a la naturaleza del entorno del lado del servidor.

El procesamiento del lado del servidor debe tener en cuenta la Interfaz de puerta de enlace común (CGI) y, más específicamente, la abstracción y extensión de PHP de CGI a través de sus API de servidor (SAPI).

PHP ha proporcionado una simplificación en torno a la clasificación de entrada a través de superglobales como:

- `$_COOKIE`, que deserializa y proporciona acceso simplificado a las cookies HTTP.
- `$_GET`, que deserializa y proporciona acceso simplificado a los argumentos de la string de consulta.
- `$_POST`, que deserializa y proporciona acceso simplificado para los parámetros codificados en la URL enviados a través de HTTP POST; genéricamente, se puede considerar el resultado de analizar el cuerpo del mensaje.
- `$_FILES`, que proporciona metadatos serializados sobre la carga de archivos.
- `$_SERVER`, que proporciona acceso a las variables de entorno CGI/SAPI, que comúnmente incluyen el *request-method*, el *scheme*, el URI de solicitud y los encabezados.

`ServerRequestInterface` extiende `RequestInterface` para proporcionar una abstracción en torno a estas superglobales.

Esta práctica ayuda a reducir el acoplamiento de los consumidores con las superglobales y fomenta y promueve la capacidad de probar a los consumidores solicitados.

La solicitud del servidor proporciona una propiedad adicional, *attributes* (atributos), para permitir a los consumidores la capacidad de introspectar, descomponer y comparar la solicitud con las reglas específicas de la aplicación (como coincidencia de **path**, coincidencia de **scheme** , coincidencia de **host**, etc.).

Como tal, la solicitud del servidor también puede proporcionar mensajes entre múltiples consumidores de solicitudes.

Uploaded files (Carga de archivos)

`ServerRequestInterface` especifica un método para recuperar un árbol de *upload files* en una estructura normalizada, con cada hoja una instancia de `UploadedFileInterface`.

La superglobal `$_FILES` tiene algunos problemas bien conocidos cuando se trata de arrays de entradas de archivos.

Como ejemplo, si tiene un formulario que envía una serie de archivos, por ejemplo, el nombre de entrada "`files`", envío de `files[0]` y `files[1]`, PHP lo representará como:

```
array(
    'files' => array(
        'name' => array(
            0 => 'file0.txt',
            1 => 'file1.html',
        ),
        'type' => array(
            0 => 'text/plain',
            1 => 'text/html',
        ),
        /* etc. */
    ),
)
```

en lugar de lo esperado:

```
array(
    'files' => array(
        0 => array(
            'name' => 'file0.txt',
            'type' => 'text/plain',
            /* etc. */
        ),
        1 => array(
            'name' => 'file1.html',
            'type' => 'text/html',
            /* etc. */
        ),
    ),
)
```

El resultado es que los consumidores necesitan conocer este detalle de implementación del lenguaje y escribir código para recopilar los datos para una carga determinada.

Además, existen escenarios en los que `$_FILES` no se rellena cuando se cargan archivos:

- Cuando el método HTTP no es `POST`.
- Al realizar pruebas unitarias.
- Cuando se opera en un entorno que no es SAPI, como ReactPHP.

En tales casos, los datos deberán incorporarse de manera diferente. Como ejemplos:

- Un proceso puede analizar el cuerpo del mensaje para descubrir las cargas de archivos. En tales casos, la implementación puede optar por no escribir las cargas de archivos en el sistema de archivos, sino envolverlas en un stream para reducir la sobrecarga de memoria, E/S y almacenamiento.
- En escenarios de pruebas unitarias, los desarrolladores deben poder copiar y/o simular los metadatos de carga de archivos para validar y verificar diferentes escenarios.

`getUploadedFiles()` proporciona la estructura normalizada para los consumidores.

Se espera que las implementaciones:

- Agreguen toda la información para la carga de un archivo determinado y la utilicen para completar una instancia de `Psr\Http\Message\UploadedFileInterface`.
- Vuelvan a crear la estructura de árbol enviada, con cada hoja siendo la instancia apropiada de `Psr\Http\Message\UploadedFileInterface` para la ubicación dada en el árbol.

La estructura de árbol a la que se hace referencia debe imitar la estructura de nombres en la que se enviaron los archivos.

En el ejemplo más simple, esto podría ser un solo elemento de formulario con nombre enviado como:

```
<input type="file" name="avatar" />
```

En este caso, la estructura en `$_FILES` se vería así:

```
array(
    'avatar' => array(
        'tmp_name' => 'phpUxcOty',
        'name' => 'my-avatar.png',
        'size' => 90996,
        'type' => 'image/png',
        'error' => 0,
    ),
)
```

La forma normalizada devuelta por `getUploadedFiles()` sería:

```
array(
    'avatar' => /* Instancia de UploadedFileInterface */
)
```

En el caso de una entrada que usa la notación del array para el nombre:

```
<input type="file" name="my-form[details][avatar]" />
```

`$_FILES` termina siendo así:

```
array (
    'my-form' => array (
        'name' => array (
            'details' => array (
                'avatar' => 'my-avatar.png',
            ),
        ),
        'type' => array (
            'details' => array (
                'avatar' => 'image/png',
            ),
        ),
        'tmp_name' => array (
            'details' => array (
                'avatar' => 'phpmFLrzD',
            ),
        ),
        'error' => array (
            'details' => array (
                'avatar' => 0,
            ),
        ),
        'size' => array (
            'details' => array (
                'avatar' => 90996,
            ),
        ),
    ),
)
```

Y el árbol correspondiente devuelto por `getUploadedFiles()` debería ser:

```
array(
    'my-form' => array(
        'details' => array(
            'avatar' => /* Instancia de UploadedFileInterface */
        ),
    ),
)
```

En algunos casos, puede especificar un array de archivos:

```
Upload un avatar: <input type="file" name="my-form[details][avatars][]" />
Upload un avatar: <input type="file" name="my-form[details][avatars][]" />
```

(Por ejemplo, los controles de JavaScript pueden generar entradas de carga de archivos adicionales para permitir la carga de varios archivos a la vez).

En tal caso, la implementación de la especificación debe agregar toda la información relacionada con el archivo en el índice dado. La razón es porque `$_FILES` se desvía de su estructura normal en tales casos:

```
array (
    'my-form' => array (
```

```
        'name' => array (
            'details' => array (
                'avatars' => array (
                    0 => 'my-avatar.png',
                    1 => 'my-avatar2.png',
                    2 => 'my-avatar3.png',
                ),
            ),
        ),
        'type' => array (
            'details' => array (
                'avatars' => array (
                    0 => 'image/png',
                    1 => 'image/png',
                    2 => 'image/png',
                ),
            ),
        ),
        'tmp_name' => array (
            'details' => array (
                'avatars' => array (
                    0 => 'phpmFLrzD',
                    1 => 'phpV2pBil',
                    2 => 'php8RUG8v',
                ),
            ),
        ),
        'error' => array (
            'details' => array (
                'avatars' => array (
                    0 => 0,
                    1 => 0,
                    2 => 0,
                ),
            ),
        ),
        'size' => array (
            'details' => array (
                'avatars' => array (
                    0 => 90996,
                    1 => 90996,
                    3 => 90996,
                ),
            ),
        ),
    ),
)
```

El array `$_FILES` anterior correspondería a la siguiente estructura devuelta por `getUploadedFiles()`:

```
array(
    'my-form' => array(
        'details' => array(
            'avatars' => array(
                0 => /* UploadedFileInterface instance */,
                1 => /* UploadedFileInterface instance */,
                2 => /* UploadedFileInterface instance */,
            ),
        ),
    ),
)
```

Los consumidores accederían al índice `1` del array anidado usando:

```
$request->getUploadedFiles()['my-form']['details']['avatars'][1];
```

Debido a que los datos de los archivos cargados son derivados (derivados de `$_FILES` o del cuerpo de la solicitud), un método mutador, `withUploadedFiles()`, también está presente en la interfaz, lo que permite la delegación de la normalización a otro proceso.

En el caso de los ejemplos originales, el consumo se asemeja al siguiente:

```
$file0 = $request->getUploadedFiles()['files'][0];
$file1 = $request->getUploadedFiles()['files'][1];

printf(
    " Recibió los archivos %s y %s",
    $file0->getClientFilename(),
    $file1->getClientFilename()
);

// "Recibió los archivos file0.txt y file1.html"
```

Esta propuesta también reconoce que las implementaciones pueden operar en entornos que no son SAPI. Como tal, `UploadedFileInterface` proporciona métodos para garantizar que las operaciones funcionen independientemente del entorno.

En particular:

- `moveTo($targetPath)` se proporciona como una alternativa segura y recomendada para llamar a `move_uploaded_file()` directamente en el archivo de carga temporal. Las implementaciones detectarán el funcionamiento correcto a utilizar en función del entorno.

- `getStream()` devolverá una instancia de `StreamInterface`. En entornos que no son de SAPI, una posibilidad propuesta es analizar archivos de carga individuales en `php://temp` streams en lugar de directamente en los archivos; en tales casos, no hay ningún archivo de carga. Por lo tanto, se garantiza que `getStream()` funcionará independientemente del entorno.

Como ejemplos:

```
// Mover un archivo a un directorio de carga
$filename = sprintf(
    '%s.%s',
    create_uuid(),
    pathinfo($file0->getClientFilename(), PATHINFO_EXTENSION)
);
$file0->moveTo(DATA_DIR . '/' . $filename);


// Transmita un archivo a Amazon S3.
// Suponga que $s3wrapper es un stream PHP que escribirá en S3, y que
// Psr7StreamWrapper es una clase que decorará una StreamInterface como PHP
// StreamWrapper.

$stream = new Psr7StreamWrapper($file1->getStream());
stream_copy_to_stream($stream, $s3wrapper);
```

Package (Paquete)

Las interfaces y clases descritas se proporcionan como parte del paquete psr/http-message.

Interfaces

Psr\Http\Message\MessageInterface

```
<?php
namespace Psr\Http\Message;

/**
 * Los mensajes HTTP consisten en solicitudes de un cliente a un servidor y respuestas
 * de un servidor a un cliente. Esta interfaz define los métodos comunes.
 *
 * Los mensajes se consideran inmutables; todos los métodos que pueden cambiar de estado DEBEN
```

```
 * implementarse de manera que conserven el estado interno del mensaje en curso
 * y devuelvan una instancia que contiene el estado cambiado.
 *
 * @see http://www.ietf.org/rfc/rfc7230.txt
 * @see http://www.ietf.org/rfc/rfc7231.txt
 */
interface MessageInterface
{
    /**
     * Recupera la versión del protocolo HTTP como una string.
     *
     * La string DEBE contener solo el número de versión HTTP (por ejemplo, "1.1", "1.0").
     *
     * @return string HTTP protocol version.
     */
    public function getProtocolVersion();

    /**
     * Devuelve una instancia con la versión del protocolo HTTP especificado.
     *
     * La string de versión DEBE contener solo el número de versión HTTP (p. Ej.,
     * "1.1", "1.0").
     *
     * Este método DEBE implementarse de tal manera que retenga la
     * inmutabilidad del mensaje, y DEBE devolver una instancia que tenga la
     * nueva versión del protocolo.
     *
     * @param string $version versión de HTTP protocol
     *
     * @return static
     */
    public function withProtocolVersion($version);

    /**
     * Recupera todos los valores del encabezado del mensaje.
     *
     * Las keys (claves) representan el nombre del encabezado, y
     * cada valor es un array de strings asociadas con el encabezado.
     *
     *     // Representa los encabezados como una string
     *     foreach ($message->getHeaders() as $name => $values) {
     *         echo $name . ': ' . implode(', ', $values);
     *     }
     *
     *     // Emite encabezados de forma iterativa:
     *     foreach ($message->getHeaders() as $name => $values) {
     *         foreach ($values as $value) {
     *             header(sprintf('%s: %s', $name, $value), false);
     *         }
     *     }
     *
     * Si bien los nombres de los encabezados son insensible a mayúsculas, getHeaders() conservará
     * el modo exacto en el que se especificaron originalmente los encabezados.
     *
     * @return string[][]Devuelve un array asociativo de los encabezados del mensaje.
     *     Cada key (clave) DEBE ser un nombre de encabezado, y cada valor DEBE ser un array de
     *     strings para ese encabezado.
     */
    public function getHeaders();

    /**
     * Comprueba si existe un encabezado con el nombre insensible a mayúsculas.
     *
     * @param string $name Nombre de campo del encabezado dado insensible a mayúsculas.
     *
     * @return bool Devuelve true si algún nombre de encabezado coincide con el encabezado dado
     *     usando una comparación de string insensible a mayúsculas.
     *     Devuelve false si no se encuentra ningún nombre de encabezado coincidente en el mensaje.
     */
    public function hasHeader($name);

    /**
     * Recupera un valor de encabezado de mensaje por el nombre dado insensible a mayúsculas.
     *
     * Este método devuelve un array de todos los valores de encabezado de los
     * nombre de encabezado insensible a mayúsculas.
     *
     * Si el encabezado no aparece en el mensaje, este método DEBE devolver un
```

```
     * array vacío.
     *
     * @param string $name Nombre de campo insensible a mayúsculas.
     *
     * @return string[] Un aray de valores de cadena como se proporciona para el
     *     encabezado. Si el encabezado no aparece en el mensaje, este método DEBE
     *     devuelve un array vacío.
     */
    public function getHeader($name);

    /**
     * Recupera una string de valores separados por comas para un solo encabezado.
     *
     * Este método devuelve todos los valores de encabezado de los nombres de encabezado
     * insensible a mayúsculas como una string concatenada mediante comas.
     *
     * NOTA: Es posible que no todos los valores de encabezado se representen correctamente
     * utilizando concatenación de comas. Para tales encabezados, use getHeader() en su lugar
     * y proporcione su propio delimitador al concatenar.
     *
     * Si el encabezado no aparece en el mensaje, este método DEBE regresar
     * una string vacía.
     *
     * @param string $name Nombre de campo de encabezado insensible a mayúsculas.
     *
     * @return string Una string de valores como se proporciona para el encabezado dado
     *     concatenados juntos usando comas. Si el encabezado no aparece en
     *     el mensaje, este método DEBE devolver una string vacía.
     */
    public function getHeaderLine($name);

    /**
     * Devuelve una instancia con el valor proporcionado reemplazando el encabezado especificado.
     *
     * Si bien los nombres de los encabezados son insensibles a mayúsculas, las mayúsculas del encabezado
     * tienen que preservarse y devolverse desde getHeaders().
     *
     * Este método DEBE implementarse de tal manera que retenga la
     * inmutabilidad del mensaje, y DEBE devolver una instancia que tenga
     * el valor nuevo y/o actualizado.
     *
     * @param string $name Nombre de campo de encabezado que insensible a mayúsculas.
     * @param string|string[] $value Valor(es) de encabezado.
     *
     * @return static
     *
     * @throws \InvalidArgumentException para nombres o valores de encabezado no válidos.
     */
    public function withHeader($name, $value);

    /**
     * Devuelve una instancia con el encabezado especificado adjunto con el valor dado.
     *
     * Se mantendrán los valores existentes para el encabezado especificado. El nuevo
     * valor(es) se agregarán a la lista existente. Si el encabezado no
     * existe previamente, se agregará.
     *
     * Este método DEBE implementarse de tal manera que retenga la
     * inmutabilidad del mensaje, y DEBE devolver una instancia que tenga el
     * nuevo encabezado y/o valor.
     *
     * @param string $name Nombre de campo de encabezado que se agrega insensible a mayúsculas.
     * @param string|string[] $value Valor(es) de encabezado.
     *
     * @return static
     *
     * @throws \InvalidArgumentException para nombres de encabezados no válidos.
     * @throws \InvalidArgumentException para valores de encabezado no válidos.
     */
    public function withAddedHeader($name, $value);

    /**
     * Devuelve una instancia sin el encabezado especificado.
     *
     * La resolución del encabezado DEBE realizarse insensible a mayúsculas
     *
     * Este método DEBE implementarse de tal manera que retenga la
     * inmutabilidad del mensaje, y DEBE devolver una instancia que elimine
```

```
     * el encabezado nombrado.
     *
     * @param string $name Nombre del campo de encabezado a eliminar, insensible a mayúsculas.
     *
     * @return static
     */
    public function withoutHeader($name);

    /**
     * Obtiene el cuerpo del mensaje.
     *
     * @return StreamInterface Devuelve el cuerpo como un stream.
     */
    public function getBody();

    /**
     * Devuelve una instancia con el cuerpo del mensaje especificado.
     *
     * El cuerpo DEBE ser un objeto StreamInterface.
     *
     * Este método DEBE implementarse de tal manera que retenga la
     * inmutabilidad del mensaje, y DEBE devolver una nueva instancia que tenga
     * el nuevo stream.
     *
     * @param StreamInterface $body Cuerpo.
     *
     * @return static
     *
     * @throws \InvalidArgumentException Cuando el cuerpo no es válido.
     */
    public function withBody(StreamInterface $body);
}
```

Psr\Http\Message\RequestInterface

```
<?php
namespace Psr\Http\Message;

/**
 * Representación de una solicitud client-side (lado del cliente).
 *
 * Según la especificación HTTP, esta interfaz incluye propiedades para
 * cada uno de las siguientes:
 *
 * - Protocol version
 * - HTTP method
 * - URI
 * - Headers
 * - Message body
 *
 * Durante la construcción, las implementaciones DEBEN intentar configurar el encabezado
 * del host desde un URI proporcionado si no se proporciona un encabezado de host.
 *
 * Las solicitudes se consideran inmutables; todos los métodos que pueden
 * cambiar de estado DEBEN implementarse de manera que conserven el estado interno
 * del mensaje y devuelvan una instancia que contiene el estado cambiado.
 */
interface RequestInterface extends MessageInterface
{
    /**
     * Recupera el destino de la request-target (solicitud de mensaje).
     *
     * Recupera el destino de la request-target tal como aparecerá (para
     * clientes), como apareció en la petición (para servidores), o como estaba
     * especificada para la instancia (ver withRequestTarget()).
     *
     * En la mayoría de los casos, esta será la forma origin-form del URI compuesto,
     * a menos que se haya proporcionado un valor a la implementación concreta (ver
     * withRequestTarget() a continuación).
     *
     * Si no hay ningún URI disponible y no se ha especificado ningún request-target,
     * este método DEBE devolver la string "/".
     *
```

```
     * @return string
     */
    public function getRequestTarget();

    /**
     * Devuelve una instancia con el request-target específico.
     *
     * Si la solicitud necesita un non-origin-form request-target, por ejemplo, para
     * especificar un absolute-form, authority-form, o asterisk-form, este método
     * se puede utilizar para crear una instancia con el request-target especificado, literalmente.
     *
     * Este método DEBE implementarse de tal manera que retenga la
     * inmutabilidad del mensaje, y DEBE devolver una instancia que tenga el
     * request-target modificado.
     *
     * @see http://tools.ietf.org/html/rfc7230#section-5.3 (para las diversas
     *     formas request-target en request_messages)
     *
     * @param mixed $requestTarget
     *
     * @return static
     */
    public function withRequestTarget($requestTarget);

    /**
     * Recupera el método HTTP de la solicitud.

     *
     * @return string Devuelve el método de la solicitud.
     */
    public function getMethod();

    /**
     * Devuelve una instancia con el método HTTP proporcionado.
     *
     * Si bien los nombres de los métodos HTTP suelen ser todos con caracteres en mayúscula,
     * los nombres de los métodos HTTP distinguen entre mayúsculas y minúsculas y, por lo tanto,
     * las implementaciones NO DEBEN modificar la string dada.
     *
     * Este método DEBE implementarse de tal manera que retenga la
     * inmutabilidad del mensaje, y DEBE devolver una instancia que tenga el
     * método solicitado modificado.
     *
     * @param string $method Método insensible a mayúsculas.
     *
     * @return static
     *
     * @throws \InvalidArgumentException para métodos HTTP no válidos.
     */
    public function withMethod($method);

    /**
     * Recupera la instancia de URI.
     *
     * Este método DEBE devolver una instancia de UriInterface.
     *
     * @see http://tools.ietf.org/html/rfc3986#section-4.3
     *
     * @return UriInterface Devuelve una instancia de UriInterface
     *     que representa el URI de la solicitud.
     */
    public function getUri();

    /**
     * Devuelve una instancia con el URI proporcionado.
     *
     * Este método DEBE actualizar el encabezado del host de la solicitud devuelta
     * predeterminada si el URI contiene un componente de host. Si el URI no
     * contiene un componente de host, DEBE devolver cualquier encabezado de host preexistente.
     *
     * Puede optar por conservar el estado original del encabezado del host al
     * establecer '$preserveHost' en 'true'. Cuando '$preserveHost' se establece en
     * 'true', este método interactúa con el encabezado del host de las siguientes formas:
     *
     * - Si el encabezado del host falta o está vacío, y el nuevo URI contiene
     *   un componente del host, este método DEBE actualizar el encabezado del host la
     *   solicitud.
     * - Si el encabezado del host falta o está vacío, y el nuevo URI no contiene un
```

```
     *   componente de host, este método NO DEBE actualizar el encabezado del host en la
     *   solicitud.
     * - Si un encabezado de host está presente y no está vacío, este método NO DEBE actualizar
     *   el encabezado del Host en la solicitud devuelta.
     *
     * Este método DEBE implementarse de tal manera que retenga la
     * inmutabilidad del mensaje, y DEBE devolver una instancia que tenga la
     * nueva instancia de UriInterface.
     *
     * @see http://tools.ietf.org/html/rfc3986#section-4.3
     *
     * @param UriInterface $uri Nueva solicitud de URI para usar.
     * @param bool $preserveHost Conserva el estado original del encabezado del host.
     *
     * @return static
     */
    public function withUri(UriInterface $uri, $preserveHost = false);
}
```

Psr\Http\Message\ServerRequestInterface

```
<?php
namespace Psr\Http\Message;

/**
 * Representación de una solicitud server-side (lado del servidor) HTTP.
 *
 * Según la especificación HTTP, esta interfaz incluye propiedades para
 * cada uno de las siguientes:
 *
 * - Protocol version
 * - HTTP method
 * - URI
 * - Headers
 * - Message body
 *
 * Además, encapsula todos los datos a medida que llegan a la
 * aplicación desde el entorno CGI y/o PHP, que incluye:
 *
 * - Los valores representados en $_SERVER.
 * - Cualquier cookie proporcionada (generalmente a través de $_COOKIE)
 * - Argumentos de la string de consulta (generalmente a través de $_GET, o analizados a través de
 *   parse_str())
 * - Cargar archivos, si los hay (representados por $_FILES)
 * - Parámetros deserializados del cuerpo (generalmente de $_POST)
 *
 * Los valores de $_SERVER DEBEN tratarse como inmutables, ya que representan el estado
 * de la aplicación en el momento de la solicitud; como tal, no se proporcionan métodos
 * para permitir modificalos. Los otros valores proporcionan tales métodos, ya que se
 * pueden restaurar desde $_SERVER o el cuerpo de la solicitud, y es posible que
 * necesiten tratarse durante la aplicación (p. ej., los parámetros del cuerpo se pueden
 * deserializar en función del tipo de contenido).
 *
 * Además, esta interfaz reconoce la utilidad de introspectar una
 * solicitud para derivar y hacer coincidir parámetros adicionales (por ejemplo, a través de
 * URI path matching, decrypting cookie values, deserializing non-form-encoded body
 * content, matching authorization headers to users, etc). Esos parámetros se guardan
 * en la propiedad "attributes".
 *
 * Las solicitudes se consideran inmutables; todos los métodos que pueden cambiar de estado
 * DEBEN implementarse de manera que conserven el estado interno del mensaje y devuelver
 * una instancia que   contiene el estado cambiado.
 */
interface ServerRequestInterface extends RequestInterface
{
    /**
     * Recuperar los parámetros del servidor.
     *
     * Recupera datos relacionados con el entorno de solicitud entrante,
     * típicamente derivado del superglobal $_SERVER de PHP. Los datos NO SON
     * REQUERIDO para originarse desde $_SERVER.
     *
     * @return array
```

```
     */
    public function getServerParams();

    /**
     * Recuperar cookies.
     *
     * Recupera las cookies enviadas por el cliente al servidor.
     *
     * Los datos DEBEN ser compatibles con la estructura superglobal $_COOKIE.
     *
     * @return array
     */
    public function getCookieParams();

    /**
     * Devuelve una instancia con las cookies especificadas.
     *
     * NO SE REQUIERE que los datos provengan de la superglobal $_COOKIE, pero DEBEN
     * ser compatible con la estructura de $_COOKIE. Normalmente, estos datos se
     * inyectan en la instanciación.
     *
     * Este método NO DEBE actualizar el encabezado de la cookie relacionado
     * con la solicitud, ni valores relacionados en los parámetros del servidor.
     *
     * Este método DEBE implementarse de tal manera que retenga la
     * inmutabilidad del mensaje, y DEBE devolver una instancia que tenga los
     * valores de cookies actualizados.
     *
     * @param array $cookies Array de pares key/value (clave/valor) que representan cookies.
     *
     * @return static
     */
    public function withCookieParams(array $cookies);

    /**
     * Recuperar argumentos de la query string (string de consulta).
     *
     * Recupera, si los hay, los argumentos de la query string deserializados.
     *
     * Nota: es posible que los parámetros de consulta no estén sincronizados con el URI
     * o los parámetros del servidor. Si necesita asegurarse de que solo obtiene los
     * valores originales, es posible que deba analizar la query string de
     * 'getUri()->getQuery()'o desde el parámetro del servidor 'QUERY_STRING'.
     *
     * @return array
     */
    public function getQueryParams();

    /**
     * Devuelve una instancia con los argumentos de query string especificados.
     *
     * Estos valores DEBERÍAN permanecer inmutables durante el transcurso de la petición
     * solicitada. PUEDEN inyectarse durante la instanciación, como desde la superglobal de
     * PHP $_GET, o PUEDE derivarse de algún otro valor como el URI. En los casos en que
     * los argumentos se analizan a partir del URI, los datos DEBEN ser compatible con
     * lo que devolvería parse_str() de PHP a propósito de cómo maneja los parámetros
     * de consulta duplicados y cómo se anidan.
     *
     * La configuración de los argumentos de la query string NO DEBE cambiar
     * el URI almacenado por la solicitud, ni los valores en los parámetros del servidor.
     *
     * Este método DEBE implementarse de tal manera que retenga la
     * inmutabilidad del mensaje, y DEBE devolver una instancia que tenga los
     * argumentos de la query string actualizados.
     *
     * @param array $query Array de argumentos de la query string, normalmente de $_GET.
     *
     * @return static
     */
    public function withQueryParams(array $query);

    /**
     * Recuperar datos normalizados de la carga de archivos.
     *
     * Este método devuelve los metadatos de carga en un árbol normalizado, con cada hoja
     * una instancia de Psr\Http\Message\UploadedFileInterface.
     *
     * Estos valores PUEDEN prepararse a partir de $_FILES o del cuerpo del mensaje durante
```

```
     * la instanciación, o PUEDE inyectarse a través de withUploadedFiles().
     *
     * @return array Un árbol del array de instancias de UploadedFileInterface;
     *     DEBE devolverse un array vacío si no hay datos presentes.
     */
    public function getUploadedFiles();

    /**
     * Crea una nueva instancia con los archivos cargados especificados.
     *
     * Este método DEBE implementarse de tal manera que retenga la
     * inmutabilidad del mensaje, y DEBE devolver una instancia que tenga los
     * parámetros del cuerpo actualizados.
     *
     * @param array $uploadedFiles Un árbol de array de instancias de UploadedFileInterface.
     *
     * @return static
     *
     * @throws \InvalidArgumentException si se proporciona una estructura no válida.
     */
    public function withUploadedFiles(array $uploadedFiles);

    /**
     * Recuperar los parámetros proporcionados en el cuerpo de la solicitud.
     *
     * Si el Content-Type de la solicitud es application/x-www-form-urlencoded
     * o multipart/form-data, y el método de solicitud es POST, este método DEBE
     * devolver el contenido de $_POST.
     *
     * De lo contrario, este método puede devolver cualquier resultado de deserialización
     * del contenido del cuerpo de la solicitud; a medida que el análisis devuelve contenido
     * estructurado, los type potenciales DEBEN ser arrays u objetos únicamente.
     * Un valor null indica la ausencia de contenido en el cuerpo.
     *
     * @return null|array|object Los parámetros, si los hay, del cuerpo deserializados.
     *     Por lo general, serán un array u objeto.
     */
    public function getParsedBody();

    /**
     * Devuelve una instancia con los parámetros de cuerpo especificados.
     *
     * Estos PUEDEN inyectarse durante la instanciación.
     *
     * Si el Content-Type de la solicitud es application/x-www-form-urlencoded
     * o multipart/form-data, y el método de solicitud es POST, use este método
     * SOLO para inyectar el contenido de $_POST.
     *
     * NO SE REQUIERE que los datos provengan de $_POST, pero DEBEN ser los resultados de la
     * deserialización del contenido del cuerpo de la solicitud. Deserialización/parsing
     * devuleve datos estructurados y, como tal, este método SOLO acepta arrays u objetos,
     * o un valor null si no había nada disponible para analizar.
     *
     * Como ejemplo, si la negociación de contenido determina que los datos de la solicitud
     * es una carga útil JSON, este método podría usarse para crear una solicitud
     * con los parámetros deserializados.
     *
     * Este método DEBE implementarse de tal manera que retenga la
     * inmutabilidad del mensaje, y DEBE devolver una instancia que tenga los
     * parámetros del cuerpo actualizados.
     *
     * @param null|array|object $data Los datos del cuerpo deserializados.
     *     Tipicamente en un array u objeto.
     *
     * @return static
     *
     * @throws \InvalidArgumentException si un tipo de argumento no admitido es recibido.
     */
    public function withParsedBody($data);

    /**
     * Recuperar atributos derivados de la solicitud.
     *
     * Los "attributes" de la solicitud se pueden utilizar para permitir la inyección de cualquier
     * parámetro derivado de la solicitud: por ejemplo, el path de las operaciones
     * emparejamiento; los resultados de descifrar las cookies; Los resultados de
     * deserialización de mensajes non-form-encoded; etc.
     * "attributes" será específico de la aplicación y la solicitud, y PUEDE ser mutable.
```

```
     *
     * @return mixed[] Attributes derivados de la solicitud.
     */
    public function getAttributes();

    /**
     * Recuperar un solo atributo de solicitud derivado.
     *
     * Recupera un solo atributo de solicitud derivado como se describe en
     * getAttributes(). Si el atributo no se ha configurado previamente, devuelve
     * el valor predeterminado proporcionado.
     *
     * Este método evita la necesidad de un método hasAttribute(), ya que permite
     * especificar un valor predeterminado para devolver si no se encuentra el atributo.
     *
     * @see getAttributes()
     *
     * @param string $name El nombre del atributo.
     * @param mixed $default Valor predeterminado que se devolverá si el atributo no existe.
     *
     * @return mixed
     */
    public function getAttribute($name, $default = null);

    /**
     * Devuelve una instancia con el atributo de solicitud derivado especificado.
     *
     * Este método permite establecer un único atributo de solicitud derivado como
     * descrito en getAttributes().
     *
     * Este método DEBE implementarse de tal manera que retenga la
     * inmutabilidad del mensaje, y DEBE devolver una instancia que tenga el
     * atributo actualizado.
     *
     * @see getAttributes()
     *
     * @param string $name El nombre del atributo.
     * @param mixed $value El valor del atributo.
     *
     * @return static
     */
    public function withAttribute($name, $value);

    /**
     * Devuelve una instancia que elimina el atributo de solicitud derivado especificado.
     *
     * Este método permite eliminar un solo atributo de solicitud derivado como
     * descrito en getAttributes().
     *
     * Este método DEBE implementarse de tal manera que retenga la
     * inmutabilidad del mensaje, y DEBE devolver una instancia que elimine
     * el atributo.
     *
     * @see getAttributes()
     *
     * @param string $name El nombre del atributo.
     *
     * @return static
     */
    public function withoutAttribute($name);
}
```

Psr\Http\Message\ResponseInterface

```
<?php
namespace Psr\Http\Message;

/**
 * Representación de una server-side response (respuesta del lado del servidor saliente).
 *
 * Según la especificación HTTP, esta interfaz incluye propiedades para
 * cada uno de las siguientes:
 *
```

```
 * - Protocol version
 * - Status code and reason phrase
 * - Headers
 * - Message body
 *
 * Las respuestas se consideran inmutables; todos los métodos que pueden cambiar de estado DEBEN
 * implementarse de manera que conserven el estado interno actual del
 * mensaje y devuelve una instancia que contiene el estado cambiado.

 */
interface ResponseInterface extends MessageInterface
{
    /**
     * Obtiene el status code (código de estado) de respuesta.
     *
     * El status code es un integer de 3 dígitos resultado del intento del servidor
     * para comprender y satisfacer la solicitud.
     *
     * @return int código de estado.
     */
    public function getStatusCode();

    /**
     * Devuelve una instancia con el status code especificado y, opcionalmente, la frase de motivo.
     *
     * Si no se especifica ninguna frase de motivo, las implementaciones PUEDEN elegir el valor
     * predeterminado en la RFC 7231 o la frase de motivo recomendada por IANA para la respuesta de
     * status code.
     *
     * Este método DEBE implementarse de tal manera que retenga la
     * inmutabilidad del mensaje, y DEBE devolver una instancia que tenga el
     * estado actualizado y frase de motivo.
     *
     * @see http://tools.ietf.org/html/rfc7231#section-6
     * @see http://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml
     *
     * @param int $code El status code que se va a establecer, un integer de 3 dígitos.
     * @param string $reasonPhrase La frase de razón para usar con el status code proporcionado;
     *     si no se proporciona ninguno, las implementaciones PUEDEN utilizar los valores
     *     predeterminados como se sugiere en la especificación HTTP.
     *
     * @return static
     *
     * @throws \InvalidArgumentException Para argumentos de status code no válidos.
     */
    public function withStatus($code, $reasonPhrase = '');

    /**
     * Obtiene la frase de motivo de respuesta asociada con el código de estado.
     *
     * Porque una frase de motivo no es un elemento obligatorio en una respuesta
     * status line, el valor de la frase de motivo PUEDE estar vacío.
     * La implementación PUEDE elejir devolver la frase de motivo recomendada en
     * la RFC 7231 (o aquellas enumeradas en el Registro de códigos de estado HTTP de IANA)
     * para la respuesta de status code.
     *
     * @see http://tools.ietf.org/html/rfc7231#section-6
     * @see http://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml
     *
     * @return string Frase de motivo; debe devolver una string vacía si no hay ninguna presente.
     */
    public function getReasonPhrase();
}
```

Psr\Http\Message\StreamInterface

```
<?php
namespace Psr\Http\Message;

/**
```

```
 * Describe un data stream (flujo de datos).
 *
 * Normalmente, una instancia envolverá un stream de PHP; esta interfaz proporciona
 * un contenedor de las operaciones más comunes, incluida la serialización de
 * todo el stream en una string.
 */
interface StreamInterface
{
    /**
     * Lee todos los datos del stream en una string, desde el principio hasta el final.
     *
     * Este método DEBE intentar buscar el principio del stream antes de
     * leer datos y leer el stream hasta llegar al final.
     *
     * Advertencia: Esto podría intentar cargar una gran cantidad de datos en la memoria.
     *
     * Este método NO DEBE generar una excepción conforme con las operaciones
     * de string casting de PHP
     *
     * @see http://php.net/manual/es/language.oop5.magic.php#object.tostring
     *
     * @return string
     */
    public function __toString();

    /**
     * Cierra el stream y los recursos subyacentes.
     *
     * @return void
     */
    public function close();

    /**
     * Separa los recursos subyacentes del stream.
     *
     * Una vez que se ha desconectado el stream, el stream se encuentra en un estado inutilizable.
     *
     * @return resource|null Stream PHP subyacente, si lo hubiera
     */
    public function detach();

    /**
     * Obtener el tamaño del stream si se conoce.
     *
     * @return int|null Devuelve el tamaño en bytes si se conoce, o nulo si se desconoce.
     */
    public function getSize();

    /**
     * Devuelve la posición actual del puntero de lectura/escritura del archivo
     *
     * @return int Posición del puntero del archivo.
     *
     * @throws \RuntimeException en caso de error.
     */
    public function tell();

    /**
     * Devuelve true si el stream está al final del stream.
     *
     * @return bool
     */
    public function eof();

    /**
     * Devuelve si se puede buscar o no en el stream.
     *
     * @return bool
     */
    public function isSeekable();

    /**
     * Buscar una posición en el stream.
     *
     * @see http://www.php.net/manual/es/function.fseek.php
     *
     * @param int $offset Stream offset (Desplazamiento en el stream)
     * @param int $whence Especifica cómo se calculará la posición del cursor
```

```
     *     basado en el desplazamiento de búsqueda.
     *     Los valores válidos son idénticos a los de PHP 'fseek()' $whence.
     *     SEEK_SET: Establecer posición igual al offset en bytes.
     *     SEEK_CUR: Establecer la posición en la ubicación actual más el offset.
     *     SEEK_END: establece la posición al final del stream más el desplazamiento.
     *
     * @throws \RuntimeException en caso de error.
     */
    public function seek($offset, $whence = SEEK_SET);

    /**
     * Buscar al comienzo del stream.
     *
     * Si en el stream no se puede buscar, este método generará una excepción;
     * de lo contrario, realizará una búsqueda seek(0).
     *
     * @see seek()
     * @see http://www.php.net/manual/es/function.fseek.php
     *
     * @throws \RuntimeException en caso de error.
     */
    public function rewind();

    /**
     * Devuelve si el stream se puede escribir o no.
     *
     * @return bool
     */
    public function isWritable();

    /**
     * Escribe datos en el stream.
     *
     * @param string $string La string que se va a escribir.
     *
     * @return int Devuelve el número de bytes escritos en el stream.
     *
     * @throws \RuntimeException on failure.
     */
    public function write($string);

    /**
     * Devuelve si el stream es legible o no.
     *
     * @return bool
     */
    public function isReadable();

    /**
     * Leer datos del stream.
     *
     * @param int $length Lee hasta $length bytes del objeto y los devuelve.
     *     Puede devolver menos bytes que $length si el stream retorna menos bytes.
     *
     * @return string Devuelve los datos leídos del stream o una string vacía
     *     si no hay bytes disponibles.
     *
     * @throws \RuntimeException si ocurre un error.
     */
    public function read($length);

    /**
     * Devuelve el contenido restante en una string
     *
     * @return string
     *
     * @throws \RuntimeException si no se puede leer.
     * @throws \RuntimeException si se produce un error durante la lectura.
     */
    public function getContents();

    /**
     * Obtiene metadatos del stream como un array asociativo o recupera una clave específica.
     *
     * Las claves devueltas son idénticas a las claves devueltas de PHP
     * en la función PHP stream_get_meta_data().
     *
     * @see http://php.net/manual/es/function.stream-get-meta-data.php
```

```
     *
     * @param string $key Metadatos específicos para recuperar.
     *
     * @return array|mixed|null Devuelve un array asociativo si no se proporciona una clave.
     *     Devuelve un valor de clave específico si se proporciona una clave y el
     *     valor es encontrado, o null si no se encuentra la clave.
     */
    public function getMetadata($key = null);
}
```

Psr\Http\Message\UriInterface

```
<?php
namespace Psr\Http\Message;

/**
 * Objeto de valor que representa un URI.
 *
 * Esta interfaz está diseñada para representar URI de acuerdo con la RFC 3986 y para
 * proporcionar métodos para las operaciones más comunes. Funcionalidad adicional para
 * el trabajo con URI se puede proporcionar al principio de la interfaz o externamente.
 * Su uso principal es para solicitudes HTTP, pero también puede usarse en otros
 * contextos.
 *
 * Las instancias de esta interfaz se consideran inmutables; todos los métodos que
 * puede cambiar de estado DEBE implementarse de manera que conserven el
 * estado de la instancia actual y devolver una instancia que contiene el
 * cambio de estado.
 *
 * Normalmente, el encabezado del host también estará presente en el mensaje de solicitud.
 * Para las solicitudes server-side (del lado del servidor), scheme (el esquema) normalmente
 * será detectable en los parámetros del servidor.
 *
 * @see http://tools.ietf.org/html/rfc3986 (la especificación URI)
 */
interface UriInterface
{
    /**
     * Recuperar el componente scheme de la URI.
     *
     * Si no hay ningún scheme, este método DEBE devolver una string vacía.
     *
     * El valor devuelto DEBE normalizarse a minúsculas, según RFC 3986
     * Sección 3.1.
     *
     * El carácter ":" final no es parte del scheme y NO DEBE añadirse.
     *
     * @see https://tools.ietf.org/html/rfc3986#section-3.1
     *
     * @return string El scheme del URI.
     */
    public function getScheme();

    /**
     * Recuperar el componente authority del URI.
     *
     * Si no hay información de authority, este método DEBE devolver un string vacío.
     *
     * La sintaxis de authority del URI es:
     *
     * <pre>
     * [user-info@]host[:port]
     * </pre>
     *
     * Si el componente port no está configurado o es el port estándar para el scheme
     * actual, NO DEBE incluirse.
     *
     * @see https://tools.ietf.org/html/rfc3986#section-3.2
     *
     * @return string La authority del URI, en formato "[user-info@]host[:port]".
     */
    public function getAuthority();
```

```
    /**
     * Recuperar el componente user information del URI.
     *
     * Si no hay user information, este método DEBE devolver una string vacía.
     *
     * Si un user está presente en el URI, este devolverá ese valor;
     * Además, si el password también está presente, se añadira al
     * valor de user, con dos puntos (":") separando los valores.
     *
     * El carácter "@" final no es parte del user information y NO DEBE añadirse.
     *
     * @return string La user information de URI, en formato "username[:password]".
     */
    public function getUserInfo();

    /**
     * Recuperar el componente host del URI.
     *
     * Si no hay host, este método DEBE devolver una string vacía.
     *
     * El valor devuelto DEBE normalizarse a minúsculas, según RFC 3986
     * Sección 3.2.2.
     *
     * @see http://tools.ietf.org/html/rfc3986#section-3.2.2
     *
     * @return string El host del URI.
     */
    public function getHost();

    /**
     * Recuperar el componente port del URI.
     *
     * Si hay un port y no es estándar para el scheme actual, este método
     * DEBE devolverlo como un integer. Si el port es el port estándar
     * utilizado con el scheme actual, este método DEBERÍA devolver un valor null.
     *
     * Si no hay ningún port y no hay ningún scheme, este método DEBE devolver
     * un valor null.
     *
     * Si no hay ningún port, pero hay un scheme, este método PUEDE devolver
     * el port estándar para ese scheme, pero DEBERÍA devolver null.
     *
     * @return null|int El port del URI.
     */
    public function getPort();

    /**
     * Recuperar el componente path del URI.
     *
     * El path puede estar vacío o absoluto (comenzando con una barra) o
     * sin raiz (sin comenzar con una barra). Las implementaciones DEBEN soportar
     * las tres sintaxis.
     *
     * Normalmente, el path vacío "" y el path absoluto "/" se consideran iguales a
     * lo definido en la RFC 7230 Sección 2.7.3. Pero este método NO DEBE automáticamente
     * hacer esta normalización automáticamente porque en contextos con un path base recortado,
     * p. ej. el front controller, esta diferencia se vuelve significativa. Es la tarea
     * del usuario manejar tanto "" como "/".
     *
     * El valor devuelto DEBE estar en percent-encoded, pero NO DEBE double-encode
     * cualquier carácter. Para determinar qué caracteres codificar, consulte la
     * RFC 3986, Secciones 2 y 3.3.
     *
     * Por ejemplo, si el valor debe incluir una barra inclinada ("/") que no pretende ser
     * un delimitador entre segmentos del path, ese valor DEBE pasarse en forma codificada
     * a la instancia (p. ej., "%2F").
     *
     * @see https://tools.ietf.org/html/rfc3986#section-2
     * @see https://tools.ietf.org/html/rfc3986#section-3.3
     *
     * @return string El path del URI.
     */
    public function getPath();

    /**
     * Recuperar la query string del URI.
     *
     * Si no hay una query string, este método DEBE devolver una string vacía.
```

```
     *
     * El carácter "?" no es parte de la consulta y NO DEBE añadirse.
     *
     * El valor devuelto DEBE estar en percent-encoded, pero NO DEBE double-encode
     * cualquier carácter. Para determinar qué caracteres codificar, consulte la
     * RFC 3986, Secciones 2 y 3.4.
     *
     * Como ejemplo, si un valor en un par clave/valor de la query string debe
     * incluir un ampersand ("&") que no pretende ser un delimitador entre valores,
     * ese valor DEBE pasarse a la instancia en forma codificada (p. ej., "%26").
     *
     * @see https://tools.ietf.org/html/rfc3986#section-2
     * @see https://tools.ietf.org/html/rfc3986#section-3.4
     *
     * @return string La query string del URI.
     */
    public function getQuery();

    /**
     * Recuperar el componente fragment de la URI.
     *
     * Si no hay ningún fragment presente, este método DEBE devolver una string vacía.
     *
     * El carácter "#" inicial no es parte del fragment y NO DEBE añadirse.
     *
     * El valor devuelto DEBE estar en percent-encoded, pero NO DEBE double-encode
     * cualquier carácter. Para determinar qué caracteres codificar, consulte la
     * RFC 3986, Secciones 2 y 3.5.
     *
     * @see https://tools.ietf.org/html/rfc3986#section-2
     * @see https://tools.ietf.org/html/rfc3986#section-3.5
     *
     * @return string El fragment del URI.
     */
    public function getFragment();

    /**
     * Devuelve una instancia con el scheme especificado.
     *
     * Este método DEBE conservar el estado de la instancia actual y devolver
     * una instancia que contiene el scheme especificado.
     *
     * Las implementaciones DEBEN admitir los scheme "http" y "https",
     * y PUEDE adaptarse a otros scheme si es necesario.
     *
     * Un scheme vacío equivale a eliminar el scheme.
     *
     * @param string $scheme El scheme a usar con la nueva instancia.
     *
     * @return static Una nueva instancia con el scheme especificado.
     *
     * @throws \InvalidArgumentException para scheme no válidos.
     * @throws \InvalidArgumentException para scheme no soportados.
     */
    public function withScheme($scheme);

    /**
     * Devuelve una instancia con la user information especificada.
     *
     * Este método DEBE conservar el estado de la instancia actual y devolver
     * una instancia que contiene la user information especificada.
     *
     * El password es opcional, pero la user information DEBE incluir el
     * user; una string vacía para user es equivalente a eliminar la user information.
     *
     * @param string $user El user name que se utilizará para authority.
     * @param null|string $password El password asociado con $user.
     *
     * @return static A new instance with the specified user information.

     * @return static Una nueva instancia con la user information especificada.
     */
    public function withUserInfo($user, $password = null);

    /**
     * Devuelve una instancia con el host especificado.
     *
     * Este método DEBE conservar el estado de la instancia actual y devolver
```

```
     * una instancia que contiene el host especificado.
     *
     * Un valor de host vacío equivale a eliminar el host.
     *
     * @param string $host El hostname que se usará con la nueva instancia.
     *
     * @return static Una nueva instancia con el host especificado.
     *
     * @throws \InvalidArgumentException para hostnames no válidos.
     */
    public function withHost($host);

    /**
     * Devuelve una instancia con el port especificado.
     *
     * Este método DEBE conservar el estado de la instancia actual y devolver
     * una instancia que contiene el port especificado.
     *
     * Las implementaciones DEBEN generar una excepción para los port fuera del
     * rangos de puertos establecidos para TCP y UDP.
     *
     * Un valor null proporcionado para el port equivale a eliminar el la información de port.
     *
     * @param null|int $port El port que se utilizará con la nueva instancia; un valor null
     *     elimina la información del port.
     *
     * @return static Una nueva instancia con el port especificado.
     *
     * @throws \InvalidArgumentException para port no válido.
     */
    public function withPort($port);

    /**
     * Devuelve una instancia con el path (camino) especificado.
     *
     * Este método DEBE conservar el estado de la instancia actual y devolver
     * una instancia que contiene el path especificado.
     *
     * El path puede ser vacío, absoluto (comenzando con una barra) o
     * sin raíz (sin comenzar con una barra). Las implementaciones DEBEN apoyar las
     * tres sintaxis.
     *
     * Si se pretende que un path HTTP sea relativo al host en lugar de relativo al path
     * entonces debe comenzar con una barra ("/"). Un path HTTP que no comienza con una barra
     * se supone que es relativo a algún path base conocido por la aplicación o el consumidor.
     *
     * Los usuarios pueden proporcionar caracteres de path codificados y decodificados.
     * Las implementaciones aseguran la codificación correcta como se describe en getPath().
     *
     * @param string $path El path a usar con la nueva instancia.
     *
     * @return static Una nueva instancia con el path especificado.
     *
     * @throws \InvalidArgumentException para path no válidos.
     */
    public function withPath($path);

    /**
     * Devuelve una instancia con la query string (cadena de consulta) especificada.
     *
     * Este método DEBE conservar el estado de la instancia actual y devolver
     * una instancia que contiene la query string especificada.
     *
     * Los usuarios pueden proporcionar caracteres de consulta codificados y decodificados.
     * Las implementaciones aseguran la codificación correcta como se describe en getQuery().
     *
     * Un valor de query string vacío equivale a eliminar la query string.
     *
     * @param string $query La query string que se utilizará con la nueva instancia.
     *
     * @return static Una nueva instancia con la query string especificada.
     *
     * @throws \InvalidArgumentException para query strings no válidas.
     */
    public function withQuery($query);

    /**
     * Devuelve una instancia con el fragment (fragmento) de URI especificado.
```

```
     *
     * Este método DEBE conservar el estado de la instancia actual y devolver
     * una instancia que contiene el fragment de URI especificado.
     *
     * Los usuarios pueden proporcionar el fragment con caracteres codificados y decodificados.
     * Las implementaciones aseguran la codificación correcta como se describe en getFragment().
     *
     * Un valor de fragment vacío equivale a eliminar el fragment.
     *
     * @param string $fragment El fragment que se usará con la nueva instancia.
     *
     * @return static Una nueva instancia con el fragment especificado.
     */
    public function withFragment($fragment);

    /**
     * Devuelve la string representation como una referencia de URI.
     *
     * Dependiendo de los componentes del URI presentes, el string resultante
     * es un URI completo o una referencia relativa de acuerdo con el RFC 3986,
     * Sección 4.1. El método concatena los distintos componentes de la URI,
     * utilizando los delimitadores adecuados:
     *
     * - Si hay scheme, DEBE tener el sufijo ":".
     * - Si authority está presente, DEBE tener el prefijo "//".
     * - El path se puede concatenar sin delimitadores. Pero hay dos
     *   casos en los que el path debe ajustarse para hacer válida la referencia URI
     *   ya que PHP no permite lanzar una excepción en __toString():
     *     - Si el path no tiene raíces y hay una authority, el path DEBE tener
     *       el prefijo "/".
     *     - Si el path comienza con más de un "/" y no hay authority presente,
     *       las barras iniciales DEBEN reducirse a una.
     * - Si hay una query, DEBE tener el prefijo "?".
     * - Si hay un fragment, DEBE tener el prefijo "#".
     *
     * @see http://tools.ietf.org/html/rfc3986#section-4.1
     *
     * @return string
     */
    public function __toString();
}
```

Psr\Http\Message\UploadedFileInterface

```
<?php
namespace Psr\Http\Message;

/**
 * Objeto de valor que representa un archivo cargado a través de una solicitud HTTP.
 *
 * Las instancias de esta interfaz se consideran inmutables; todos los métodos que
 * pueden cambiar de estado DEBE implementarse de manera que conserven el
 * estado de la instancia actual y devolver una instancia que contiene el
 * cambio de estado.
 */
interface UploadedFileInterface
{
    /**
     * Recupera un stream que represente el archivo cargado.
     *
     * Este método DEBE devolver una instancia de StreamInterface, que representa el
     * archivo cargado. El propósito de este método es permitir la funcionalidad nativa
     * de PHP para manipular la carga de archivos, como stream_copy_to_stream() (aunque
     * el resultado deberá estar decorado en un manejador de stream nativo de PHP
     * para trabajar con las funciones).
     *
     * Si el método moveTo() ha sido llamado previamente, este método DEBE emitir una excepción.
     *
     * @return StreamInterface Representación del stream del archivo cargado.
     *
     * @throws \RuntimeException en los casos en los que no hay ningún stream disponible.
     * @throws \RuntimeException en los casos en los que no se puede crear el stream.
     */
    public function getStream();
```

```
    /**
     * Mueva el archivo cargado a una nueva ubicación.
     *
     * Utilice este método como alternativa a move_uploaded_file(). Este método esta
     * garantizado para trabajar en entornos SAPI y no SAPI.
     * Las implementaciones deben determinar en qué entorno se encuentran y utilizar el
     * método apropiado (move_uploaded_file(), rename(), o operaciones con streams)
     * para realizar la operación.
     *
     * $targetPath puede ser un path absoluto o un path relativo. Si es un
     * path relativo, la resolución debe ser la misma que la utilizada por la función rename() de PHP.
     *
     * El archivo o stream original DEBE eliminarse al finalizar.
     *
     * Si este método se llama más de una vez, cualquier llamada posterior DEBE generar
     * una excepción.
     *
     * Cuando se usa en un entorno SAPI donde se completa $_FILES, al escribir
     * archivos a través de moveTo(), is_uploaded_file() y move_uploaded_file() DEBEN ser
     * utilizado para garantizar que los permisos y el estado de carga se verifiquen correctamente.
     *
     * Si desea mover a un stream, use getStream(), como operaciones de SAPI
     * no puede garantizar la escritura en el stream de destino.
     *
     * @see http://php.net/is_uploaded_file
     * @see http://php.net/move_uploaded_file
     *
     * @param string $targetPath Path a la que mover el archivo cargado.
     *
     * @throws \InvalidArgumentException si el $targetPath especificado no es válido.
     * @throws \RuntimeException en cualquier error durante la operación de movimiento.
     * @throws \RuntimeException en la segunda o subsiguiente llamada al método.
     */
    public function moveTo($targetPath);

    /**
     * Recuperar el tamaño del archivo.
     *
     * Las implementaciones DEBERÍAN devolver el valor almacenado en la clave "size" de
     * el archivo en la matriz $_FILES si está disponible, ya que PHP lo calcula basandose
     * en el tamaño real transmitido.
     *
     * @return int|null El tamaño del archivo en bytes o null si se desconoce.
     */
    public function getSize();

    /**
     * Recupera el error asociado con el archivo cargado.
     *
     * El valor de retorno DEBE ser una de las constantes de PHP UPLOAD_ERR_XXX.
     *
     * Si el archivo se cargó correctamente, este método DEBE devolver UPLOAD_ERR_OK.
     *
     * Las implementaciones DEBERÍAN devolver el valor almacenado en la clave "error" de
     * el array $_FILES.
     *
     * @see http://php.net/manual/es/features.file-upload.errors.php
     *
     * @return int Una de las constantes de PHP UPLOAD_ERR_XXX.
     */
    public function getError();

    /**
     * Recupera el filename (nombre de archivo) enviado por el cliente.
     *
     * No confíe en el valor devuelto por este método. Un cliente puede enviar
     * un nombre de archivo malicioso con la intención de corromper o piratear su
     * solicitud.
     *
     * Las implementaciones DEBERÍAN devolver el valor almacenado en la clave "name" de
     * el archivo en el array $_FILES.
     *
     * @return string|null El filename enviado por el cliente o null si no fue dado.
     */
    public function getClientFilename();

    /**
```

```
     * Recuperar el media type (tipo de medio) enviado por el cliente.
     *
     * No confíe en el valor devuelto por este método. Un cliente puede enviar
     * un tipo de medio malicioso con la intención de corromper o piratear su
     * solicitud.
     *
     * Las implementaciones DEBERÍAN devolver el valor almacenado en la clave "type" de
     * el archivo en el array $_FILES.
     *
     * @return string|null El media type enviado por el cliente o null si no fue dado.
     */
    public function getClientMediaType();
}
```

PSR 11 - Container Interface (Interfaz de contenedor)

Información general

Puede consultar el original en: https://www.php-fig.org/psr/psr-11

Este documento describe una interfaz común para contenedores de ***dependency injection*** (inyección de dependencia).

El objetivo establecido por `ContainerInterface` es estandarizar cómo los Framework y las librerías hacen uso de un contenedor para obtener objetos y parámetros (***called entries*** (entradas de llamada)en el resto de este documento).

La palabra ***implementor*** (implementador) en este documento debe interpretarse como alguien que implementa `ContainerInterface` en una librería o Framework relacionado con la inyección de dependencias.

Los usuarios de contenedores de ***dependency injection - DIC*** (inyección de dependencia) se denominan ***user***.

Conceptos básicos

Identificadores de entrada

Un ***entry identifier*** (identificador de entrada) es cualquier string legal en PHP de al menos un carácter que identifica de forma única un item dentro de un contenedor.

Un ***entry identifier*** es una string opaca, por lo que los ***callers*** NO DEBEN asumir que la estructura de la string tiene algún significado semántico.

Lectura de un contenedor

`Psr\Container\ContainerInterface` expone dos métodos: `get` y `has`.

`get` toma un parámetro obligatorio: un ***entry identifier***, que DEBE ser una string. `get` puede devolver cualquier cosa (un valor `mixed`) o lanzar una `NotFoundExceptionInterface` si el contenedor no conoce el identificador.

Dos ***call*** (llamadas) sucesivas con el mismo identificador DEBERÍAN devolver el mismo valor, sin embargo, según el diseño del implementador y/o la configuración del usuario, se pueden devolver diferentes valores, por lo que el usuario NO DEBE confiar en obtener el mismo valor en 2 ***call*** sucesivas.

`has` toma un parámetro único: un ***entry identifier***, que DEBE ser una string, DEBE devolver `true` si el contenedor conoce el ***entry identifier*** y `false` si no. Si `has($id)` devuelve `false`, `get($id)` DEBE lanzar una `NotFoundExceptionInterface`.

Excepciones

Las excepciones lanzadas directamente por el contenedor DEBERÍAN implementar `Psr\Container\ContainerExceptionInterface`.

Una ***call*** al método `get` con un id inexistente DEBE arrojar un `Psr\Container\NotFoundExceptionInterface`.

Uso recomendado

Los usuarios NO DEBEN pasar un contenedor a un objeto para que el objeto pueda recuperar sus propias dependencias.

Esto significa que el contenedor se utiliza como un Service Locator (localizador de servicios), que es un patrón que generalmente se desaconseja.

Consulte la sección 4 del META documento para obtener más detalles.

Paquete

Las interfaces y clases descritas, así como las excepciones relevantes, se proporcionan como parte del paquete psr/container.

Los paquetes que proporcionan una implementación de contenedor de PSR deben declarar que proporcionan `psr/container-implementation 1.0.0`.

Los proyectos que requieran una implementación deben requerir `psr/container-implementation 1.0.0`.

Interfaces

Psr\Container\ContainerInterface

```php
<?php
namespace Psr\Container;

/**
 * Describe la interfaz de un contenedor que expone métodos para read (leer) sus entradas.
 */
interface ContainerInterface
{
    /**
     * Busca una entrada del contenedor por su identificador y lo devuelve.
     *
     * @param string $id Identificador de la entrada a buscar.
     *
     * @throws NotFoundExceptionInterface  No se encontró ninguna entrada para **this** identificador.
     * @throws ContainerExceptionInterface Error al recuperar la entrada.
     *
     * @return mixed Entry.
     */
    public function get($id);

    /**
     * Devuelve true si el contenedor puede devolver una entrada para el identificador dado.
     * Devuelve false en caso contrario.
     *
     * «has($id)» devolviendo true no significa que «get($id)» no lanzará una excepción.
     * Sin embargo, significa que «get($id)» no arrojará una «NotFoundExceptionInterface».
     *
     * @param string $id Identificador de la entrada a buscar.
     *
     * @return bool
     */
    public function has($id);
}
```

Desde psr/container versión 1.1, la interfaz anterior se ha actualizado para agregar sugerencias de *type* de argumento.

Desde psr/container versión 2.0, la interfaz anterior se ha actualizado para agregar sugerencias de *type* de retorno (pero solo en método `has()`).

Psr\Container\ContainerExceptionInterface

```php
<?php
namespace Psr\Container;

/**
 * Interfaz base que representa una excepción genérica en un contenedor.
 */
interface ContainerExceptionInterface
{
}
```

Psr\Container\NotFoundExceptionInterface

```
<?php
namespace Psr\Container;

/**
 * No se encontró entrada en el contenedor.
 */
interface NotFoundExceptionInterface extends ContainerExceptionInterface
{
}
```

PSR 12 - Extended Coding Style (Estilo de codificación extendido)

Esta especificación amplía y reemplaza PSR-2, la guía de estilo de codificación y requiere el cumplimiento de PSR-1, el estándar de codificación básico.

Al igual que PSR-2, la intención de esta especificación es reducir la fricción cognitiva al escanear código de diferentes autores.

Lo hace enumerando un conjunto compartido de reglas y expectativas sobre cómo formatear el código PHP.

Este PSR busca proporcionar una forma establecida en que las herramientas de estilo de codificación puedan implementarse, los proyectos pueden declarar la adherencia y los desarrolladores pueden relacionarse fácilmente entre diferentes proyectos.

Cuando varios autores colaboran en varios proyectos, es útil tener un conjunto de pautas para usar entre todos esos proyectos, por lo tanto, el beneficio de esta guía no está en las reglas en sí, sino en compartir esas reglas.

PSR-2 fue aceptado en 2012 y desde entonces se han realizado varios cambios en PHP que tienen implicaciones para las pautas de estilo de codificación. Si bien PSR-2 es muy completo en cuanto a la funcionalidad PHP que existía en el momento de escribir este artículo, la nueva funcionalidad está muy abierta a la interpretación.

Este PSR, por lo tanto, busca aclarar el contenido del PSR-2 en un contexto más moderno con nueva funcionalidad disponible, y hacer que las erratas del PSR-2 sean vinculantes.

Versiones de lenguajes anteriores

A lo largo de este documento, se PUEDE ignorar cualquier instrucción si no existe en las versiones de PHP compatibles con su proyecto.

Ejemplo

Este ejemplo abarca algunas de las reglas siguientes como una descripción general rápida:

```
<?php

declare(strict_types=1);

namespace Vendor\Package;

use Vendor\Package\{ClassA as A, ClassB, ClassC as C};
use Vendor\Package\SomeNamespace\ClassD as D;

use function Vendor\Package\{functionA, functionB, functionC};

use const Vendor\Package\{ConstantA, ConstantB, ConstantC};

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

    final public static function bar()
    {
        // cuerpo del método
    }
}
```

General

Estándar de codificación básico

El código DEBE seguir todas las reglas descritas en PSR-1.

El término *StudlyCaps* en PSR-1 DEBE interpretarse como *PascalCase*, donde la primera letra de cada palabra está en mayúscula, incluida la primera letra.

Archivos

Todos los archivos PHP DEBEN usar el final de línea Unix `LF` (salto de línea) solamente.

Todos los archivos PHP DEBEN terminar con una línea que no esté en blanco, terminada con un solo `LF`.

La etiqueta de cierre `?>` DEBE omitirse de los archivos que solo contienen PHP.

Líneas

NO DEBE haber un límite estricto en la longitud de la línea.

El límite flexible de la longitud de la línea DEBE ser de 120 caracteres.

Las líneas NO DEBEN tener más de 80 caracteres; las líneas más largas que esa DEBERÍAN dividirse en varias líneas posteriores de no más de 80 caracteres cada una.

NO DEBE haber espacios en blanco al final de las líneas.

PUEDEN agregarse líneas en blanco para mejorar la legibilidad y para indicar bloques de código relacionados, excepto donde esté explícitamente prohibido.

NO DEBE haber más de una declaración por línea.

Sangría

El código DEBE usar una sangría de *4 espacios* para cada nivel de sangría, y NO DEBE usar tabulaciones para sangrar.

Keywords and Types (Palabras clave y tipos)

Todos los *type* y *reserved keywords* (palabras clave reservadas) de PHP [Lista de palabras reservadas] [Listado de otras palabras reservadas] DEBEN estar en minúsculas.

Cualquier nuevo *type* y *keyword* que se agregue a futuras versiones de PHP DEBE estar en minúsculas.

DEBE usarse una forma corta de *keyword* de *type*, es decir, `bool` en lugar de `boolean`, `int` en lugar de `integer`, etc.

Declarar Statements, Namespace, and Import Statements

El encabezado de un archivo PHP puede constar de varios bloques diferentes.

Si está presente, cada uno de los bloques a continuación DEBE estar separado por una sola línea en blanco y NO DEBE contener una línea en blanco.

Cada bloque DEBE estar en el orden que se indica a continuación, aunque los bloques que no son relevantes pueden omitirse.

- Abriendo la etiqueta `<?` Php.

- *Docblock* a nivel de archivo.
- Una o más declare statements.
- La declaración de namespace del archivo.
- Una o más declaraciones `use` de importación de clases.
- Una o más declaraciones `use` de importación de funciones.
- Una o más declaraciones `use` de importación de constantes.
- El resto del código en el archivo.

Cuando un archivo contiene una combinación de HTML y PHP, se puede seguir utilizando cualquiera de las secciones anteriores.

Si es así, DEBEN estar presentes en la parte superior del archivo, incluso si el resto del código consiste en una etiqueta PHP de cierre y luego una mezcla de HTML y PHP.

Cuando la etiqueta de apertura Php `<?` está en la primera línea del archivo, DEBE estar en su propia línea sin otras declaraciones a menos que sea un archivo que contenga marcas fuera de las etiquetas de apertura y cierre de PHP.

Las declaraciones *import* nunca DEBEN comenzar con una barra invertida inicial, ya que siempre deben estar completamente calificadas.

El siguiente ejemplo ilustra una lista completa de todos los bloques:

```
<?php

/**
 * Este archivo contiene un ejemplo de estilos de codificación.
 */

declare(strict_types=1);

namespace Vendor\Package;

use Vendor\Package\{ClassA as A, ClassB, ClassC as C};
use Vendor\Package\SomeNamespace\ClassD as D;
use Vendor\Package\AnotherNamespace\ClassE as E;

use function Vendor\Package\{functionA, functionB, functionC};
use function Another\Vendor\functionD;

use const Vendor\Package\{CONSTANT_A, CONSTANT_B, CONSTANT_C};
use const Another\Vendor\CONSTANT_D;

/**
 * FooBar es un ejemplo de class.
 */
class FooBar
{
    // ... código PHP adicional ...
}
```

NO SE DEBEN utilizar *namespaces* compuestos con una profundidad de más de dos. Por lo tanto, la siguiente es la profundidad máxima de composición permitida:

```
<?php

use Vendor\Package\SomeNamespace\{
    SubnamespaceOne\ClassA,
    SubnamespaceOne\ClassB,
    SubnamespaceTwo\ClassY,
    ClassZ,
};
```

Y no se permitiría lo siguiente:

```php
<?php

use Vendor\Package\SomeNamespace\{
    SubnamespaceOne\AnotherNamespace\ClassA,
    SubnamespaceOne\ClassB,
    ClassZ,
};
```

Cuando desee declarar un ***type*** estricto en archivos que contienen marcas fuera de las etiquetas de apertura y cierre de PHP, la declaración DEBE estar en la primera línea del archivo e incluir una etiqueta PHP de apertura, la declaración de un ***type*** estricto y la etiqueta de cierre.

Por ejemplo:

```php
<?php declare(strict_types=1) ?>
<html>
<body>
    <?php
        // ... código PHP adicional ...
    ?>
</body>
</html>
```

Los ***declare statements*** NO DEBEN contener espacios y DEBEN ser exactamente `declare(strict_types=1)` (con un terminador de punto y coma opcional).

Los ***declare statements*** en bloque están permitidas y DEBEN tener el formato siguiente. Tenga en cuenta la posición de las llaves (`{}`) y el espaciado:

```php
declare(ticks=1) {
    // algún código
}
```

Clases, propiedades y métodos

El término "***clase***" se refiere a todas las ***clases***, ***interfaces*** y ***traits***.

Cualquier llave de cierre NO DEBE ir seguida de ningún comentario o declaración en la misma línea.

Al crear una instancia de una nueva clase, los paréntesis DEBEN estar siempre presentes incluso cuando no se pasen argumentos al constructor.

```php
new Foo();
```

Extend e Implement

Las palabras clave `extends` e `implements` DEBEN declararse en la misma línea que el nombre de la clase.

La llave de apertura para la clase DEBE ir en su propia línea; la llave de cierre para la clase DEBE ir en la siguiente línea después del cuerpo.

Las llaves de apertura DEBEN estar en su propia línea y NO DEBEN ir precedidas o seguidas de una línea en blanco.

Las llaves de cierre DEBEN estar en su propia línea y NO DEBEN ir precedidas de una línea en blanco.

```php
<?php
```

```
namespace Vendor\Package;

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

class ClassName extends ParentClass implements \ArrayAccess, \Countable
{
    // constantes, propiedades, métodos
}
```

Las listas de implementos y, en el caso de interfaces y extensiones PUEDEN dividirse en varias líneas, donde cada línea subsiguiente se sangra una vez. Al hacerlo, el primer elemento de la lista DEBE estar en la siguiente línea y DEBE haber solo una interfaz por línea.

```
<?php

namespace Vendor\Package;

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

class ClassName extends ParentClass implements
    \ArrayAccess,
    \Countable,
    \Serializable
{
    // constantes, propiedades, métodos
}
```

Uso de traits

La palabra clave `use` usada dentro de las clases para implementar `trait` DEBE declararse en la siguiente línea después de la llave de apertura.

```
<?php

namespace Vendor\Package;

use Vendor\Package\PrimerTrait;

class ClassName
{
    use PrimerTrait;
}
```

Cada `trait` individual que se importa a una clase DEBE incluirse uno por línea y cada inclusión DEBE tener su propia declaración de importación de uso.

```
<?php

namespace Vendor\Package;

use Vendor\Package\PrimerTrait;
use Vendor\Package\SegundoTrait;
use Vendor\Package\TercerTrait;

class ClassName
{
    use PrimerTrait;
    use SegundoTrait;
    use TercerTrait;
}
```

Cuando la clase no tiene nada después de la declaración de importación `use`, la llave de cierre de la clase (`}`) DEBE estar en la siguiente línea después de la declaración `use`.

```php
<?php

namespace Vendor\Package;

use Vendor\Package\PrimerTrait;

class ClassName
{
    use PrimerTrait;
}
```

De lo contrario, DEBE tener una línea en blanco después de la declaración de importación `use`.

```php
<?php

namespace Vendor\Package;

use Vendor\Package\PrimerTrait;

class ClassName
{
    use PrimerTrait;

    private $property;
}
```

Al usar los operadores `insteadof` y `as`, deben usarse de la siguiente manera, tomando nota de la sangría, el espaciado y las nuevas líneas, mas información en php.net Rasgos (Traits).

```php
<?php

class Talker
{
    use A;
    use B {
        A::smallTalk insteadof B;
    }
    use C {
        B::bigTalk insteadof C;
        C::mediumTalk as FooBar;
    }
}
```

Propiedades y constantes

La visibilidad DEBE declararse en todas las propiedades.

La visibilidad DEBE declararse en todas las constantes si la versión mínima de PHP de su proyecto es PHP 7.1 o posterior.

La *keyword* (palabra clave) `var` NO DEBE usarse para declarar una propiedad.

NO DEBE haber más de una propiedad declarada por declaración.

Los nombres de las propiedades NO DEBEN tener un prefijo con un solo subrayado para indicar visibilidad protegida o privada. Es decir, un prefijo de subrayado no tiene ningún significado explícitamente.

DEBE haber un espacio entre la declaración de tipo y el nombre de la propiedad.

Una declaración de propiedad se parece a lo siguiente:

```php
<?php

namespace Vendor\Package;

class ClassName
{
```

```
    public $foo = null;
    public static int $bar = 0;
}
```

Métodos y funciones

La visibilidad DEBE declararse en todos los métodos.

Los nombres de los métodos NO DEBEN tener un prefijo con un solo subrayado para indicar visibilidad protegida o privada. Es decir, un prefijo de subrayado no tiene ningún significado explícitamente.

Los nombres de métodos y funciones NO DEBEN declararse con un espacio después del nombre del método.

La llave de apertura ({)DEBE ir en su propia línea, y la llave de cierre (}) DEBE ir en la siguiente línea que sigue al cuerpo.

NO DEBE haber un espacio después del paréntesis de apertura ((), y NO DEBE haber un espacio antes del paréntesis de cierre ()).

Una declaración de método tiene el siguiente aspecto, tenga en cuenta la ubicación de los paréntesis, las comas, los espacios y las llaves:

```
<?php

namespace Vendor\Package;

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

Una declaración de función se parece a la siguiente, tenga en cuenta la ubicación de los paréntesis, las comas, los espacios y las llaves:

```
<?php

function fooBarBaz($arg1, &$arg2, $arg3 = [])
{
    // cuerpo de la función
}
```

Argumentos de método y función

En la lista de argumentos, NO DEBE haber un espacio antes de cada coma, y DEBE haber un espacio después de cada coma.

Los argumentos de método y función con valores predeterminados DEBEN ir al final de la lista de argumentos.

```
<?php

namespace Vendor\Package;

class ClassName
{
    public function foo(int $arg1, &$arg2, $arg3 = [])
    {
        // cuerpo del método
    }
}
```

Las listas de argumentos PUEDEN dividirse en varias líneas, donde cada línea subsiguiente se sangra una vez. Al hacerlo, el primer elemento de la lista DEBE estar en la siguiente línea y DEBE haber solo un argumento por línea.

Cuando la lista de argumentos se divide en varias líneas, el paréntesis de cierre y la llave de apertura DEBEN colocarse juntos en su propia línea con un espacio entre ellos.

```
<?php

namespace Vendor\Package;

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

Cuando tenga una declaración de *type* de retorno presente, DEBE haber un espacio después de los dos puntos seguido de la declaración de tipo.

Los dos puntos y la declaración DEBEN estar en la misma línea que el paréntesis de cierre de la lista de argumentos sin espacios entre los dos caracteres.

```
<?php

declare(strict_types=1);

namespace Vendor\Package;

class ReturnTypeVariations
{
    public function functionName(int $arg1, $arg2): string
    {
        return 'foo';
    }

    public function anotherFunction(
        string $foo,
        string $bar,
        int $baz
    ): string {
        return 'foo';
    }
}
```

En las declaraciones de *type* que aceptan valores `NULL`, NO DEBE haber un espacio entre el signo de interrogación y el tipo.

```
<?php

declare(strict_types=1);

namespace Vendor\Package;

class ReturnTypeVariations
{
    public function functionName(?string $arg1, ?int &$arg2): ?string
    {
        return 'foo';
    }
}
```

Cuando se usa el operador de referencia `&` antes de un argumento, NO DEBE haber un espacio después, como en el ejemplo anterior.

NO DEBE haber un espacio entre el operador variable de tres puntos y el nombre del argumento:

```
public function process(string $algorithm, ...$parts)
{
```

```
    // procesando
}
```

Al combinar el operador de referencia y el operador de tres puntos `variadic`, NO DEBE haber ningún espacio entre los dos:

```
public function process(string $algorithm, &...$parts)
{
    // procesando
}
```

abstract, final, y static

Cuando están presentes, las declaraciones `abstract` y `final` DEBEN preceder a la declaración de visibilidad.

Cuando está presente, la declaración `static` DEBE ir después de la declaración de visibilidad.

```
<?php

namespace Vendor\Package;

abstract class ClassName
{
    protected static $foo;

    abstract protected function zim();

    final public static function bar()
    {
        // cuerpo del método
    }
}
```

Llamadas a métodos y funciones

Al realizar una llamada a un método o función, NO DEBE haber un espacio entre el método o el nombre de la función y el paréntesis de apertura, NO DEBE haber un espacio después del paréntesis de apertura y NO DEBE haber un espacio antes del paréntesis de cierre.

En la lista de argumentos, NO DEBE haber un espacio antes de cada coma, y DEBE haber un espacio después de cada coma.

```
<?php

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

Las listas de argumentos PUEDEN dividirse en varias líneas, donde cada línea subsiguiente se sangra una vez, al hacerlo, el primer elemento de la lista DEBE estar en la siguiente línea y DEBE haber solo un argumento por línea.

Un solo argumento dividido en varias líneas (como podría ser el caso de una función o array anónimo) no constituye una división de la lista de argumentos en sí.

```
<?php

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

```php
<?php

somefunction($foo, $bar, [
  // ...
], $baz);

$app->get('/hola/{nombre}', function ($nombre) use ($app) {
    return 'Hola ' . $app->escape($nombre);
});
```

Estructuras de control

Las reglas generales de estilo para las estructuras de control son las siguientes:

- DEBE haber un espacio después de la *keyword* (palabra clave) de estructura de control
- NO DEBE haber un espacio después del paréntesis de apertura
- NO DEBE haber un espacio antes del paréntesis de cierre
- DEBE haber un espacio entre el paréntesis de cierre y la llave de apertura
- El cuerpo de la estructura DEBE ser sangrado una vez
- El cuerpo DEBE estar en la siguiente línea después de la llave de apertura.
- La llave de cierre DEBE estar en la siguiente línea después del cuerpo.

El cuerpo de cada estructura DEBE estar encerrado por llaves. Esto estandariza el aspecto de las estructuras y reduce la probabilidad de introducir errores a medida que se agregan nuevas líneas al cuerpo.

if, elseif, else

Una estructura `if` se parece a la siguiente, tenga en cuenta la ubicación de paréntesis, espacios y llaves; y que `else` y `elseif` están en la misma línea que la llave de cierre del cuerpo anterior.

```php
<?php

if ($expr1) {
    // if body
} elseif ($expr2) {
    // cuerpo de elseif
} else {
    // cuerpo de else
}
```

La *keyword* (palabra clave) `elseif` DEBE usarse en lugar de `else if` para que todas las *keyword* de control se vean como palabras individuales.

Las expresiones entre paréntesis PUEDEN dividirse en varias líneas, donde cada línea subsiguiente se sangra al menos una vez, al hacerlo, la primera condición DEBE estar en la siguiente línea.

El paréntesis de cierre y la llave de apertura DEBEN colocarse juntos en su propia línea con un espacio entre ellos.

Los operadores booleanos entre condiciones DEBEN estar siempre al principio o al final de la línea, no una combinación de ambos.

```php
<?php

if (
    $expr1
    && $expr2
) {
    // cuerpo del if
```

```
} elseif (
    $expr3
    && $expr4
) {
    // cuerpo del elseif
}
```

switch, case

Una estructura `switch` se parece a la siguiente, tenga en cuenta la ubicación de los paréntesis, los espacios y las llaves.

La declaración `case` DEBE tener una sangría una vez desde el `switch` , y la *keyword* `break` (u otra *keywords* de finalización) DEBE tener una sangría al mismo nivel que el cuerpo del `case`.

DEBE haber un comentario como `// no break` cuando la interrupción es intencional en un cuerpo de `case` no vacío.

```
<?php

switch ($expr) {
    case 0:
        echo 'Primer case, con break';
        break;
    case 1:
        echo 'Segundo case, que falla';
        // no break
    case 2:
    case 3:
    case 4:
        echo 'Tercer case, volver en lugar del break';
        return;
    default:
        echo 'Case predeterminado';
        break;
}
```

Las expresiones entre paréntesis PUEDEN dividirse en varias líneas, donde cada línea subsiguiente se sangra al menos una vez.

Al hacerlo, la primera condición DEBE estar en la siguiente línea.

El paréntesis de cierre y la llave de apertura DEBEN colocarse juntos en su propia línea con un espacio entre ellos.

Los operadores booleanos entre condiciones DEBEN estar siempre al principio o al final de la línea, no una combinación de ambos.

```
<?php

switch (
    $expr1
    && $expr2
) {
    // estructura del cuerpo
}
```

while, do while

Una declaración `while` se parece a la siguiente, tenga en cuenta la ubicación de los paréntesis, los espacios y las llaves.

```
<?php

while ($expr) {
    // estructura del cuerpo
}
```

Las expresiones entre paréntesis PUEDEN dividirse en varias líneas, donde cada línea subsiguiente se sangra al menos una vez, al hacerlo, la primera condición DEBE estar en la siguiente línea.

El paréntesis de cierre y la llave de apertura DEBEN colocarse juntos en su propia línea con un espacio entre ellos.

Los operadores booleanos entre condiciones DEBEN estar siempre al principio o al final de la línea, no una combinación de ambos.

```
<?php

while (
    $expr1
    && $expr2
) {
    // estructura del cuerpo
}
```

De manera similar, una instrucción `do while` se parece a la siguiente, tenga en cuenta la ubicación de los paréntesis, los espacios y las llaves.

```
<?php

do {
    // estructura del cuerpo
} while ($expr);
```

Las expresiones entre paréntesis PUEDEN dividirse en varias líneas, donde cada línea subsiguiente se sangra al menos una vez, al hacerlo, la primera condición DEBE estar en la siguiente línea.

Los operadores booleanos entre condiciones DEBEN estar siempre al principio o al final de la línea, no una combinación de ambos.

```
<?php

do {
    // estructura del cuerpo
} while (
    $expr1
    && $expr2
);
```

for

Una declaración `for` se parece a la siguiente, tenga en cuenta la ubicación de los paréntesis, los espacios y las llaves.

```
<?php

for ($i = 0; $i < 10; $i++) {
    // cuerpo del for
}
```

Las expresiones entre paréntesis PUEDEN dividirse en varias líneas, donde cada línea subsiguiente se sangra al menos una vez, al hacerlo, la primera expresión DEBE estar en la siguiente línea.

El paréntesis de cierre y la llave de apertura DEBEN colocarse juntos en su propia línea con un espacio entre ellos.

```
<?php

for (
    $i = 0;
    $i < 10;
    $i++
) {
    // cuerpo del for
}
```

foreach

Una declaración `foreach` se parece a la siguiente, tenga en cuenta la ubicación de los paréntesis, los espacios y las llaves.

```
<?php

foreach ($iterable as $key => $value) {
    // cuerpo del foreach
}
```

try, catch, finally

Un bloque `try-catch-finally` se parece a lo siguiente, tenga en cuenta la ubicación de los paréntesis, los espacios y las llaves.

```
<?php

try {
    // cuerpo del try
} catch (FirstThrowableType $e) {
    // cuerpo del catch
} catch (OtherThrowableType | AnotherThrowableType $e) {
    // cuerpo del catch
} finally {
    // cuerpo de finally
}
```

Operadores

- Las reglas de estilo para los operadores se agrupan por aridad (el número de operandos que toman).
- Cuando se permite el espacio alrededor de un operador, se PUEDEN utilizar varios espacios para facilitar la lectura.
- Todos los operadores que no se describen aquí quedan sin definir.

Operadores unarios

Los operadores de incremento/decremento NO DEBEN tener ningún espacio entre el operador y el operando.

```
$i++;
++$j;
```

Los operadores de *Type casting* (conversión de tipos) NO DEBEN tener ningún espacio entre paréntesis:

```
$intValue = (int) $input;
```

Operadores binarios

Todos los operadores binarios de aritmética, comparación, asignación, bitwise, lógicos, string y de type DEBEN ir precedidos y seguidos de al menos un espacio:

```
if ($a === $b) {
```

```
    $foo = $bar ?? $a ?? $b;
} elseif ($a > $b) {
    $foo = $a + $b * $c;
}
```

Operadores ternarios

El operador condicional, también conocido simplemente como operador ternario, DEBE estar precedido y seguido por al menos un espacio alrededor de ambos `?` y `:` :

```
$variable = $foo ? 'foo' : 'bar';
```

Cuando se omite el operando del medio del operador condicional, el operador DEBE seguir las mismas reglas de estilo que otros operadores de comparación binaria:

```
$variable = $foo ?: 'bar';
```

Cierres

Los cierres DEBEN declararse con un espacio después de la palabra clave `function` y un espacio antes y después de la *keyword* `use`.

La llave de apertura DEBE ir en la misma línea, y la llave de cierre DEBE ir en la siguiente línea que sigue al cuerpo.

NO DEBE haber un espacio después del paréntesis de apertura de la lista de argumentos o de la lista de variables, y NO DEBE haber un espacio antes del paréntesis de cierre de la lista de argumentos o la lista de variables.

En la lista de argumentos y la lista de variables, NO DEBE haber un espacio antes de cada coma, y DEBE haber un espacio después de cada coma.

Los argumentos de cierre con valores predeterminados DEBEN ir al final de la lista de argumentos.

Si un tipo de retorno está presente, DEBE seguir las mismas reglas que con las funciones y métodos normales; si la palabra clave `use` está presente, los dos puntos DEBEN seguir a la lista `use` cerrando paréntesis sin espacios entre los dos caracteres.

Una declaración de cierre tiene el siguiente aspecto, tenga en cuenta la ubicación de los paréntesis, las comas, los espacios y las llaves:

```
<?php

$closureWithArgs = function ($arg1, $arg2) {
    // cuerpo
};

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

$closureWithArgsVarsAndReturn = function ($arg1, $arg2) use ($var1, $var2): bool {
    // cuerpo
};
```

Las listas de argumentos y las listas de variables PUEDEN dividirse en varias líneas, donde cada línea subsiguiente se sangra una vez, al hacerlo, el primer elemento de la lista DEBE estar en la siguiente línea y DEBE haber solo un argumento o variable por línea.

Cuando la lista final (ya sea de argumentos o variables) se divide en varias líneas, el paréntesis de cierre y la llave de apertura DEBEN colocarse juntos en su propia línea con un espacio entre ellos.

Los siguientes son ejemplos de cierres con y sin listas de argumentos y listas de variables divididas en varias líneas.

```php
<?php

$longArgs_noVars = function (
    $longArgument,
    $longerArgument,
    $muchLongerArgument
) {
   // cuerpo
};

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

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

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

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

Tenga en cuenta que las reglas de formato también se aplican cuando el cierre se usa directamente en una función o llamada a un método como argumento.

```php
<?php

$foo->bar(
    $arg1,
    function ($arg2) use ($var1) {
        // cuerpo
    },
    $arg3
);
```

Clases anónimas

Las clases anónimas DEBEN seguir las mismas pautas y principios que los cierres en la sección anterior.

```php
<?php

$instance = new class {};
```

La llave de apertura PUEDE estar en la misma línea que la *keyword* `class` siempre que la lista de `implements` interfaces no se ajuste. Si la lista de interfaces se ajusta, la llave DEBE colocarse en la línea que sigue inmediatamente a la última interfaz.

```
<?php

// en la misma línea
$instance = new class extends \Foo implements \HandleableInterface {
    // contenido de la clase
};

// en la siguiente línea
$instance = new class extends \Foo implements
    \ArrayAccess,
    \Countable,
    \Serializable
{
    // Contenido de la clase
};
```

PSR 13 - Hypermedia Links (Enlaces hipermedia)

Información general

Puede consultar el original en: https://www.php-fig.org/psr/psr-13

Los enlaces hipermedia (la suma de Hipertexto y Multimedia) se están convirtiendo en una parte cada vez más importante de la web, tanto en contextos HTML como en varios contextos de formato API. Sin embargo, no existe un formato hipermedia común único, ni existe una forma común de representar enlaces entre formatos.

Esta especificación tiene como objetivo proporcionar a los desarrolladores de PHP una forma simple y común de representar un enlace hipermedia independientemente del formato de serialización que se utilice. Eso, a su vez, permite que un sistema serialice una respuesta con enlaces hipermedia en uno o más formatos de cable independientemente del proceso de decidir cuáles deberían ser esos enlaces.

Referencias

- RFC 2119
- RFC 4287
- RFC 5988
- RFC 6570
- IANA Link Relations Registry
- Microformats Relations List

Especificación

Enlaces básicos

Un enlace hipermedia consta, como mínimo:

- Un URI que representa el recurso de destino al que se hace referencia.
- Una relación que define cómo se relaciona el recurso de destino con la fuente.
- Pueden existir varios otros atributos del enlace, dependiendo del formato utilizado. Como los atributos adicionales no están bien estandarizados o son universales, esta especificación no busca estandarizarlos.

Para los propósitos de esta especificación, se aplican las siguientes definiciones.

- *Implementing Object* - (Objeto de implementación) un objeto que implementa una de las interfaces definidas por esta especificación.
- *Serializer* - (Serializador) una librería u otro sistema que toma uno o más objetos *Link* y produce una representación serializada de ellos en algún formato definido.

Atributos

Todos los enlaces PUEDEN incluir cero o más atributos adicionales más allá del URI y la relación. No existe un registro formal de los valores que se permiten aquí, y la validez de los valores depende del contexto y, a menudo, de un formato de serialización particular.

Los valores comúnmente admitidos incluyen '`hreflang`', '`title`' y '`type`'.

Los *serializer* PUEDEN omitir atributos en un objeto *link* si así lo requiere el formato de serialización. Sin embargo, los *serializer*

DEBEN codificar todos los atributos posibles para permitir la extensión del usuario a menos que lo impida la definición de un formato de serialización.

Algunos atributos (comúnmente `hreflang`) pueden aparecer más de una vez en su contexto, por lo tanto, un valor de atributo PUEDE ser un array de valores en lugar de un valor simple.

Los ***serializer*** PUEDEN codificar ese array en cualquier formato que sea apropiado para el formato serializado (como una lista separada por espacios, una lista separada por comas, etc.).

Si un atributo dado no puede tener múltiples valores en un contexto particular, los ***serializer*** DEBEN usar el primer valor proporcionado e ignorar todos los valores subsiguientes.

Si un valor de atributo es booleano `true`, los ***serializer*** PUEDEN usar formas abreviadas si es apropiado y respaldado por un formato de serialización. Por ejemplo, HTML permite que los atributos no tengan valor cuando la presencia del atributo tiene un significado booleano. Esta regla se aplica si y solo si el atributo es booleano `true`, no para cualquier otro "valor verdadero" en PHP como el entero `1`.

Si un valor de atributo es booleano `false`, los ***serializer*** DEBERÍAN omitir el atributo por completo a menos que al hacerlo cambie el significado semántico del resultado. Esta regla se aplica si y solo si el atributo es booleano `false`, no para cualquier otro "***valor falso***" en PHP como el entero `0`.

Relaciones

Las relaciones de ***link*** se definen como strings y son una keyword (palabra clave) simple en el caso de una relación definida públicamente o un URI absoluto en el caso de relaciones privadas.

En caso de que se utilice una keyword simple, DEBERÍA coincidir con una del registro de IANA en:

http://www.iana.org/assignments/link-relations/link-relations.xhtml

Opcionalmente, se PUEDE usar el registro de microformats.org, pero esto puede no ser válido en todos los contextos:

http://microformats.org/wiki/existing-rel-values

Una relación que no está definida en uno de los registros anteriores o en un registro público similar se considera "privada", es decir, específica para una aplicación o caso de uso en particular. Tales relaciones DEBEN usar un URI absoluto.

Plantillas de enlaces

RFC 6570 define un formato para las plantillas de URI, es decir, un patrón para un URI que se espera que se complete con los valores proporcionados por una herramienta de cliente.

Algunos formatos de hipermedia admiten ***link*** con plantillas, mientras que otros no, y pueden tener una forma especial de indicar que un ***link*** es una plantilla.

Un serializador para un formato que no admite plantillas de URI DEBE ignorar cualquier ***link*** con plantilla que encuentre.

Proveedores evolucionables

En algunos casos, un ***link provider*** (proveedor de enlaces) puede necesitar la posibilidad de que se le agreguen ***link*** adicionales.

En otros, un ***link provider*** es necesariamente de solo lectura, con ***link*** derivados en tiempo de ejecución de alguna otra fuente de datos. Por esa razón, los proveedores modificables son una interfaz secundaria que se puede implementar opcionalmente.

Además, algunos objetos de ***link provider***, como los objetos de respuesta de PSR-7, son inmutables por diseño.

Eso significa que los métodos para agregarles ***link*** en el lugar serían incompatibles. Por lo tanto, el método único de `EvolvableLinkProviderInterface` requiere que se devuelva un nuevo objeto, idéntico al original pero con un objeto ***link*** adicional incluido.

Objetos link evolucionables

Los objetos *link* son, en la mayoría de los casos, objetos de valor. Como tal, permitirles evolucionar de la misma manera que los objetos de valor PSR-7 es una opción útil. Por esa razón, se incluye una `EvolvableLinkInterface` adicional que proporciona métodos para producir nuevas instancias de objetos con un solo cambio.

El mismo modelo es utilizado por PSR-7 y, gracias al comportamiento de copia en escritura de PHP, sigue siendo eficiente en CPU y memoria.

Sin embargo, no existe un método evolutivo para los valores con plantilla, ya que el valor con plantilla de un *link* se basa exclusivamente en el valor `href`. NO DEBE establecerse de forma independiente, sino que se deriva de si el valor `href` es o no una plantilla de enlace RFC 6570.

Paquete

Las interfaces y clases descritas se proporcionan como parte del paquete psr/link.

Interfaces

Psr\Link\LinkInterface

```php
<?php

namespace Psr\Link;

/**
 * Un objeto link legible.
 */
interface LinkInterface
{
    /**
     * Devuelve el destino del link.
     *
     * El link de destino debe ser uno de los siguientes:
     * - Un URI absoluto, como se define en RFC 5988.
     * - Un URI relativo, como se define en RFC 5988. La base del link relativo
     *   se supone que el cliente lo conoce en función del contexto.
     * - Una plantilla de URI definida por RFC 6570.
     *
     * Si se devuelve una plantilla de URI, isTemplated() DEBE devolver True.
     *
     * @return string
     */
    public function getHref();

    /**
     * Devuelve si se trata de un link de plantilla o no.
     *
     * @return bool
     *   True si este objeto de link tiene una plantilla, False en caso contrario.
     */
    public function isTemplated();

    /**
     * Devuelve el (los) type(s) de relación del link.
     *
     * Este método devuelve 0 o más type de relación para un link, expresado
     * como un array de strings.
     *
     * @return string[]
     */
    public function getRels();

    /**
```

```
     * Devuelve una lista de atributos que describen el URI de destino.
     *
     * @return array
     *  Una lista de atributos key-value (clave-valor), donde la key es una string y el value
     *  es una primitiva de PHP o un array de string de PHP. Si no hay valores
     *  encontrado un array vacío DEBE ser devuelto.
     */
    public function getAttributes();
}
```

Psr\Link\EvolvableLinkInterface

```
<?php

namespace Psr\Link;

/**
 * Un objeto de valor de link evolutivo.
 */
interface EvolvableLinkInterface extends LinkInterface
{
    /**
     * Devuelve una instancia con el href especificado.
     *
     * @param string $href
     *   El valor href a incluir. Debe ser uno de:
     *    - Un URI absoluto, como se define en RFC 5988.
     *    - Un URI relativo, como se define en RFC 5988. La base del link relativo
     *      se supone que el cliente lo conoce en función del contexto.
     *    - Una plantilla de URI definida por RFC 6570.
     *    - Un objeto que implementa __toString() que produce uno de los anteriores
     *      valores.
     *
     * Una implementing library DEBE evaluar un objeto pasado a una string
     * inmediatamente en lugar de esperar a que se devuelva más tarde.
     *
     * @return static
     */
    public function withHref($href);

    /**
     * Devuelve una instancia con la relación especificada incluida.
     *
     * Si el rel especificado ya está presente, este método DEBE retornar
     * normalmente sin errores, pero sin agregar el rel por segunda vez.
     *
     * @param string $rel
     *   El valor de la relación a agregar.
     *
     * @return static
     */
    public function withRel($rel);

    /**
     * Devuelve una instancia con la relación especificada excluida.
     *
     * Si el rel especificado ya no está presente, este método DEBE retornar
     * normalmente sin errores.
     *
     * @param string $rel
     *   El valor de la relación para excluir.
     *
     * @return static
     */
    public function withoutRel($rel);

    /**
     * Devuelve una instancia con el atributo especificado agregado.
     *
     * Si el atributo especificado ya está presente, se sobrescribirá
     * con el nuevo valor.
     *
     * @param string $attribute
```

```
     *   El atributo a incluir.
     * @param string $value
     *   El valor del atributo a establecer.
     *
     * @return static
     */
    public function withAttribute($attribute, $value);

    /**
     * Devuelve una instancia con el atributo especificado excluido.
     *
     * Si el atributo especificado no está presente, este método DEBE retornar
     * normalmente sin errores.
     *
     * @param string $attribute
     *   El atributo a eliminar.
     *
     * @return static
     */
    public function withoutAttribute($attribute);
}
```

Psr\Link\LinkProviderInterface

```
<?php

namespace Psr\Link;

/**
 * Un objeto de proveedor de link.
 */
interface LinkProviderInterface
{
    /**
     * Devuelve un iterable de objetos LinkInterface.
     *
     * El iterable puede ser un array o cualquier objeto PHP\Traversable. Si no hay link
     * disponibles, DEBE devolverse un array vacío o \Traversable.
     *
     * @return LinkInterface[]|\Traversable
     */
    public function getLinks();

    /**
     * Devuelve un iterable de objetos LinkInterface que tienen una relación específica.
     *
     * El iterable puede ser un array o cualquier objeto PHP\Traversable. Si no hay link
     * con esa relación disponible, DEBE devolverse un array vacío o \Traversable.
     *
     * @return LinkInterface[]|\Traversable
     */
    public function getLinksByRel($rel);
}
```

Psr\Link\EvolvableLinkProviderInterface

```
<?php

namespace Psr\Link;

/**
 * Un objeto de valor de proveedor de link evolutivo.
 */
interface EvolvableLinkProviderInterface extends LinkProviderInterface
{
    /**
     * Devuelve una instancia con el link especificado incluido.
     *
     * Si el link especificado ya está presente, este método DEBE retornar sin errores.
```

```
     * El link está presente si $link es === idéntico a un objeto link que ya está
     * en la colección.
     *
     * @param LinkInterface $link
     *   Un objeto link que debería incluirse en esta colección.
     *
     * @return static
     */
    public function withLink(LinkInterface $link);

    /**
     * Devuelve una instancia con el link especificado eliminado.
     *
     * Si el link especificado no está presente, este método DEBE retornar sin errores.
     * El link está presente si $link es === idéntico a un objeto link que ya está
     * en la colección.
     *
     * @param LinkInterface $link
     *   The link to remove.
     *   El link para eliminar.
     *
     * @return static
     */
    public function withoutLink(LinkInterface $link);
}
```

Desde `psr/link` versión 1.1, las interfaces anteriores se han actualizado para agregar sugerencias de *type* de argumento.
Desde `psr/link` versión 2.0, las interfaces anteriores se han actualizado para agregar sugerencias de *type* de retorno.
Las referencias a `array|\Traversable` se han reemplazado por `iterable`.

PSR 14 - Event Dispatcher (Despachador de eventos)

Información general

Puede consultar el original en: https://www.php-fig.org/psr/psr-14

El ***Event Dispatcher*** (Despachador de eventos) es un mecanismo común y bien probado que permite a los desarrolladores inyectar lógica en una aplicación de manera fácil y coherente.

El objetivo de este PSR es establecer un mecanismo común para las extensiones basadas en eventos y colaboración para que las librerías y los componentes se puedan reutilizar más libremente entre varias aplicaciones y Frameworks.

Objetivo

Tener interfaces comunes para enviar y manejar eventos permite a los desarrolladores crear librerías que pueden interactuar con muchos Frameworks y otras librerías de una manera común.

Algunos ejemplos:

- Un Framework de seguridad que evitará guardar/acceder a datos cuando un usuario no tiene permiso.

- Un sistema común de almacenamiento en caché de página completa.

- Librerías que amplían otras librerías, independientemente del Framework en el que estén integradas.

- Un paquete de registro para rastrear todas las acciones realizadas dentro de la aplicación.

Definiciones

Event

Un ***event*** (evento) es un mensaje producido por un ***emitter*** (emisor). Puede ser cualquier objeto PHP arbitrario.

Listener

Un ***listener*** (oyente) es cualquier PHP invocable que espera que se le pase un ***event***.

Se puede pasar el mismo ***event*** a cero o más listener.

Un ***listener*** PUEDE poner en cola algún otro comportamiento asincrónico si así lo desea.

Emitter

Un ***emitter*** (emisor) es cualquier código arbitrario que desee enviar un ***event***.

Esto también se conoce como el "***calling code***" (código de llamada).

No está representado por ninguna estructura de datos en particular, sino que se refiere al caso de uso.

Dispatcher

Un ***dispatcher*** es un objeto de servicio al que un ***emitter*** le da un objeto ***event***.

El ***dispatcher*** es responsable de asegurar que el ***event*** se transmita a todos los ***listener*** relevantes, pero DEBE diferir la determinación de los ***listener*** responsables a un ***listener provider***.

Listener Provider

Un *listener provider* (proveedor de oyentes) es responsable de determinar qué *listener* son relevantes para un *event* determinado, pero NO DEBE llamar a los *listener* en sí.

Un *listener provider* puede especificar cero o más *listener* relevantes.

Events (Eventos)

Los *event* son objetos que actúan como la unidad de comunicación entre un *emitter* y los *listener* apropiados.

Los objetos *event* PUEDEN ser mutables si el caso de uso llama a los *listener* que brindan información al *emitter*.

Sin embargo, si no se necesita dicha comunicación bidireccional, se RECOMIENDA que el *event* se defina como inmutable; es decir, definido de tal manera que carece de métodos mutantes.

Los implementadores DEBEN asumir que se pasará el mismo objeto a todos los *listener*.

Se RECOMIENDA, pero NO ES NECESARIO, que los objetos *event* admitan la serialización y deserialización sin pérdidas; `$event == unserialize (serialize ($event))` DEBERÍA mantenerse verdadero.

Los objetos PUEDEN aprovechar la interfaz `serializable` de PHP, los métodos mágicos `__sleep()` o `__wakeup()`, o una funcionalidad de lenguaje similar si corresponde.

Stoppable Events (Eventos que se pueden detener)

Un *stoppable event* (event que se puede detener) es un caso especial de *event* que contiene formas adicionales de evitar que se llamen a más *listener*. Se indica mediante la implementación de `StoppableEventInterface`.

Un *event* que implementa `StoppableEventInterface` DEBE devolver true desde `isPropagationStopped()` cuando se haya completado cualquier *event* que represente.

Depende del implementador de la clase determinar cuándo pasa eso.

Por ejemplo, un *event* que solicita que un objeto `RequestInterface` PSR-7 coincida con un objeto `ResponseInterface` correspondiente podría tener un método `setResponse(ResponseInterface $res)` para que lo llame un *listener*, lo que hace que `isPropagationStopped()` devuelva `true`.

Listeners (Oyentes)

Un *listener* (oyente) puede ser cualquier PHP invocable.

Un *listener* DEBE tener uno y solo un parámetro, que es el *event* al que responde.

Los *listener* DEBEN escribir un indicio de ese parámetro tan específicamente como sea relevante para su caso de uso; es decir, un *listener* PUEDE dar una pista contra una interfaz para indicar que es compatible con cualquier tipo de *event* que implemente esa interfaz, o con una implementación específica de esa interfaz.

Un *listener* DEBE devolver `void`, y DEBE escribir una sugerencia de tipo que se devuelva explícitamente. Un *dispatcher* DEBE ignorar los valores de retorno de los *listener*.

Un *listener* PUEDE delegar acciones a otro código. Eso incluye que un *listener* sea un envoltorio delgado alrededor de un objeto que ejecuta la lógica empresarial real.

Un *listener* PUEDE poner en cola información del *event* para su posterior procesamiento por un proceso secundario, usando cron, un servidor de cola o técnicas similares. PUEDE serializar el objeto *event* en sí mismo; sin embargo, se debe tener cuidado pues no todos los objetos *event* puedan serializarse de manera segura. Un proceso secundario DEBE asumir que cualquier cambio que realice en un objeto *event* NO se propagará a otros *listener*.

Dispatcher (Despachador)

Un ***dispatcher*** es un objeto de servicio que implementa `EventDispatcherInterface`.

Es responsable de recuperar ***listener*** de un ***listener provider*** para el ***event*** enviado e invocar a cada ***listener*** con ese ***event***.

Un ***dispatcher***:

- DEBE llamar a los ***listener*** sincrónicamente en el orden en que se devuelven desde un ***listener provider***.
- DEBE devolver el mismo objeto ***event*** que se le pasó después de que haya terminado de invocar a los ***listener***.
- NO DEBE regresar al ***emitter*** hasta que todos los ***listener*** hayan ejecutado.

Si se pasa un ***stoppable event***, al ***dispatcher***

- DEBE llamar a `isPropagationStopped()` en el ***event*** antes de que se haya llamado a cada ***listener***. Si ese método devuelve `true`, DEBE devolver el ***event*** al ***emitter*** inmediatamente y NO DEBE llamar a más ***listener***. Esto implica que si se pasa un ***event*** al ***dispatcher*** que siempre devuelve `true` desde `isPropagationStopped()`, se llamará a cero ***listener***.

Un ***dispatcher*** DEBE asumir que cualquier ***listener*** devuelto por un ***listener provider*** es type-safe . Es decir, el ***dispatcher*** DEBE asumir que llamar a `$listener($event)` no producirá un `TypeError`.

Manejo de errores

Una excepción o error ***thrown*** (arrojado) por un ***listener*** DEBE bloquear la ejecución de cualquier otro ***listener***.

Se DEBE permitir que una Excepción o Error arrojado por un ***listener*** se propague de regreso al ***emitter***.

Un ***dispatcher*** PUEDE ***catch*** (atrapar) un objeto lanzado para registrarlo, permitir que se tomen medidas adicionales, etc., pero luego DEBE ***rethrow*** (volver a lanzar) el objeto original.

Listener Provider (Proveedor de oyentes)

Un ***listener provider*** (proveedor de oyentes) es un objeto de servicio responsable de determinar para qué ***listener*** son relevantes y deben llamarse para un ***event*** determinado.

Puede determinar tanto qué ***listener*** son relevantes como el orden en el que devolverlos por cualquier medio que elija.

Eso PUEDE incluir:

- Permitir algún tipo de mecanismo de registro para que los implementadores puedan asignar un ***listener*** a un ***event*** en un orden fijo.
- Derivar una lista de ***listener*** aplicables a través de la reflexión basada en el type y las interfaces implementadas del ***event***.
- Generar una lista compilada de ***listener*** con anticipación que se pueden consultar en tiempo de ejecución.
- Implementar alguna forma de control de acceso para que ciertos ***listener*** solo sean llamados si el usuario actual tiene un permiso determinado.
- Extraer información de un objeto al que hace referencia el ***event***, como una ***entity***, y llamar a métodos de ciclo de vida predefinidos en ese objeto.
- Delegar su responsabilidad a uno o más ***listener provider*** utilizando alguna lógica arbitraria.

Cualquier combinación de los mecanismos anteriores u otros, PUEDE usarse según se desee.

Los ***listener provider*** DEBEN usar el nombre de clase de un ***event*** para diferenciar un ***event*** de otro.

También PUEDEN considerar cualquier otra información sobre el ***event***, según corresponda.

Los *listener provider* DEBEN tratar los types principales de manera idéntica al type del propio *event* al determinar la aplicabilidad del *listener*.

En el siguiente caso:

```
class A {}

class B extends A {}

$b = new B();

function listener(A $event): void {};
```

Un *listener provider* DEBE tratar a `listener()` como un *listener* aplicable para `$b`, ya que es compatible con el type, a menos que algún otro criterio le impida hacerlo.

Composición del objeto

Un *dispatcher* DEBE tener un *listener provider* para determinar los *listener* relevantes.

Se RECOMIENDA que se implemente un *listener provider* como un objeto distinto del *dispatcher*, pero eso NO ES NECESARIO.

Interfaces

```
namespace Psr\EventDispatcher;

/**
 * Define un dispatcher para event.
 */
interface EventDispatcherInterface
{
    /**
     * Proporciona a todos los listener relevantes un event para procesar.
     *
     * @param object $event
     *   El objeto a procesar.
     *
     * @return object
     *   El event que se pasó, ahora modificado por los listener.
     */
    public function dispatch(object $event);
}
```

```
namespace Psr\EventDispatcher;

/**
 * Mapea un event a los listener que son aplicables a ese event.
 */
interface ListenerProviderInterface
{
    /**
     * @param object $event
     *   Un event para el que devolver los listener.
     * @return iterable<callable>
     *   Un iterable (array, iterator o generador) invocable.
     *   Cada invocable DEBE ser type-compatible con el $event.
     */
    public function getListenersForEvent(object $event) : iterable;
}
```

```php
namespace Psr\EventDispatcher;

/**
 * Un event cuyo procesamiento puede ser interrumpido cuando el event ha sido manejado.
 *
 * Una implementación de dispatcher DEBE verificar para determinar si un event
 * se marca como detenido después de llamar a cada listener. Si es así, debería
 * retornar inmediatamente sin llamar a más listener.
 */
interface StoppableEventInterface
{
    /**
     * ¿Se detiene la propagación?
     *
     * Esto normalmente solo lo utilizará el dispatcher para determinar si el
     * listener anterior detuvo la propagación.
     *
     * @return bool
     *   True si el evento está completo y no se deben llamar más listener.
     *   False para seguir llamando a los listener.
     */
    public function isPropagationStopped() : bool;
}
```

PSR 15 - HTTP Handlers (Controladores HTTP)

Información general

Puede consultar el original en: https://www.php-fig.org/psr/psr-15

Este documento describe las interfaces comunes para los controladores de solicitudes del servidor HTTP ("***request handlers***") y los componentes de middleware del servidor HTTP ("***middleware***") que utilizan mensajes HTTP como se describe en PSR-7 o las PSR de reemplazo posteriores.

Los ***request handlers*** HTTP son una parte fundamental de cualquier aplicación web.

El código del lado del servidor recibe un mensaje de solicitud, lo procesa y produce un mensaje de respuesta.

El ***middleware*** HTTP es una forma de alejar el procesamiento común de solicitudes y respuestas de la capa de aplicación.

Las interfaces descritas en este documento son abstracciones para ***request handlers*** y ***middleware***.

Nota: Todas las referencias a "***request handlers***" y "***middleware***" son específicas del procesamiento de solicitudes del servidor.

Referencias

- PSR-7
- RFC 2119

Especificación

Request Handlers

Un ***request handler*** (controlador de solicitudes) es un componente individual que procesa una solicitud y produce una respuesta, según lo definido por PSR-7.

Un ***request handler*** PUEDE lanzar una excepción si las condiciones de la solicitud le impiden producir una respuesta.

El tipo de excepción no está definido.

Los ***request handler*** que utilizan este estándar DEBEN implementar la siguiente interfaz:

- `Psr\Http\Server\RequestHandlerInterface`

Middleware

Un componente de ***middleware*** es un componente individual que participa, a menudo junto con otros componentes de ***middleware***, en el procesamiento de una solicitud entrante y la creación de una respuesta resultante, según lo definido por PSR-7.

Un componente de ***middleware*** PUEDE crear y devolver una respuesta sin delegar en un ***request handler***, si se cumplen las condiciones suficientes.

El ***middleware*** que utiliza este estándar DEBE implementar la siguiente interfaz:

- `Psr\Http\Server\MiddlewareInterface`

Generando respuestas

Se RECOMIENDA que cualquier ***middleware*** o manejador de solicitudes que genere una respuesta compondrá un prototipo de una `ResponseInterface` PSR-7 o una factory capaz de generar una instancia de `ResponseInterface` para evitar la dependencia de una implementación de mensaje HTTP específica.

Manejo de excepciones

Se RECOMIENDA que cualquier aplicación que utilice *middleware* incluya un componente que capte las excepciones y las convierta en respuestas.

Este *middleware* DEBE ser el primer componente ejecutado y envolver todo el procesamiento posterior para garantizar que siempre se genere una respuesta.

Interfaces

Psr\Http\Server\RequestHandlerInterface

Los *request handlers* DEBEN implementar la siguiente interfaz:

```
namespace Psr\Http\Server;

use Psr\Http\Message\ResponseInterface;
use Psr\Http\Message\ServerRequestInterface;

/**
 * Maneja una solicitud de servidor y produce una respuesta.
 *
 * Un request handler HTTP procesa una solicitud HTTP para producir una respuesta HTTP.
 */
interface RequestHandlerInterface
{
    /**
     * Maneja una solicitud y produce una respuesta.
     *
     * Puede llamar a otro código colaborador para generar la respuesta.
     */
    public function handle(ServerRequestInterface $request): ResponseInterface;
}
```

Psr\Http\Server\MiddlewareInterface

La siguiente interfaz DEBE ser implementada por componentes de *middleware* compatibles:

```
namespace Psr\Http\Server;

use Psr\Http\Message\ResponseInterface;
use Psr\Http\Message\ServerRequestInterface;

/**
 * Participante en el procesamiento de una solicitud y respuesta del servidor.
 *
 * Un componente de middleware HTTP participa en el procesamiento de un mensaje HTTP:
 * actuando sobre la solicitud, generando la respuesta o reenviando la solicitud
 * a un middleware posterior y posiblemente actuando sobre su respuesta.
 */
interface MiddlewareInterface
{
    /**
     * Procesar una solicitud de servidor entrante.
     *
     * Procesa una solicitud de servidor entrante para producir una respuesta.
     * Si no puede producir la respuesta por sí mismo, puede delegar en el
     * manejador que lo haga.
     */
    public function process(ServerRequestInterface $request, RequestHandlerInterface $handler):
ResponseInterface;
}
```

PSR 16 - Simple Cache (Caché simple)

Información general

Puede consultar el original en: https://www.php-fig.org/psr/psr-16

Common Interface for Caching Libraries (Interfaz común para librerías de almacenamiento en caché)

Este documento describe una interfaz simple pero extensible para un *item* de caché y *cache driver* (controlador de caché).

Las implementaciones finales PUEDEN decorar los objetos con más funcionalidad que la propuesta pero DEBEN implementar primero las interfaces/funcionalidad indicadas.

El *Caching* (almacenamiento en caché) es una forma común de mejorar el rendimiento de cualquier proyecto, lo que hace que las librerías de almacenamiento en caché sean una de las características más comunes de muchos Frameworks y librerías.

La interoperabilidad en este nivel significa que las librerías pueden eliminar sus propias implementaciones de almacenamiento en caché y confiar fácilmente en la que les proporciona el Framework u otra librería de caché dedicada.

El PSR-6 ya resuelve este problema, pero de una manera bastante formal y detallada para lo que necesitan los casos de uso más simples.

Este enfoque más simple tiene como objetivo construir una interfaz simplificada estandarizada para casos comunes.

Es independiente de PSR-6, pero ha sido diseñado para que la compatibilidad con PSR-6 sea lo más sencilla posible.

Definiciones

Las definiciones de *Calling Library*, *Implementing Library*, *TTL*, *Expiration* y *Key* son las mismas que en el PSR-6, ya que se cumplen las mismas suposiciones.

Cache

Un objeto que implementa la interfaz `Psr\SimpleCache\CacheInterface`.

Cahe Miss (Error de caché)

Un *cache miss* (error de caché) devolverá un valor `null` y, por lo tanto, no es posible detectar si se almacena un `null`. Ésta es la principal desviación de los supuestos del PSR-6.

Caché

Las implementaciones PUEDEN proporcionar un mecanismo para que un usuario especifique un *TTL* predeterminado si no se especifica uno para un *item* de caché específico.

Si no se proporciona ningún valor por defecto especificado por el usuario, las implementaciones DEBEN tomar por defecto el valor legal máximo permitido por la implementación subyacente.

Si la implementación subyacente no admite *TTL*, el *TTL* especificado por el usuario DEBE ignorarse.

Datos

La *implementing library* DEBE admitir todos los tipos de datos PHP serializables, incluidos:

String - Cadenas de caracteres de tamaño arbitrario en cualquier codificación compatible con PHP.

Integer - Todos los enteros de cualquier tamaño admitidos por PHP, hasta 64 bits con signo.

Float - Todos los valores de coma flotante.

Boolean - True y False.

Null - El valor null (aunque no se distinguirá de un error de caché al volver a leerlo).

Array - Matrices indexadas, asociativas y multidimensionales de profundidad arbitraria.

Object - Cualquier objeto que admita serialización y deserialización sin pérdida de modo que `o == unserialize(serialize($o))`. Los objetos PUEDEN aprovechar la interfaz serializable de PHP, los métodos mágicos `__sleep ()` o `__wakeup ()`, o una funcionalidad del lenguaje similar si corresponde.

Todos los datos pasados a la *Implementing Library* DEBEN devolverse exactamente como se pasaron. Eso incluye el tipo de variable. Es decir, es un error devolver `(string) 5` si `(int) 5` fue el valor guardado.

La *implementing library* PUEDE usar las funciones `serialize()/unserialize()` de PHP internamente, pero no es necesario que lo haga, la compatibilidad con ellas se utiliza simplemente como base para valores de objeto aceptables.

Si no es posible, por cualquier motivo, devolver exactamente el valor guardado, la *implementing library* DEBE responder con un error de caché en lugar de los datos dañados.

CacheInterface

La *CacheInterface* define las operaciones más básicas en una colección de entradas de caché, lo que implica la lectura, escritura y eliminación básicas de items de caché individuales.

Además, tiene métodos para manejar múltiples conjuntos de entradas de caché, como escribir, leer o eliminar múltiples entradas de caché a la vez. Esto es útil cuando tiene que realizar muchas lecturas/escrituras de caché y le permite realizar sus operaciones en una sola llamada al servidor de caché, lo que reduce drásticamente los tiempos de latencia.

Una instancia de *CacheInterface* corresponde a una única colección de items con una `key` de namespace única y es equivalente a un "`Pool`" en PSR-6.

Diferentes instancias de *CacheInterface* PUEDEN estar respaldadas por el mismo almacén de datos, pero DEBEN ser lógicamente independientes.

```
<?php

namespace Psr\SimpleCache;

interface CacheInterface
{
    /**
     * Obtiene un valor de la caché.
     *
     * @param string $key     La key (clave única) de este item en la caché.
     * @param mixed  $default Valor predeterminado que se devolverá si la clave no existe.
     *
     * @return mixed El valor del item de la caché, o $default en caso de cache miss.
     *
     * @throws \Psr\SimpleCache\InvalidArgumentException
     *   DEBE lanzarse si la string $key no es un valor legal.
     */
    public function get($key, $default = null);

    /**
     * Persiste los datos en la caché, referenciados de forma única por una key con un tiempo TTL de
     * vencimiento opcional.
     *
     * @param string                $key   La clave del item a almacenar.
     * @param mixed                 $value El valor del item para almacenar. Debe ser serializable.
     * @param null|int|\DateInterval $ttl   Opcional. El valor TTL de este item. Si no se envía ningún
     *    valor y el controlador admite TTL, entonces la librería puede establecer un valor redeterminado
```

```
     *    para ello o dejar que el driver se encargue de eso.
     *
     * @return bool True en caso de éxito y false en caso de error.
     *
     * @throws \Psr\SimpleCache\InvalidArgumentException
     *   DEBE lanzarse si la string $key no es un valor legal.
     */
    public function set($key, $value, $ttl = null);

    /**
     * Eliminar un item de la caché por su key única.
     *
     * @param string $key  La clave única del item a eliminar.
     *
     * @return bool  True si el item se eliminó correctamente. False si hubo un error.
     *
     * @throws \Psr\SimpleCache\InvalidArgumentException
     *   DEBE lanzarse si la string $key no es un valor legal.
     */
    public function delete($key);

    /**
     * Limpia las key (claves) de toda la caché.
     *
     * @return bool  True en caso de éxito y false en caso de error.
     */
    public function clear();

    /**
     * Obtiene múltiples item de caché por sus key únicas.
     *
     * @param iterable $keys    Una lista de claves que se pueden obtener en una sola operación.
     * @param mixed    $default Valor predeterminado para devolver para claves que no existen.
     *
     * @return iterable Una lista de pares key => valor. Las claves de caché que no existen o están
     *    obsoletas tendrán $default como valor.
     *
     * @throws \Psr\SimpleCache\InvalidArgumentException
     *   DEBE lanzarse si $keys no es un array ni un Traversable,
     *   o si alguna de las $keys no tiene valor legal.
     */
    public function getMultiple($keys, $default = null);

    /**
     * Conserva un conjunto de pares clave => valor en la caché, con un TTL opcional.
     *
     * @param iterable $values Una lista de pares key => valor para una operación de conjuntos múltiples.
     * @param null|int|\DateInterval $ttl    Opcional. El valor TTL de este item. Si no se envía
     *    ningún valor y el controlador admite TTL, entonces la librería puede establecer un valor
     *    predeterminado para ello o dejar que el driver se encargue de eso.
     *
     * @return bool True en caso de éxito y false en caso de error.
     *
     * @throws \Psr\SimpleCache\InvalidArgumentException
     *    DEBE arrojarse si $values no es un array ni un Traversable,
     *    o si alguno de los $values no es un valor legal.
     */
    public function setMultiple($values, $ttl = null);

    /**
     * Elimina varios item de la caché en una sola operación.
     *
     * @param iterable $keys Una lista de claves en una string que se eliminarán.
     *
     * @return bool True si los item se eliminaron correctamente. False si hubo un error.
     *
     * @throws \Psr\SimpleCache\InvalidArgumentException
     *    DEBE lanzarse si $keys no es un array ni un Traversable,
     *    o si alguna de las $keys no tiene valor legal.
     */
    public function deleteMultiple($keys);

    /**
     * Determina si un item está presente en la caché.
     *
     * NOTA: Se recomienda que has() solo se utilice para fines de avisos de caché y no debe utilizarse
     * en las operaciones de sus aplicaciones en vivo para get/set, ya que este método está sujeto a una
     * condición en la que devolverá true e inmediatamente después, otra secuencia de comandos puede
```

```
     * eliminarlo, lo que hace que el estado de su aplicación quede desactualizado.
     *
     * @param string $key La clave del item de caché.
     *
     * @return bool
     *
     * @throws \Psr\SimpleCache\InvalidArgumentException
     *    DEBE lanzarse si la string $key no es un valor legal.
     */
    public function has($key);
}
```

CacheException

```
<?php

namespace Psr\SimpleCache;

/**
 * Interfaz utilizada para todo tipo de excepciones lanzadas por la Implementing Library.
 */
interface CacheException
{
}
```

InvalidArgumentException

```
<?php

namespace Psr\SimpleCache;

/**
 * Interfaz de excepción para argumentos de caché no válidos.
 *
 * Cuando se pasa un argumento no válido, debe lanzar una excepción que implemente
 * esta interfaz.
 */
interface InvalidArgumentException extends CacheException
{
}
```

PSR 17 - HTTP Factories (Fábricas HTTP)

Información general

Puede consultar el original en: https://www.php-fig.org/psr/psr-17

Este documento describe un estándar común para las ***factories*** que crean objetos HTTP compatibles con PSR-7.

El PSR-7 no incluyó una recomendación sobre cómo crear objetos HTTP, lo que genera dificultades cuando es necesario crear nuevos objetos HTTP dentro de componentes que no están vinculados a una implementación específica de PSR-7.

Las interfaces descritas en este documento describen métodos mediante los cuales se pueden crear instancias de objetos PSR-7.

Especificación

Una ***factory HTTP*** es un método mediante el cual se crea un nuevo objeto HTTP, según lo definido por PSR-7.

Las ***factories HTTP*** DEBEN implementar estas interfaces para cada tipo de objeto proporcionado por el paquete.

Interfaces

Las siguientes interfaces PUEDEN implementarse juntas dentro de una sola clase o en clases separadas.

RequestFactoryInterface

Tiene la capacidad de crear solicitudes de clientes.

```
namespace Psr\Http\Message;

use Psr\Http\Message\RequestInterface;
use Psr\Http\Message\UriInterface;

interface RequestFactoryInterface
{
    /**
     * Crear una nueva solicitud.
     *
     * @param string $method El método HTTP asociado con la solicitud.
     * @param UriInterface|string $uri El URI asociado con la solicitud.
     */
    public function createRequest(string $method, $uri): RequestInterface;
}
```

ResponseFactoryInterface

Tiene la capacidad de generar respuestas.

```
namespace Psr\Http\Message;

use Psr\Http\Message\ResponseInterface;

interface ResponseFactoryInterface
{
    /**
     * Crea una nueva respuesta.
     *
     * @param int $code El código de estado HTTP. El valor predeterminado es 200.
     * @param string $reasonPhrase La frase de razón para asociar con el código de estado
```

```
     *     en la respuesta generada. Si no se proporciona ninguno, las implementaciones PUEDEN usar
     *     los valores predeterminados sugeridos en la especificación HTTP.
     */
    public function createResponse(int $code = 200, string $reasonPhrase = ''): ResponseInterface;
}
```

ServerRequestFactoryInterface

Tiene la capacidad de crear solicitudes de servidor.

```
namespace Psr\Http\Message;

use Psr\Http\Message\ServerRequestInterface;
use Psr\Http\Message\UriInterface;

interface ServerRequestFactoryInterface
{
    /**
     * Crear una nueva solicitud de servidor.
     *
     * Tenga en cuenta que los parámetros del servidor se toman exactamente como se dan,
     * no se realiza un análisis/procesamiento de los valores dados. En particular, no se intenta
     * determinar el método HTTP o URI, que debe proporcionarse explícitamente.
     *
     * @param string $method El método HTTP asociado con la solicitud.
     * @param UriInterface|string $uri El URI asociado con la solicitud.
     * @param array $serverParams Un array de parámetros de API de servidor (SAPI) con
     *     que sembrar la instancia de solicitud generada.
     */
    public function createServerRequest(string $method, $uri, array $serverParams = []):
ServerRequestInterface;
}
```

StreamFactoryInterface

Tiene la capacidad de crear flujos de solicitudes y respuestas.

```
namespace Psr\Http\Message;

use Psr\Http\Message\StreamInterface;

interface StreamFactoryInterface
{
    /**
     * Crea un nuevo stream a partir de una string.
     *
     * El stream DEBE crearse con un recurso temporal.
     *
     * @param string $content Contenido de la string con el que completar el stream.
     */
    public function createStream(string $content = ''): StreamInterface;

    /**
     * Crea un stream a partir de un archivo existente.
     *
     * El archivo DEBE abrirse usando el modo dado, que puede ser cualquier modo
     * compatible con la función 'fopen()'.
     *
     * El '$filename' PUEDE ser cualquier string compatible con fopen().
     *
     * @param string $filename El nombre de archivo o stream URI que se utilizará como base del stream.
     * @param string $mode El modo con el que se abre el nombre de archivo/stream.
     *
     * @throws \RuntimeException Si el archivo no se puede abrir.
     * @throws \InvalidArgumentException Si el modo no es válido.
     */
    public function createStreamFromFile(string $filename, string $mode = 'r'): StreamInterface;

    /**
```

```
     * Crea un nuevo stream a partir de un recurso existente.
     *
     * El stream DEBE ser legible y se puede escribir.
     *
     * @param resource $resource El recurso PHP que se utilizará como base para el stream.
     */
    public function createStreamFromResource($resource): StreamInterface;
}
```

Las implementaciones de esta interfaz DEBERÍAN usar un stream temporal al crear recursos a partir de strings. El método RECOMENDADO para hacerlo es:

```
$resource = fopen('php://temp', 'r+');
```

UploadedFileFactoryInterface

Tiene la capacidad de crear streams para archivos cargados.

```
namespace Psr\Http\Message;

use Psr\Http\Message\StreamInterface;
use Psr\Http\Message\UploadedFileInterface;

interface UploadedFileFactoryInterface
{
    /**
     * Crea un nuevo archivo cargado.
     *
     * Si no se proporciona un tamaño, se determinará comprobando el tamaño del stream.
     *
     * @link http://php.net/manual/features.file-upload.post-method.php
     * @link http://php.net/manual/features.file-upload.errors.php
     * @link http://php.net/manual/features.file-upload.post-method.php
     * @link http://php.net/manual/features.file-upload.errors.php
     *
     * @param StreamInterface $stream El stream subyacente que representa el
     *        contenido de archivo cargado.
     * @param int $size El tamaño del archivo en bytes.
     * @param int $error El error de carga (upload error) del archivo PHP.
     * @param string $clientFilename El nombre de archivo proporcionado por el cliente, si lo hubiera.
     * @param string $clientMediaType El tipo de medio proporcionado por el cliente, si lo hubiera.
     *
     * @throws \InvalidArgumentException Si el recurso del archivo no es legible.
     */
    public function createUploadedFile(
        StreamInterface $stream,
        int $size = null,
        int $error = \UPLOAD_ERR_OK,
        string $clientFilename = null,
        string $clientMediaType = null
    ): UploadedFileInterface;
}
```

UriFactoryInterface

Tiene la capacidad de crear URI para solicitudes de clientes y servidores.

```
namespace Psr\Http\Message;

use Psr\Http\Message\UriInterface;

interface UriFactoryInterface
{
    /**
     * Crea un nuevo URI.
     *
```

```
     * @param string $uri El URI a analizar.
     *
     * @throws \InvalidArgumentException Si el URI dado no se puede analizar.
     */
    public function createUri(string $uri = '') : UriInterface;
}
```

PSR 18 - HTTP Client (Cliente HTTP)

Información general

Puede consultar el original en: https://www.php-fig.org/psr/psr-18

Este documento describe una interfaz común para enviar solicitudes HTTP y recibir respuestas HTTP.

Objetivo

El objetivo de este PSR es permitir a los desarrolladores crear librerías desacopladas de las implementaciones del cliente HTTP.

Esto hará que las librerías sean más reutilizables, ya que reduce el número de dependencias y disminuye la probabilidad de conflictos de versiones.

Un segundo objetivo es que los clientes HTTP se puedan reemplazar según el principio de sustitución de Liskov. Esto significa que todos los clientes DEBEN comportarse de la misma manera al enviar una solicitud.

Definiciones

Client

Un ***client*** (cliente) es una librería que implementa esta especificación con el propósito de enviar mensajes de solicitud HTTP compatibles con PSR-7 y devolver un mensaje de respuesta HTTP compatible con PSR-7 a una ***Calling Library***.

Calling Library

Una ***Calling Library*** es cualquier código que hace uso de un client. No implementa las interfaces de esta especificación pero consume un objeto que las implementa (un ***client***).

Client

Un ***client*** es un objeto que implementa `ClientInterface`.

Un ***client*** PUEDE:

- Alterar la solicitud HTTP recibida y enviarla. Por ejemplo, podría comprimir el cuerpo de un mensaje saliente.
- Alterar la respuesta HTTP recibida antes de devolverla a la ***Calling Library***. Por ejemplo, podría descomprimir el cuerpo de un mensaje entrante.

Si un ***client*** altera la solicitud HTTP o la respuesta HTTP, DEBE asegurarse de que el objeto siga siendo coherente internamente.

Por ejemplo, si un ***client*** elige descomprimir el cuerpo del mensaje, DEBE también eliminar el encabezado `Content-Encoding` y ajustar el encabezado `Content-Length`.

Tenga en cuenta que, como resultado, dado que los objetos ***PSR-7 son inmutables***, la ***Calling Library*** NO DEBE asumir que el objeto pasado a `ClientInterface::sendRequest()` será el mismo objeto PHP que realmente se envía.

Por ejemplo, el objeto `Request` que es devuelto por una excepción PUEDE ser un objeto diferente al pasado a `sendRequest()`, por lo que la comparación por referencia (`===`) no es posible.

Un ***client*** DEBE:

- Volver a ensamblar una respuesta HTTP 1xx de varios pasos para que lo que se devuelva a la *Calling Library* sea una respuesta HTTP válida con código de estado 200 o superior.

Manejo de errores

Un *client* NO DEBE tratar una solicitud HTTP bien formada o una respuesta HTTP como una condición de error.

Por ejemplo, los códigos de estado de respuesta en el rango 400 y 500 NO DEBEN causar una excepción y DEBEN ser devueltos a la *Calling Library* como normales.

Un *client* DEBE lanzar una instancia de `Psr\Http\Client\ClientExceptionInterface` si y solo si no puede enviar la solicitud HTTP en absoluto o si la respuesta HTTP no se pudo analizar en un objeto de respuesta PSR-7.

Si no se puede enviar una solicitud porque el mensaje de solicitud no es una solicitud HTTP bien formada o le falta información crítica (como un *host* o método), el *client* DEBE lanzar una instancia de `Psr\Http\Client\RequestExceptionInterface`.

Si la solicitud no se puede enviar debido a un fallo de red de cualquier tipo, incluido un tiempo de espera superado, el *client* DEBE lanzar una instancia de `Psr\Http\Client\NetworkExceptionInterface`.

Los *client* PUEDEN lanzar excepciones más específicas que las definidas aquí (por ejemplo una `TimeOutException` o `HostNotFoundException`), siempre que implementen la interfaz apropiada definida anteriormente.

Interfaces

ClientInterface

```
namespace Psr\Http\Client;

use Psr\Http\Message\RequestInterface;
use Psr\Http\Message\ResponseInterface;

interface ClientInterface
{
    /**
     * Envía una solicitud de PSR-7 y devuelve una respuesta de PSR-7.
     *
     * @param RequestInterface $request
     *
     * @return ResponseInterface
     *
     * @throws \Psr\Http\Client\ClientExceptionInterface Si ocurre un error al procesar la solicitud.
     */
    public function sendRequest(RequestInterface $request): ResponseInterface;
}
```

ClientExceptionInterface

```
namespace Psr\Http\Client;

/**
 * Cada excepción relacionada con el cliente HTTP DEBE implementar esta interfaz.
 */
interface ClientExceptionInterface extends \Throwable
{
}
```

RequestExceptionInterface

```
namespace Psr\Http\Client;

use Psr\Http\Message\RequestInterface;

/**
 * Excepción para cuando una solicitud falló.
 *
 * Ejemplos:
 *      - La solicitud no es válida (ej. falta el método)
 *      - Errores de solicitud en tiempo de ejecución (ej. no se puede buscar en el stream )
 */
interface RequestExceptionInterface extends ClientExceptionInterface
{
    /**
     * Returns the request.
     * Devuelve la solicitud.
     *
     * El objeto de solicitud PUEDE ser un objeto diferente del que se pasó a
     * ClientInterface::sendRequest()
     *
     * @return RequestInterface
     */
    public function getRequest(): RequestInterface;
}
```

NetworkExceptionInterface

```
namespace Psr\Http\Client;

use Psr\Http\Message\RequestInterface;

/**
 * Se lanza cuando la solicitud no se puede completar debido a problemas de red.
 *
 * No hay ningún objeto de respuesta ya que esta excepción se lanza cuando
 * no se ha recibido respuesta.
 *
 * Ejemplo: el nombre de host de destino no se puede resolver o la conexión falló.
 */
interface NetworkExceptionInterface extends ClientExceptionInterface
{
    /**
     * Devuelve la solicitud.
     *
     * El objeto de solicitud PUEDE ser un objeto diferente del que se pasó a
     * ClientInterface::sendRequest()
     *
     * @return RequestInterface
     */
    public function getRequest(): RequestInterface;
}
```

PSR 19 - Etiquetas PHPDoc (Borrador)

Información general

Puede consultar el original en: https://github.com/php-fig/fig-standards/blob/master/proposed/phpdoc-tags.md

El objetivo principal de este PSR es proporcionar un catálogo completo de las ***tag*** (etiquetas) en el estándar PHPDoc.

Este documento NO DEBE:

- Describir un catálogo de ***annotation***.
- Describir las mejores prácticas o recomendaciones para estándares de codificación sobre la aplicación del estándar PHPDoc. Este documento se limita a una especificación formal de sintaxis e intención.

Convenciones utilizadas en este documento

Las descritas en la RFC 2119

Al final del documento puede encontrar un extracto del RFC 2119 con las palabras mencionadas y su interpretación.

Definiciones

Consulte la sección Definiciones del PSR 5 - PHPDoc, ya que esas definiciones también se aplican aquí. (***PHPDoc***, ***DocBlock***, ***tag***, ***type***, ***summary***,***annotation*** ...etc.)

Sobre la herencia

Un ***PHPDoc*** que está asociado con un "***Elemento estructural***" que implementa, extiende o anula.

Un "***Elemento estructural***" tiene la capacidad de heredar partes de información del ***PHPDoc*** asociado con el "***Elemento estructural***" que implementa, extiende o reemplaza.

El ***PHPDoc*** para cada ***type*** de "***Elemento estructural***" DEBE heredar las siguientes partes si esa parte está ausente:

- *Summary*
- ***Description*** y un subconjunto específico de ***Tags***:
    - *@author*
    - *@copyright*
    - *@version*

El ***PHPDoc*** para cada ***type*** de "***Elemento estructural***" DEBE también heredar un subconjunto especializado de ***tag*** (etiquetas) dependiendo de qué "***Elemento estructural***" esté asociado.

Si un ***PHPDoc*** no incluye una parte, como ***summary*** o ***description***, que está presente en el ***PHPDoc*** de un superelemento, entonces esa parte siempre se hereda implícitamente.

La siguiente es una lista de todos los elementos cuyos ***DocBlocks*** pueden heredar información del ***DocBlock*** de un superelemento:

- ***DocBlock*** de una clase o interfaz puede heredar información de una clase o interfaz que extiende.

- *DocBlock* de una propiedad puede heredar información de una propiedad con el mismo nombre que se declara en una superclase.
- *DocBlock* de un método puede heredar información de un método con el mismo nombre que se declara en una superclase.
- *DocBlock* de un método puede heredar información de un método con el mismo nombre que se declara en una interfaz implementada en la clase actual o que se implementa en una superclase.

Por ejemplo:

Supongamos que tiene un método `\SubClass::myMethod()` y su clase `\SubClass` extiende la clase `\SuperClass`. Y en la clase `\SuperClass` hay un método con el mismo nombre (por ejemplo, `\SuperClass::myMethod`).

Si se aplica lo anterior, *DocBlock* de `\SubClass::myMethod()` heredará cualquiera de las partes mencionadas anteriormente del *PHPDoc* de `\SuperClass::myMethod`. Entonces, si *@version* no se redefinió, se asume que `\SubClass:: myMethod()` tendrá la misma *@version*.

La herencia tiene lugar desde la raíz de jerarquía de clases hasta sus hojas.

Esto significa que cualquier cosa heredada en la parte inferior del árbol DEBE '*bubble*' (burbujear) hacia arriba a menos que se anule.

Haciendo explícita la herencia usando la tag: @inheritDoc

Debido a que la herencia es implícita, puede suceder que no sea necesario incluir un *PHPDoc* con un "*Elemento estructural*".

Esto puede causar confusión, ya que es ambiguo si el *PHPDoc* se omitió a propósito o si el autor del código se había olvidado de agregar documentación.

Para resolver esta ambigüedad, se puede usr *@inheritDoc*r para indicar que este elemento heredará su información de un superelemento.

Ejemplo:

```
/**
 * Este es un summary.
 */
class SuperClass
{
}

/**
 * @inheritDoc
 */
class SubClass extends SuperClass
{
}
```

En el ejemplo anterior, el *summary* de la subclase se puede considerar igual al del elemento SuperClase, que es, por lo tanto, "Este es un *summary* ".

Usar la tag en una línea {@inheritDoc} para aumentar una description

A veces desea heredar la *description* de un superelemento y agregar su propio texto con él para proporcionar información específica a su "*Elemento estructural*". Esto DEBE hacerse usando la *tag* en una línea `{@inheritDoc}`.

La *tag* en una línea `{@inheritDoc}` indicará que en esa ubicación DEBE inyectarse o inferirse de la description del superelemento.

Ejemplo:

```
/**
 * Este es el summary de este elemento.
 *
 * {@inheritDoc}
 *
 * Además esta description contendrá más información que
 * proporcionará una información más detallada y específica para este
```

```
 * elemento.
 */
```

En el ejemplo anterior, se indica que la ***description*** de este PHPDoc es una combinación de la ***description*** del superelemento, indicado por la ***tag*** en una línea `{@inheritDoc}`, y el texto del cuerpo posterior.

Partes específicas heredadas del elemento

Clase o interfaz

Además de las ***description*** y ***tag*** heredadas como se definen en este capítulo, una clase o interfaz DEBE heredar la siguiente ***tag***:

@package

Función o método

Además de las ***description*** y ***tag*** heredadas como se definen en este capítulo, una clase o interfaz DEBE heredar las siguientes ***tag***:

@param

@return

@throws

Constante o propiedad

Además de las ***description*** y ***tag*** heredadas como se definen en este capítulo, una constante o propiedad en una clase DEBE heredar la siguiente ***tag***:

@var

Tags (Etiquetas)

A menos que se mencione específicamente en la description, cada ***tag*** PUEDE aparecer cero o más veces en cada "***DocBlock***".

@api

@api se usa para resaltar "***Elementos estructurales***" como parte de la API pública primaria de un paquete.

Sintaxis: ***@api***

Descripción: ***@api*** PUEDE aplicarse a los "***Elementos Estructurales***" públicos para resaltarlos en la documentación generada, señalando al consumidor los componentes principales de la API pública de una librería o framework.

Otros "***Elementos Estructurales***" con visibilidad pública PUEDEN figurar de manera menos prominente en la documentación generada.

Consulte también ***@internal***, que PUEDE usarse para ocultar los "***Elementos estructurales***" internos de la documentación generada.

Ejemplos:

```
class UserService
{
    /**
```

```
     * Este método es una API pública.
     *
     * @api
     */
    public function getUser()
    {
        <...>
    }

    /**
     * Este método es "alcance del paquete", no una API pública
     */
    public function callMefromAnotherClass()
    {
        <...>
    }
}
```

@author

@author se utiliza para documentar el autor de cualquier "*Elemento estructural*".

Syntaxis: *@author [name] [<email address>]*

Descripción: *@author* se puede utilizar para indicar quién ha creado un "Elemento estructural" o ha realizado modificaciones importantes.

Esta *tag* PUEDE contener también una dirección de correo electrónico. Si se proporciona una dirección de correo electrónico, DEBE seguir al nombre del autor y estar entre corchetes o corchetes angulares, y DEBE adherirse a la sintaxis definida en RFC 2822.

Ejemplo:

```
/**
 * @author Mi nombre
 * @autor Mi nombre <mi_nombre@ejemplo.com>
 */
```

@copyright

@copyright se utiliza para documentar la información de copyright de cualquier "*Elemento estructural*".

Syntaxis: *@copyright <description>*

Descripción: *@copyright* define quién tiene los derechos de autor sobre el "*Elemento estructural*". Los derechos de autor indicados con esta *tag* se aplican al "*Elemento estructural*" al que se aplica y a todos los elementos secundarios a menos que se indique lo contrario.

El formato de la *description* se rige por el estándar de codificación de cada proyecto individual.

Se RECOMIENDA mencionar el año o años que están cubiertos por este derecho de autor y la organización involucrada.

Ejemplo:

```
/**
 * @copyright 1997-2021 The PHP Group
 */
```

@deprecated

@deprecated se utiliza para indicar qué '*Elementos estructurales*' están en desuso y se eliminarán en una versión futura.

Syntaxis: *@deprecated [<"Semantic Version">] [<description>]*

Descripción: *@deprecated* declara que los '*Elementos estructurales*' asociados se eliminarán en una versión futura, ya que se han vuelto obsoletos o no se recomienda su uso, a partir de la "*Versión semántica*" si se proporciona.

Esta *tag* PUEDE proporcionar una *description* adicional que indique por qué el elemento asociado está obsoleto.

Si el elemento asociado es reemplazado por otro, se RECOMIENDA agregar una *tag @see* en el mismo '*PHPDoc*' apuntando al nuevo elemento.

Ejemplo:

```
/**
 * @deprecated
 *
 * @deprecated 1.0.0
 *
 * @deprecated Ya no se usa por código interno y no se recomienda.
 *
 * @deprecated 1.0.0 Ya no se usa por código interno y no se recomienda.
 */
```

@internal

@internal se utiliza para indicar que el "*Elemento estructural*" asociado es una estructura interna de esta aplicación o librería.

También se puede usar dentro de una *description* para insertar un fragmento de texto que solo es aplicable para los desarrolladores de este software.

Syntaxis: *@internal [description]*

o en una línea: *{@internal [description]}*

A diferencia de otras *tag* en una línea, la versión en una línea de esta *tag* también puede contener otras *tag* en una línea (consulte el segundo ejemplo a continuación).

Descripción: *@internal* indica que el "*Elemento estructural*" asociado está destinado únicamente para su uso dentro de la aplicación, librería o paquete al que pertenece.

Los autores PUEDEN usar esta *tag* para indicar que un elemento con visibilidad pública debe considerarse exento de la API, por ejemplo:

Los autores de librerías PUEDEN considerar que los cambios importantes en los elementos internos están exentos del control de versiones semántico.

Las herramientas de análisis estático PUEDEN indicar el uso de elementos internos de otra librería/paquete con una advertencia o aviso. Al generar documentación a partir de comentarios *PHPDoc*, se RECOMIENDA ocultar el elemento asociado a menos que el usuario haya indicado explícitamente que se deben incluir elementos internos.

Un uso adicional de *@internal* es agregar comentarios internos o texto descriptivo adicional en la línea *description*. Esto se puede hacer, por ejemplo, para retener cierta información crítica o confusa al generar documentación a partir del código fuente del software.

Ejemplos:

Marca la función `count()` como interna de este proyecto:

```
/**
 * @internal
 *
 * @return int Indica el número de elementos.
 */
function count()
{
```

```
    <...>
}
```

Incluye una nota en la *description* que solo se mostrarán en los documentos para desarrolladores.

```
/**
 * Cuenta el número de Foo.
 *
 * Este método obtiene un recuento de Foo.
 * {@internal Los desarrolladores deben tener en cuenta que silenciosamente
 *            agrega un Foo adicional  (see {@link http://example.com}).}
 *
 * @return int Indica el número de items.
 */
function count()
{
    <...>
}
```

@link

@link indica una relación personalizada entre el "*Elemento estructural*" asociado y un sitio web, que se identifica mediante un URI absoluto.

Syntaxis: *@link [URI] [description]*

o en una línea: *{@link [URI] [description]}*

Descripción: *@link* se puede utilizar para definir una relación, o enlace, entre el "*Elemento estructural*", o parte de la *description* cuando se utiliza en una línea, a un URI.

El URI DEBE estar completo y bien formado como se especifica en la RFC 2396.

@link PUEDE tener una *description* adjunta para indicar el tipo de relación definida por esta ocurrencia.

Ejemplos:

```
/**
 * @link http://ejemplo.com/mi/bar Documentación de Foo.
 *
 * @return int Indica el número de elementos.
 */
function count()
{
    <...>
}

/**
 * Este método cuenta las apariciones de Foo.
 *
 * Cuando no hay ningún Foo ({@link http://ejemplo.com/mi/bar})
 * la función agregará uno, ya que siempre debe haber un Foo.
 *
 * @return int Indica el número de elementos.
 */
function count()
{
    <...>
}
```

@method

@method permite que una clase sepa qué métodos mágicos son invocables.

Syntaxis: *@method [return type] [name]([type] [parameter], [...]) [description]*

Descripción: *@method* se usa en situaciones en las que una clase contiene el método mágico `__call()` y define algunos usos definidos.

Un ejemplo de esto es una clase hija cuyo padre tiene un `__call()` para tener getters o setters dinámicos para propiedades predefinidas. La hija sabe qué *getters* y *setters* deben estar presentes y confía en la clase padre para que le proporcione el método `__call()`. En esta situación, la clase hija tendría una *@method* para cada método mágico *setter* o *getter*.

@method permite al autor comunicar el *type* de argumentos y el valor de retorno al incluir esos *type* en la firma.

Cuando el método deseado no tiene un valor de retorno, entonces el *type* de retorno PUEDE omitirse; en cuyo caso está implícito el "`void`".
@method SOLO se pueden usar en un *PHPDoc* que esté asociado con una clase o interfaz.

Ejemplo:

```
class Parent
{
    public function __call()
    {
        <...>
    }
}

/**
 * @method setInteger(int $integer)
 * @method string getString()
 * @method void setString(int $integer)
 */
class Child extends Parent
{
    <...>
}
```

@package

@package se utiliza para categorizar "*Elementos estructurales*" en subdivisiones lógicas.

Syntaxis: *@package [level 1]\[level 2]\[etc.]*

Descripción: *@package* se puede utilizar como contraparte o complemento de los namespaces.

Los namespaces proporcionan una subdivisión funcional de "*Elementos estructurales*" donde *@package* puede proporcionar una subdivisión lógica en la que los elementos se pueden agrupar con una jerarquía diferente.

Si, en general, las subdivisiones lógicas y funcionales son iguales, NO SE RECOMIENDA utilizar la *@package*, para evitar gastos de mantenimiento.

Cada nivel de la jerarquía lógica DEBE estar separado con una barra invertida (\) para que los namespaces resulten familiares.

Una jerarquía PUEDE tener una profundidad infinita, pero se RECOMIENDA mantener la profundidad menor o igual a seis niveles.

@package se aplica a ese namespace, clase o interfaz y los elementos que contiene. Significa que una función que está contenida en un namespace con *@package* se asume en ese package.

Esta *tag* NO DEBE aparecer más de una vez en un "*DocBlock*".

Ejemplo:

```
/**
 * @package PSR\Documentation\API
 */
```

@param

@param se usa para documentar un solo parámetro de una función o método.

Syntaxis: *@param ["Type"] [name] [<description>]*

Descripción: Con *@param* es posible documentar el *type* y función de un solo parámetro de una función o método.

Cuando se proporcione, DEBE contener un *type* para indicar lo que se espera.

El *name* es obligatorio solo cuando se omiten algunas *@param* debido a que toda la información útil ya está visible en la propia firma del código.

La *description* es OPCIONAL pero RECOMENDADA, *@param* PUEDE tener una *description* de varias líneas y no necesita delimitación explícita.

Se RECOMIENDA el uso de esta *tag* con cada función y método.

Esta *tag* NO DEBE aparecer más de una vez por parámetro en un "*PHPDoc*" y está limitada a "*Elementos estructurales*" de tipo método o función.

Ejemplo:

```
/**
 * Cuenta el número de elementos del array proporcionado.
 *
 * @param mixed[] $items Estructura del array para contar los elementos.
 *
 * @return int Devuelve el número de elementos.
 */
function count(array $items)
{
    <...>
}
```

@property

@property se utiliza para declarar qué propiedades "*mágicas*" son compatibles.

Syntaxis: *@property[<-read|-write>] ["Type"] [name] [<description>]*

Descripción: *@property* se usa cuando una clase (o trait) implementa los métodos "*mágicos*" `__get()` y/o `__set()` para resolver propiedades no literales en tiempo de ejecución.

Las variantes *@property-read* y *@property-write* PUEDEN usarse para indicar propiedades "*mágicas*" que solo se pueden leer o escribir.

Las *@property* SÓLO se pueden usar en un *PHPDoc* que esté asociado con una clase o trait.

Ejemplo:

En el siguiente ejemplo, una clase Usuario implementa el método mágico `__get()` para implementar la propiedad "*mágica*" de solo lectura `$nombre_completo`:

```
/**
 * @property-read string $nombre_completo
 */
class Usuario
{
    /**
```

```
     * @var string
     */
    public $nombre;

    /**
     * @var string
     */
    public $apellido;

    public function __get($nombre)
    {
        if ($nombre === "nombre_completo") {
            return "{$this->nombre} {$this->apellido}";
        }
    }
}
```

@return

@return se utiliza para documentar el valor de retorno de funciones o métodos.

Syntaxis: *@return <"Type"> [description]*

Descripción: Con *@return* es posible documentar el tipo de retorno de una función o método.

Cuando se proporciona, DEBE contener un *type* para indicar lo que se devuelve; la *description*, por otro lado, es OPCIONAL pero RECOMENDADA en caso de estructuras de retorno complicadas, como arrays asociativos.

@return PUEDE tener una *description* de varias líneas y no necesita delimitación explícita.

Se RECOMIENDA utilizar esta *tag* con cada función y método.

Una excepción a esta recomendación, según la define el Estándar de codificación de cualquier proyecto individual, PUEDE ser en funciones y métodos sin un valor de retorno: *@return* PUEDE omitirse, en cuyo caso un intérprete DEBE interpretar esto como si se proporcionara *@return* `void`.

Esta *tag* NO DEBE aparecer más de una vez en un "*DocBlock*" y está limitada al "*DocBlock*" de un "*Elemento estructural*" de un método o función.

Ejemplos:

```
/**
 * @return int Indica el número de elementos.

 */
function count()
{
    <...>
}

/**
 * @return string|null El texto de la tag o null si no se proporciona ninguno.
 */
function getLabel()
{
    <...>
}
```

@see

@see indica una referencia de los "*Elementos estructurales*" asociados a un sitio web u otros "*Elementos estructurales*".

Sintaxis: *@see [URI | "FQSEN"] [<description>]*

Descripción: *@see* se puede utilizar para definir una referencia a otros "*Elementos estructurales*" o a un URI.

Al definir una referencia a otros "*Elementos Estructurales*", puede hacer referencia a un elemento específico agregando dos puntos y proporcionando el nombre de ese elemento (también llamado "*FQSEN*").

Un URI DEBE estar completo y bien formado como se especifica en RFC 2396.

@see DEBE tener una *description* para proporcionar información adicional sobre la relación entre el elemento y su objetivo. Además, *@see* PUEDE tener una especialización de *tag* para agregar más definición a esta relación.

Ejemplo:

```
/**
 * @see number_of() :alias:
 * @see MyClass::$items           Para la propiedad cuyos elementos se cuentan.
 * @see MyClass::setItems()      Para configurar los elementos de esta colección.
 * @see http://example.com/my/bar Documentación de Foo.
 *
 * @return int Indica el número de elementos.
 */
function count()
{
    <...>
}
```

@since

@since se usa para indicar cuándo se introdujo o modificó un elemento, usando alguna description de "*control de versiones*" para ese elemento.

Syntaxis: *@since [<"Semantic Version">] [<description>]*

Descripción: Documenta la "*versión*" de la introducción o modificación de cualquier elemento.

Se RECOMIENDA que la versión coincida con un número de versión semántica (x.x.x) y PUEDE tener una *description* para proporcionar información adicional.

Esta información se puede utilizar para generar un conjunto de documentación API donde se informa al consumidor qué versión de la aplicación es necesaria para un elemento específico.

@since NO DEBE usarse para mostrar la versión actual de un elemento, *@version* PUEDE usarse para ese propósito.

Ejemplo:

```
/**
 * Esto es Foo
 * @version 2.1.7 MiApp
 * @since 2.0.0 iniciada
 */
class Foo
{
    /**
     * Mover.
     *
     * @since 2.1.5 mover($arg1 = '', $arg2 = null)
     *       se incorpora el $arg2 opcional
     * @since 2.1.0 mover($arg1 = '')
     *        se incorpora el $arg1 opcional
     * @since 2.0.0 mover()
     *        se crea el nuevo método mover()
     */
    public function mover($arg1 = '', $arg2 = null)
    {
        <...>
    }
}
```

@throws

@throws se usa para indicar si los "*Elementos estructurales*" arrojan un *type* específico de *Throwable* (excepción o error).

Syntaxis: *@throws ["Type"] [<description>]*

Descripción: *@throws* PUEDE usarse para indicar que los "elementos estructurales" arrojan un *type* específico de error.

El *type* proporcionado con esta *tag* DEBE representar un objeto que es un subtipo *Throwable*.

Esta *tag* se utiliza para presentar en su documentación qué error PODRÍA ocurrir y bajo qué circunstancias. Se RECOMIENDA proporcionar una *description* que describa el motivo por el que se lanza la excepción.

También se RECOMIENDA que esta *tag* se produzca cada vez que ocurra una excepción, incluso si tiene el mismo *type*. Al documentar cada ocurrencia, se crea una vista detallada y el consumidor sabe qué errores verificar.

Ejemplo:

```
/**
 * Cuenta el número de elementos del array proporcionado.
 *
 * @param mixed[] $array  Estructura del array para contar los elementos.
 *
 * @throws InvalidArgumentException Si el argumento proporcionado no es del type 'array'.
 *
 * @return int Devuelve el número de elementos.
 */
function count($items)
{
    <...>
}
```

@todo

@todo se usa para indicar si alguna actividad de desarrollo aún debe ejecutarse en los "*Elementos Estructurales*" asociados.

Syntaxis: *@todo [description]*

Descripción: *@todo* se usa para indicar que aún debe ocurrir una actividad que rodea a los "*Elementos Estructurales*" asociados.

Cada *tag* DEBE ir acompañada de una *description* que comunique la intención del autor original; Sin embargo, esto podría ser tan breve como proporcionar un número.

Ejemplo:

```
/**
 * Cuenta el número de elementos del array proporcionado.
 *
 * @todo agrega un parámetro del array a contar
 *
 * @return int Devuelve el número de elementos.
 */
function count()
{
    <...>
}
```

@uses

@uses indica si el "*Elemento estructural*" actual consume el "*Elemento estructural*", o el archivo de proyecto, que se proporciona como destino.

Syntaxis: *@uses [file | "FQSEN"] [<description>]*

Descripción: *@uses* describe si alguna parte del "*Elemento estructural*" asociado usa, o consume, otro "*Elemento estructural*" o un archivo que se encuentra en el proyecto actual.

Al definir una referencia a otro "*Elemento estructural*", puede hacer referencia a un elemento específico agregando dos puntos dobles y proporcionando el nombre de ese elemento (también llamado "*FQSEN*").

Esta *tag* puede hacer referencia a los archivos incluidos en este proyecto. Esto se puede utilizar, por ejemplo, para indicar una relación entre un *controller* (controlador) y un *view* (archivo de plantilla).

Esta *tag* NO DEBE usarse para indicar relaciones con elementos fuera del sistema, por lo que las URL no se pueden usar. Para indicar relaciones con elementos externos, se puede usar *@see*.

Se RECOMIENDA que las aplicaciones que consumen esta *tag*, como los generadores, proporcionen una *@used-by* en el elemento de destino. Esto se puede utilizar para proporcionar una experiencia bidireccional y permitir el análisis estático.

Ejemplos:

```
/**
 * @uses \SimpleXMLElement::__construct()
 */
function initializeXml()
{
    <...>
}

/**
 * @uses MiView.php
 */
function executeMiView()
{
    <...>
}
```

@var

Puede utilizar la *@var* para documentar el *type* de los siguientes "*Elementos Estructurales*":

- Constantes, tanto de clase como de ámbito global
- Propiedades
- Variables, tanto de ámbito global como local

Syntaxis: *@var ["Type"] [element_name] [<description>]*

Descripción: *@var* define qué *type* de datos está representado por un valor de Constante, Propiedad o Variable.

Cada constante o definición de propiedad o variable donde el *type* es ambiguo o desconocido DEBE ir precedido de un *DocBlock* que contenga *@var* , cualquier otra variable PUEDE ir precedida de un *DocBlock* que contenga *@var*.

@var DEBE contener el nombre del elemento que documenta. Una excepción a esto es cuando las declaraciones de propiedad solo se refieren a una sola propiedad, en este caso, PUEDE omitirse el nombre de la propiedad.

element_name se utiliza cuando se utilizan sentencias compuestas para definir una serie de constantes o propiedades. Una declaración compuesta de este tipo solo puede tener un *DocBlock* mientras se representan varios elementos.

Ejemplos:

```
/** @var int $int Esto es un contador. * /
$int = 0;

// no debería haber ningún docblock aquí
$int++;
```

o:

```
class Foo
{
  /** @var string|null Debe contener una description * /
  protected $description = null;

  public function setDescription($description)
  {
    // no debería haber ningún docblock aquí
      $this->description = $description;
  }
}
```

Otro ejemplo es documentar la variable en un *foreach* explícitamente; muchos IDE usan esta información para ayudarlo con el autocompletado:

```
/** @var \Sqlite3 $sqlite */
foreach ($connections as $sqlite) {
    // no debería haber ningún docblock aquí
    $sqlite->open('/my/database/path');
    <...>
}
```

Incluso las declaraciones compuestas pueden documentarse:

```
class Foo
{
  protected
      /**
       * @var string Debe contener una description
       */
      $name,
      /**
       * @var string Debe contener una description
       */
      $description;
}
```

o constantes:

```
class Foo
{
  const
      /**
       * @var string Debe contener una description
       */
      MY_CONST1 = "1",
      /**
       * @var string Debe contener una description
       */
      MY_CONST2 = "2";
}
```

@version

@version se usa para denotar alguna *description* de "*control de versiones*" de un elemento.

Syntaxis: *@version ["Semantic Version"] [<description>]*

Descripción: *@version* documenta la versión actual de cualquier elemento.

Esta información se puede utilizar para generar un conjunto de documentación API donde se informa al consumidor sobre los elementos de una versión particular.

Se RECOMIENDA que el número de versión coincida con un número de versión semántica como se describe en el Versionado Semántico 2.0.0

Los vectores de versión de los sistemas de control de versiones también son compatibles, aunque DEBEN seguir la forma:

name-of-vcs: $vector$

PUEDE proporcionarse una ***description*** con el fin de comunicar cualquier información adicional específica de la versión.
@version NO PUEDE usarse para mostrar la última versión modificada o de introducción de un elemento, DEBE usarse *@since* para ese propósito.

Ejemplo:

```
/**
 * File for class Foo
 * @version 2.1.7 MyApp
 *           (esta string indica el número de versión general de la aplicación)
 * @version @package_version@
 *          (this PEAR replacement keyword expands upon package installation)
 * @version $Id$
 *          (this CVS keyword expands to show the CVS file revision number)
 */

/**
 * This is Foo
 */
class Foo
{
  <...>
}
```

PSR 20 - Clock (Reloj)

Información general

Puede consultar el original en: https://github.com/php-fig/fig-standards/blob/master/proposed/clock.md

Interfaz común para acceder al Clock (Reloj del sistema)

Este documento describe una interfaz simple para leer el reloj del sistema.

Las implementaciones finales PUEDEN incorporar más funcionales en los objetos que la propuesta, pero DEBEN implementar primero las interfaces/funcionalidades indicadas.

La creación de una forma estándar de acceder al reloj permitiría la interoperabilidad durante las pruebas, al comprobar los efectos secundarios basados en el tiempo.

Las formas comunes de obtener la hora actual incluyen llamar a `time()` o `new DateTimeImmutable('now')`. Sin embargo, esto hace que aislarse de la hora actual sea imposible en algunas situaciones.

Definiciones

Clock

El reloj que puede leer la hora y la fecha actuales.

Timestamp

Marca de tiempo, la hora actual como un número entero de segundos desde el 1 de enero de 1970 a las 00:00:00 UTC.

Uso

Hay algunos patrones de uso comunes, que se describen a continuación:

Obtener el *timestamp* actual

Esto debe hacerse usando el método `getTimestamp()` en el valor retornado en `\DateTimeImmutable`:

```
$timestamp = $clock->now()->getTimestamp();
```

Interfaces

ClockInterface (Interfaz de reloj)

ClockInterface define las operaciones básicas para leer la hora y fecha actual del reloj. DEBE devolver la hora como `DateTimeImmutable`.

```
<?php

namespace Psr\Clock;
```

```
interface ClockInterface
{
    /**
     * Devuelve la hora actual como un objeto DateTimeImmutable.
     */
    public function now(): \DateTimeImmutable;

}
```

RFC 2119 - (Extracto)

Extracto del documento: https://www.rfc-es.org/rfc/rfc2119-es.txt

El estándar usa varias palabras para indicar los requerimientos de la especificación.

Estas palabras a menudo están en mayúsculas.

Estas palabras deberían ser interpretadas de la forma siguiente:

- DEBE Esta palabra, o los términos "OBLIGATORIO" o "DEBERÁ", significa que la definición es un requerimiento insoslayable de la especificación.
- NO DEBE Esta frase, o la frase "NO DEBERÁ", significa que la definición es una prohibición insoslayable de la especificación.
- DEBERÍA Esta palabra, o el adjetivo "RECOMENDADO", significa que pueden existir razones válidas en determinadas circunstancias para ignorar un elemento determinado, pero se deben comprender y sopesar detenidamente todas las implicaciones antes de tomar una decisióndiferente.
- NO DEBERÍA Esta frase, o la frase "NO RECOMENDADO", significa que pueden existir razones válidas en determinadas circunstancias en las que el comportamiento en particular sea útil o incluso aconsejable, pero se deben comprender y sopesar detenidamente todas las implicaciones antes de tomar una decisión diferente.
- PUEDE Esta palabra, o el adjetivo "OPCIONAL", significa que un elemento es realmente opcional. Un proveedor puede elegir incluir el elemento porque un mercado en particular lo necesite o porque el proveedor sienta que realza el producto aunque otro proveedor pueda omitir el mismo elemento. Una implementación que no incluya una opción determinada DEBE estar preparada para interoperar con otra implementación que incluya la opción, aunque quizá con reducida funcionalidad. En el mismo orden de cosas, una implementación que incluya una opción en particular DEBE estar preparada para interoperar con otra implementación que no incluya la opción (excepto, por supuesto, para la característica que aporte la opción).