unidad didáctica 5

Herramientas Web

Índice

1. duración y criterios de evaluación

Duración estimada: 16 sesiones

Resultado de aprendizaje y criterios de evaluación:

4. Desarrolla aplicaciones Web embebidas en lenguajes de marcas analizando e incorporando funcionalidades según especificaciones.

g) *Se han utilizado herramientas y entornos para facilitar la programación, prueba y depuración del código.*

2. Composer

Herramienta por excelencia en PHP para la gestión de librerías y dependencias, de manera que instala y las actualiza asegurando que todo el equipo de desarrollo tiene el mismo entorno y versiones. Además, ofrece autoloading de nuestro código, de manera que no tengamos que hacerlo nosotros "a mano".

Está escrito en PHP, y podéis consultar toda su documentación en https://getcomposer.org/.

Utiliza Packagist como repositorio de librerías.

Funcionalmente, es similar a Maven (Java) / npm (JS).

2.1. instalación

Si estamos usando XAMPP, hemos de instalar Composer en el propio sistema operativo. Se recomienda seguir las instrucciones oficiales según el sistema operativo a emplear.

En cambio, si usamos Docker, necesitamos modificar la configuración de nuestro contenedor. En nuestro caso, hemos decidido modificar el archivo Dockerfile y añadir el siguiente comando:

```
COPY --from=composer:2.0 /usr/bin/composer /usr/local/bin/composer
```

Para facilitar el trabajo, hemos creado una plantilla ya preparada.

Es importante que dentro del contenedor comprobemos que tenemos la v2:

```
composer -V
```

2.2. primeros pasos

Cuando creemos un proyecto por primera vez, hemos de inicializar el repositorio. Para ello, ejecutaremos el comando `composer init` donde:

- Configuramos el nombre del paquete, descripción, autor (nombre), tipo de paquete (project), etc...

- Definimos las dependencias del proyecto (`require`) y las de desarrollo (`require-dev`) de manera interactiva.
    - En las de desarrollo se indica aquellas que no se instalarán en el entorno de producción, por ejemplo, las librerías de pruebas.

Tras su configuración, se creará automáticamente el archivo `composer.json` con los datos introducidos y descarga las librerías en la carpeta `vendor`. La instalación de las librerías siempre se realiza de manera local para cada proyecto.

```
{
    "name": "dwes/log",
    "description": "Pruebas con Monolog",
    "type": "project",
    "require": {
        "monolog/monolog": "^2.1"
    },
    "license": "MIT",
    "authors": [
        {
            "name": "Nombre Apellido", // ejemplo: Carmen Llopis
            "email": "inicialNombre.Apellido@edu.gva.es" // ejemplo:
c.llopis@edu.gva.es
        }
    ]
}
```

A la hora de indicar cada librería introduciremos:

- el nombre de la librería, compuesta tanto por el creador o "vendor", como por el nombre del proyecto. Ejemplos: `monolog/monolog` o `laravel/installer`.
- la versión de cada librería. Tenemos diversas opciones para indicarla:
    - Directamente: 1.4.2
    - Con comodines: 1.*
    - A partir de: >= 2.0.3
    - Sin rotura de cambios: ^1.3.2 // >=1.3.2 <2.0.0

2.3. actualizar librerías

Podemos definir las dependencias via el archivo composer.json o mediante comandos con el formato composer require vendor/package:version. Por ejemplo, si queremos añadir phpUnit como librería de desarrollo, haremos:

```
composer require phpunit/phpunit –dev
```

Tras añadir nuevas librerías, hemos de actualizar nuestro proyecto:

```
composer update
```

Si creamos el archivo `composer.json` nosotros directamente sin inicializar el repositorio, hemos de instalar las dependencias:

```
composer install
```

Al hacer este paso (tanto instalar como actualizar), como ya hemos comentado, se descargan las librerías en dentro de la carpeta `vendor`. Es muy importante añadir esta carpeta al archivo `.gitignore` para no subirlas a GitHub.

Además se crea el archivo `composer.lock`, que almacena la versión exacta que se ha instalado de cada librería (este archivo no se toca).

2.4. autoload.php

Composer crea de forma automática en `vendor/autoload.php` el código para incluir de forma automática todas las librerías que tengamos configuradas en `composer.json`.

Para utilizarlo, en la cabecera de nuestro archivos pondremos:

```
<?php
    require 'vendor/autoload.php';
```

En nuestro caso, de momento sólo lo podremos en los archivos donde probamos las clases

Si queremos que Composer también se encargue de cargar de forma automática nuestras clases de dominio, dentro del archivo `composer.json`, definiremos la propiedad `autoload`:

```
"autoload": {
"psr-4": {"Dwes\\": "app/Dwes"}
},
```

Posteriormente, hemos de volver a generar el autoload de Composer mediante la opción `dump-autoload` (o `du`):

```
composer dump-autoload
```

3. Monolog

Vamos a probar Composer añadiendo la librería de Monolog a nuestro proyecto. Se trata de un librería para la gestión de logs de nuestras aplicaciones, soportando diferentes niveles (info, warning, etc...), salidas (ficheros, sockets, BBDD, Web Services, email, etc) y formatos (texto plano, HTML, JSON, etc...).

Para ello, incluiremos la librería en nuestro proyecto con:

```
composer require monolog/monolog
```

Monolog 2 requiere al menos PHP 7.2, cumple con el estandar de logging PSR-3, y es la librería empleada por Laravel y Symfony para la gestión de logs.

cómo usar un log

- Seguir las acciones/movimientos de los usuarios
- Registrar las transacciones
- Rastrear los errores de usuario
- Fallos/avisos a nivel de sistema
- Interpretar y coleccionar datos para posterior investigación de patrones

3.1. niveles

A continuación mostramos los diferentes niveles de menos a más restrictivo:

- *debug -100* : Información detallada con propósitos de debug. No usar en entornos de producción.
- *info - 200* : Eventos interesantes como el inicio de sesión de usuarios.
- *notice - 250* : Eventos normales pero significativos.
- *warning - 300* : Ocurrencias excepcionales que no llegan a ser error.
- *error - 400* : Errores de ejecución que permiten continuar con la ejecución de la aplicación pero que deben ser monitorizados.
- *critical - 500* : Situaciones importantes donde se generan excepciones no esperadas o no hay disponible un componente.
- *alert - 550* : Se deben tomar medidas inmediatamente. Caída completa de la web, base de datos no disponible, etc... Además, se suelen enviar mensajes por email.
- *emergency - 600* : Es el error más grave e indica que todo el sistema está inutilizable.

3.2. Hola Monolog

Por ejemplo, en el archivo `pruebaLog.php` que colocaríamos en el raíz, primero incluimos el *autoload*, importamos las clases a utilizar para finalmente usar los métodos de Monolog:

```php
<?php
    include __DIR__ ."/vendor/autoload.php";

    use Monolog\Logger;
    use Monolog\Handler\StreamHandler;
```

```
    $log->debug("Esto es un mensaje de DEBUG");
    $log->info("Esto es un mensaje de INFO");
    $log->warning("Esto es un mensaje de WARNING");
    $log->error("Esto es un mensaje de ERROR");
    $log->critical("Esto es un mensaje de CRITICAL");
    $log->alert("Esto es un mensaje de ALERT");
```

En todos los métodos de registro de mensajes (`debug`, `info`, ...), además del propio mensaje, le podemos pasar información como el contenido de alguna variable, usuario de la aplicación, etc.. como segundo parámetro dentro de un array, el cual se conoce como **array de contexto**. Es conveniente hacerlo mediante un array asociativo para facilitar la lectura del log.

```
<?php
    $log->warning("Producto no encontrado", [$producto]);
    $log->warning("Producto no encontrado", ["datos" => $producto]);
```

3.3. funcionamiento

Cada instancia `Logger` tiene un nombre de canal y una pila de manejadores (*handler*). Cada mensaje que mandamos al log atraviesa la pila de manejadores, y cada uno decide si debe registrar la información, y si se da el caso, finalizar la propagación. Por ejemplo, un `StreamHandler` en el fondo de la pila que lo escriba todo en disco, y en el tope añade un `MailHandler` que envíe un mail sólo cuando haya un error.

3.4. manejadores

Cada manejador tambien tiene un formateador (`Formatter`). Si no se indica ninguno, se le asigna uno por defecto. El último manejador insertado será el primero en ejecutarse. Luego se van ejecutando conforme a la pila.

Los manejadores más utilizados son:

- `StreamHandler(ruta, nivel)`
- `RotatingFileHandler(ruta, maxFiles, nivel)`
- `NativeMailerHandler(para, asunto, desde, nivel)`
- `FirePHPHandler(nivel)`

Si queremos que los mensajes de la aplicación salgan por el log del servidor, en nuestro caso el archivo `error.log` de *Apache* utilizaremos como ruta la salida de error:

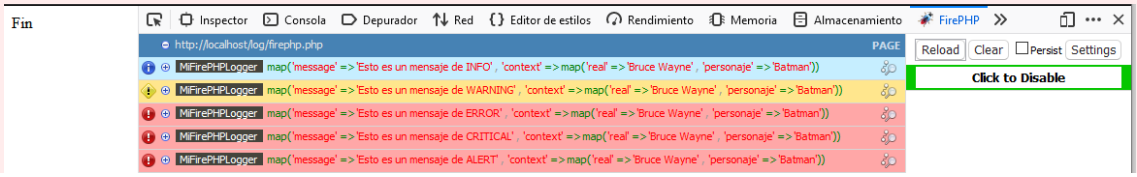
```
<?php
    // error.log
    $log->pushHandler(new StreamHandler("php://stderr", Logger::DEBUG));
```

FirePHP

Por ejemplo, mediante FirePHPHandler, podemos utilizar FirePHP, la cual es una herramienta para hacer debug en la consola de Firefox. Tras instalar la extensión en Firefox, habilitar las opciones y configurar el Handler, podemos ver los mensajes coloreados con sus datos:

```php
<?php
    $log = new Logger("MiFirePHPLogger");
    $log->pushHandler(new FirePHPHandler(Logger::INFO));

    $datos = ["real" => "Bruce Wayne", "personaje" => "Batman"];
    $log->debug("Esto es un mensaje de DEBUG", $datos);
    $log->info("Esto es un mensaje de INFO", $datos);
    $log->warning("Esto es un mensaje de WARNING", $datos);
```



3.5. canales

Se les asigna al crear el Logger. En grandes aplicaciones, se crea un canal por cada subsistema: ventas, contabilidad, almacén. No es una buena práctica usar el nombre de la clase como canal, esto se hace con un *procesador*.

Para su uso, es recomiendo asignar el log a una propiedad privada a Logger, y posteriormente, en el constructor de la clase, asignar el canal, manejadores y formato.

```php
<?php
    $this->log = new Logger("MiApp");
    $this->log->pushHandler(new StreamHandler("logs/milog.log", Logger::DEBUG));
    $this->log->pushHandler(new FirePHPHandler(Logger::DEBUG));
```

Y dentro de los métodos para escribir en el log:

```php
<?php
    $this->log->warning("Producto no encontrado", [$producto]);
```

3.6. procesadores

Los procesadores permiten añadir información a los mensajes. Para ello, se apilan después de cada manejador mediante el método `pushProcessor($procesador)`.

Algunos procesadores conocidos son `IntrospectionProcessor` (muestran la linea, fichero, clase y metodo desde el que se invoca el log), `WebProcessor` (añade la URI, método e IP) o `GitProcessor` (añade la rama y el commit).

PHP

```php
<?php
    $log = new Logger("MiLogger");
    $log->pushHandler(new RotatingFileHandler("logs/milog.log", 0,
Logger::DEBUG));
    $log->pushProcessor(new IntrospectionProcessor());
    $log->pushHandler(new StreamHandler("php://stderr", Logger::WARNING));
    // no usa Introspection pq lo hemos apilado después, le asigno otro
    $log->pushProcessor(new WebProcessor());
```

Consola en formato texto

```
[2020-11-26T13:35:31.076138+01:00] MiLogger.DEBUG: Esto es un mensaje de DEBUG []
{"file":"C:\\xampp\\htdocs\\log\\procesador.php","line":12,"class":null,"function
":null}
[2020-11-26T13:35:31.078344+01:00] MiLogger.INFO: Esto es un mensaje de INFO []
{"file":"C:\\xampp\\htdocs\\log\\procesador.php","line":13,"class":null,"function
":null}
```

3.7. formateadores

Se asocian a los manejadores con setFormatter. Los formateadores más utilizados son LineFormatter, HtmlFormatter o JsonFormatter.

PHP

```php
<?php
    $log = new Logger("MiLogger");
    $rfh = new RotatingFileHandler("logs/milog.log", Logger::DEBUG);
    $rfh->setFormatter(new JsonFormatter());
    $log->pushHandler($rfh);
```

Consola en JSON

```
{"message":"Esto es un mensaje de DEBUG","context":
{},"level":100,"level_name":"DEBUG","channel":"MiLogger","datetime":"2020-11-
27T15:36:52.747211+01:00","extra":{}}
{"message":"Esto es un mensaje de INFO","context":
{},"level":200,"level_name":"INFO","channel":"MiLogger","datetime":"2020-11-
27T15:36:52.747
```

más información

Más información sobre manejadores, formateadores y procesadores en https://github.com/Seldaek/monolog/blob/master/doc/02-handlers-formatters-processors.md

3.8. uso de factorías

En vez de instanciar un log en cada clase, es conveniente crear una factoría (por ejemplo, siguiendo la idea del patrón de diseño Factory Method).

Para el siguiente ejemplo, vamos a suponer que creamos la factoría en el `namespace Dwes\Ejemplos\Util`.

```php
<?php
    namespace Dwes\Ejemplos\Util

    use Monolog\Logger;
    use Monolog\Handler\StreamHandler;

    class LogFactory {
```

```
            $log = new Logger($canal);
            $log->pushHandler(new StreamHandler("logs/miApp.log",
Logger::DEBUG));

            return $log;
        }
    }
```

Si en vez de devolver un `Monolog\Logger` utilizamos el interfaz de PSR, si en el futuro cambiamos la implementación del log, no tendremos que modificar nuestro codigo. Así pues, la factoría ahora devolverá `Psr\Log\LoggerInterface`:

```
<?php
    namespace Dwes\Ejemplos\Util

    use Monolog\Handler\StreamHandler;
    use Monolog\Logger;
    use Psr\Log\LoggerInterface;

    class LogFactory {

        public static function getLogger(string $canal = "miApp") :
LoggerInterface {
            $log = new Logger($canal);
            $log->pushHandler(new StreamHandler("log/miApp.log", Logger::DEBUG));

            return $log;
        }
    }
```

Finalmente, para utilizar la factoría, sólo cambiamos el código que teníamos en el constructor de las clases que usan el log, quedando algo asi:

```
<?php
    namespace Dwes\Ejemplos\Util

    use Monolog\Handler\StreamHandler;
    use Monolog\Logger;
    use Psr\Log\LoggerInterface;

    class LogFactory {

        public static function getLogger(string $canal = "miApp") :
LoggerInterface {
            $log = new Logger($canal);
            $log->pushHandler(new StreamHandler("log/miApp.log", Logger::DEBUG));

            return $log;
        }
    }
```

4. referencias

- Tutorial de Composer
- Web Scraping with PHP – How to Crawl Web Pages Using Open Source Tools
- PHP Monolog
- Unit Testing con PHPUnit — Parte 1, de Emiliano Zublena.