

unidad didáctica 05

# Herramientas Web



## Índice

1. **duración y criterios de evaluación**
2. **Composer**
  2. 1. [instalación](#)
  2. 2. [primeros pasos](#)
  2. 3. [actualizar librerías](#)
  2. 4. [autoload.php](#)
3. **Monolog**
  3. 1. [niveles](#)
  3. 2. [Hola Monolog](#)
  3. 3. [funcionamiento](#)
  3. 4. [manejadores](#)
  3. 5. [canales](#)
  3. 6. [procesadores](#)
  3. 7. [formateadores](#)
  3. 8. [uso de factorías](#)
4. **Documentación con phpDocumentor**
  4. 1. [instalación como binario](#)
  4. 2. [uso del Docker](#)
  4. 3. [DocBlock](#)
  4. 4. [documentando el código](#)
5. **Web Scraping**
  5. 1. [Goutte](#)
  5. 2. [Goutte con selectores CSS](#)
  5. 3. [Crawler](#)
6. **Pruebas con PHPUnit**
  6. 1. [puesta en marcha](#)
  6. 2. [diseñando pruebas](#)
  6. 3. [aserciones](#)
  6. 4. [comparando la salida](#)
  6. 5. [proveedores de datos](#)
  6. 6. [probando excepciones](#)
  6. 7. [cobertura de código](#)
7. **referencias**

# 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:

```
1 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:

```
1 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.

```

1  {
2    "name": "dwes/log",
3    "description": "Pruebas con Monolog",
4    "type": "project",
5    "require": {
6      "monolog/monolog": "^2.1"
7    },
8    "license": "MIT",
9    "authors": [
10     {
11       "name": "Nombre Apellido", // ejemplo: Carmen Llopis
12       "email": "inicialNombre.Apellido@edu.gva.es" // ejemplo:
13         c.llopis@edu.gva.es
14     }
15   ]
16 }
```

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:

```
1 | composer require phpunit/phpunit -dev
```

Tras añadir nuevas librerías, hemos de actualizar nuestro proyecto:

```
1 | composer update
```

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

```
1 | 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:

```
1 <?php
2     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`:

```
1 "autoload": {
2     "psr-4": {"Dwes\\": "app/Dwes"}
3 },
```

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

```
1 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:

```
1 | 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:

```

1  <?php
2      include __DIR__ . "/vendor/autoload.php";
3
4      use Monolog\Logger;
5      use Monolog\Handler\StreamHandler;
6
7      $log = new Logger("MiLogger");
8      $log->pushHandler(new StreamHandler("logs/milog.log", Logger::DEBUG));
9
10     $log->debug("Esto es un mensaje de DEBUG");
11     $log->info("Esto es un mensaje de INFO");
12     $log->warning("Esto es un mensaje de WARNING");
13     $log->error("Esto es un mensaje de ERROR");
14     $log->critical("Esto es un mensaje de CRITICAL");
15     $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.

```

1  <?php
2      $log->warning("Producto no encontrado", [$producto]);
3      $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 también 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:



```

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

```

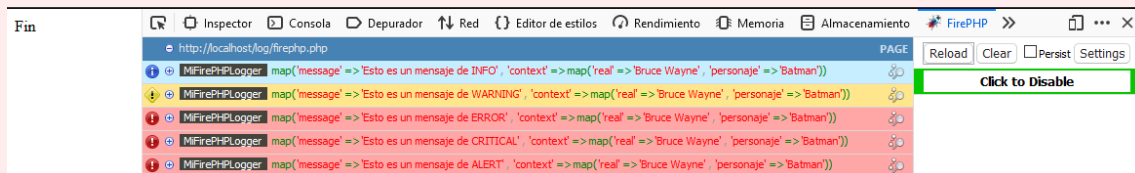
### FirePHP

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:

```

1 <?php
2     $log = new Logger("MiFirePHPLogger");
3     $log->pushHandler(new FirePHPHandler(Logger::INFO));
4
5     $datos = ["real" => "Bruce Wayne", "personaje" => "Batman"];
6     $log->debug("Esto es un mensaje de DEBUG", $datos);
7     $log->info("Esto es un mensaje de INFO", $datos);
8     $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 recomendando asignar el log a una propiedad privada a Logger, y posteriormente, en el constructor de la clase, asignar el canal, manejadores y formato.

```

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

```

Y dentro de los métodos para escribir en el log:

```

1 <?php
2     $this->log->warning("Producto no encontrado", [$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 línea, fichero, clase y método desde el que se invoca el log), `WebProcessor` (añade la URI, método e IP) o `GitProcessor` (añade la rama y el commit).

**PHP**

```

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

```

**Consola en formato texto**

```

1 [2020-11-26T13:35:31.076138+01:00] MiLogger.DEBUG: Esto es un mensaje de DEBUG
2 []
3 {"file":"C:\\xampp\\htdocs\\log\\procesador.php","line":12,"class":null,"function":null}
4 [2020-11-26T13:35:31.078344+01:00] MiLogger.INFO: Esto es un mensaje de INFO
5 []
6 {"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**

```

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

```

**Consola en JSON**

```

1 {"message":"Esto es un mensaje de DEBUG","context":
2 {}, "level":100, "level_name":"DEBUG", "channel":"MiLogger", "datetime":"2020-11-
3 27T15:36:52.747211+01:00", "extra":{}}
4 {"message":"Esto es un mensaje de INFO","context":
5 {}, "level":200, "level_name":"INFO", "channel":"MiLogger", "datetime":"2020-11-
6 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`.

```

1  <?php
2      namespace Dwes\Ejemplos\Util
3
4      use Monolog\Logger;
5      use Monolog\Handler\StreamHandler;
6
7      class LogFactory {
8
9          public static function getLogger(string $canal = "miApp") : Logger {
10              $log = new Logger($canal);
11              $log->pushHandler(new StreamHandler("logs/miApp.log", Logger::DEBUG));
12
13              return $log;
14          }
15      }

```

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 código. Así pues, la factoría ahora devolverá `Psr\Log\LoggerInterface`:

```

1  <?php
2      namespace Dwes\Ejemplos\Util
3
4      use Monolog\Handler\StreamHandler;
5      use Monolog\Logger;
6      use Psr\Log\LoggerInterface;
7
8      class LogFactory {
9
10         public static function getLogger(string $canal = "miApp") :
11         LoggerInterface {
12             $log = new Logger($canal);
13             $log->pushHandler(new StreamHandler("log/miApp.log", Logger::DEBUG));
14
15             return $log;
16         }
17     }

```

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

```

1  <?php
2      namespace Dwes\Ejemplos\Util
3
4      use Monolog\Handler\StreamHandler;
5      use Monolog\Logger;

```

```
6      use Psr\Log\LoggerInterface;
7
8      class LoggerFactory {
9
10         public static function getLogger(string $canal = "miApp") :
LoggerInterface {
11             $log = new Logger($canal);
12             $log->pushHandler(new StreamHandler("log/miApp.log", Logger::DEBUG));
13
14             return $log;
15         }
16     }
```

## 4. Documentación con phpDocumentor

[phpDocumentor](#) es la herramienta de facto para documentar el código PHP. Es similar en propósito y funcionamiento a *Javadoc*.

Así pues, es una herramienta que facilita la documentación del código PHP, creando un sitio web con el API de la aplicación.

Se basa en el uso de anotaciones sobre los docblocks. Para ponerlo en marcha, en nuestro caso nos decantaremos por utilizar la imagen que ya existe de Docker.

### 4.1. instalación como binario

Otra opción es seguir los pasos que recomienda la [documentación oficial](#) para instalarlo como un ejecutable, que son descargar el archivo `phpDocumentor.phar` y darles permisos de ejecución:

```
1 wget https://phpdoc.org/phpDocumentor.phar
2 chmod +x phpDocumentor.phar
3 mv phpDocumentor.phar /usr/local/bin/phpdoc
4 phpdoc --version
```

Una vez instalado, desde el raíz del proyecto, suponiendo que tenemos nuestro código dentro de `app` y que queremos la documentación dentro de `docs/api` ejecutamos:

```
1 phpdoc -d ./app -t docs/api
```

### 4.2. uso del Docker

En el caso de usar Docker, usaremos el siguiente comando para ejecutarlo (crea el contenedor, ejecuta el comando que le pidamos, y automáticamente lo borra):

```
1 docker run --rm -v "$(pwd)":/data phpdoc/phpdoc:3
```

A dicho comando, le adjuntaremos los diferentes parámetros que admite phpDocumentor, por ejemplo:

```
1 # Muestra la versión
2 docker run --rm -v "$(pwd)":/data phpdoc/phpdoc:3 --version
3 # Mediante -d se indica el origen a parsear
4 # Mediante -t se indica el destino donde generar la documentación
5 docker run --rm -v "$(pwd)":/data phpdoc/phpdoc:3 -d ./src/app -t ./docs/api
```

### 4.3. DocBlock

Un docblock es el bloque de código que se coloca encima de un recurso. Su formato es:

```
1 <?php
2 /*
3  * *Sumario*, una sola línea
4  *
5  * *Descripción* que puede utilizar varias líneas
```

```

6      * y que ofrece detalles del elemento o referencias
7      * para ampliar la información
8      *
9      * @param string $miArgumento con una *descripción* del argumento
10     * que puede usar varias líneas.
11     *
12     * @return void
13     */
14     function miFuncion(tipo $miArgumento)
15     {
16     }

```

## 4.4. documentando el código

En todos los elementos, además del sumario y/o descripción, pondremos:

- En las clases:
  - `@author` nombre
  - `@package` ruta del namespace
- En las propiedades:
  - `@var` tipo descripción
- En los métodos:
  - `@param` tipo \$nombre descripción
  - `@throws` ClaseException descripción
  - `@return` tipo descripción

Veámoslo con un ejemplo. Supongamos que tenemos una clase que representa un cliente:

```

1  <?php
2      /*
3       * Clase que representa un cliente
4       *
5       * El cliente se encarga de almacenar los soportes que tiene alquilado,
6       * de manera que podemos alquilar y devolver productos mediante las
7       * operaciones
8       *
9       * @package Dwes\Videoclub\Model
10      * @author Aitor Medrano <a.medrano@edu.gva.es>
11      */
12      class Cliente {
13
14          public string $nombre;
15          private string $numero;
16
17          /*
18           * Colección de soportes alquilados
19           * @var array<Soporte>
20           */
21          private $soportesAlquilados[];
22
23

```

```

24      /*
25       * Comprueba si el soporte recibido ya lo tiene alquilado el cliente
26       * @param Soporte $soporte Soporte a comprobar
27       * @return bool true si lo tiene alquilado
28       */
29      public function tieneAlquilado(Soporte $soporte) : bool {
30          // ...
31      }

```

Si generamos la documentación y abrimos con un navegador el archivo `docs/api/index.html` podremos navegar hasta la clase `Cliente`:

## Documentation

### Packages

- Application
- Dwes
- Videoclub

### Reports

- Class Diagram
- Deprecated
- Errors
- Markers

### Indices

- Files

Search (Press "/" to focus)

Dwes \ Videoclub \ Model

## Cliente

[Cliente.php](#) : 13

in package Dwes \ Videoclub \ Model

*Clase que representa un cliente*

El cliente se encarga de almacenar los soportes que tiene alquilado, de manera que podemos alquilar y devolver productos mediante las operaciones homónimas.

### Tags

author

Aitor Medrano [a.medrano@edu.gva.es](mailto:a.medrano@edu.gva.es)

## Table of Contents

- P** [\\$nombre](#) : string
- P** [\\$numero](#) : string
- P** [\\$soportesAlquilados](#) : array<string|int, mixed>
- M** [tieneAlquilado\(\)](#) : bool  
*Comprueba si el soporte recibido ya lo tiene alquilado el cliente*

## Properties

### \$nombre

[Cliente.php](#) : 16

```
public string $nombre
```

Nombre del cliente

## 5. Web Scraping

Consiste en navegar a una página web y extraer información automáticamente, a partir del código HTML generado, y organizar la información pública disponible en Internet.

Esta práctica requiere el uso de una librería que facilite la descarga de la información deseada imitando la interacción de un navegador web. Este "robot" puede acceder a varias páginas simultáneamente.

### es legal?

Si el sitio web indica que tiene el contenido protegido por derechos de autor o en las normas de acceso via usuario/contraseña nos avisa de su prohibición, estaríamos incurriendo en un delito. Es recomendable estudiar el archivo `robots.txt` que se encuentra en el raíz de cada sitio web. Más información en el artículo [El manual completo para el web scraping legal y ético en 2021](#).

### 5.1. Goutte

[Goutte](#) es un sencillo cliente HTTP para PHP creado específicamente para hacer web scraping. Lo desarrolló el mismo autor del framework Symfony y ofrece un API sencilla para extraer datos de las respuestas HTML/XML de los sitios web.

**Datos:** Revisar <https://godofredo.ninja/web-scraping-con-php-utilizando-goutte/>

Los componentes principales que abstrae Goutte sobre Symfony son:

- `BrowserKit` : simula el comportamiento de un navegador web.
- `CssSelector` : traduce consultas CSS en consultas XPath.
- `DomCrawler` : facilita el uso del DOM y XPath.

Para poder utilizar Goutte en nuestro proyecto, ejecutaremos el siguiente comando en el terminal:

```
1 composer require fabpot/goutte
```

### 5.2. Goutte con selectores CSS

A continuación vamos a hacer un ejemplo muy sencillo utilizando los selectores CSS, extrayendo información de la web <https://books.toscrape.com/>, la cual es una página para pruebas que no rechazará nuestras peticiones.

Tras crear un cliente con Goutte, hemos de realizar un petición a una URL. Con la respuesta obtenida, podemos utilizar el método `filter` para indicarle la ruta CSS que queremos recorrer e iterar sobre los resultados mediante una función anónima. Una vez estamos dentro de un determinado nodo, el método `text()` nos devolverá el contenido del propio nodo.

En concreto, vamos a meter en un array asociativo el título y el precio de todos los libros de la categoría *Classics*.



```

1  <?php
2      require '../vendor/autoload.php';
3
4      $httpClient = new \Goutte\Client();
5      $response = $httpClient->request('GET',
6      'https://books.toscrape.com/catalogue/category/books/classics_6/index.html');
7      // colocamos los precios en un array
8      $precios = [];
9      $response->filter('.row li article div.product_price p.price_color')->each(
10         // le pasamos $precios por referencia para poder editarla dentro del
11         closure
12         function ($node) use (&$precios) {
13             $precios[] = $node->text();
14         }
15     );
16
17     // colocamos el nombre y el precio en un array asociativo
18     $contadorPrecios = 0;
19     $libros = [];
20     $response->filter('.row li article h3 a')->each(
21         function ($node) use ($precios, &$contadorPrecios, &$libros) {
22             $libros[$node->text()] = $precios[$contadorPrecios];
23             $contadorPrecios++;
24         }
25     )

```

### 5.3. Crawler

Un caso muy común es obtener la información de una página que tiene los resultados paginados, de manera que vayamos recorriendo los enlaces y accediendo a cada uno de los resultados.

En este caso vamos a coger todos los precios de los libros de fantasía, y vamos a sumarlos:

```

1  <?php
2      require '../vendor/autoload.php';
3
4      use Goutte\Client;
5      use Symfony\Component\HttpClient\HttpClient;
6
7      $client = new Client(HttpClient::create(['timeout' => 60]));
8      $crawler = $client->request('GET',
9      'https://books.toscrape.com/catalogue/category/books/fantasy_19/index.html');
10
11     $salir = false;
12
13     $precios = [];
14     while (!$salir) {
15         $crawler->filter('.row li article div.product_price p.price_color')->each(
16             function ($node) use (&$precios) {
17                 $texto = $node->text();
18                 $cantidad = substr($texto, 2); // Le quitamos las libras ¿2
19                 posiciones?
20                 $precios[] = floatval($cantidad);
21             }
22         )
23     }

```

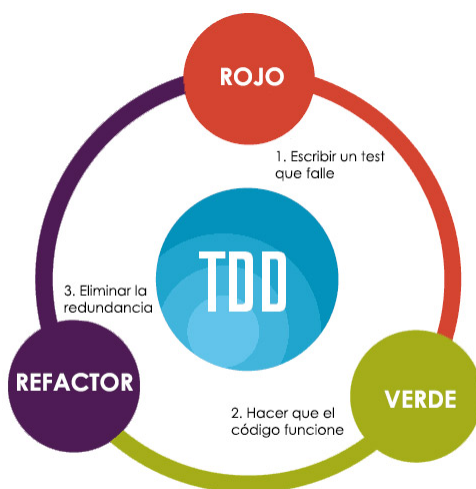
```
20     );
21
22     $enlace = $crawler->selectLink('next');
23     if ($enlace->count() != 0) {
24         // el enlace next existe
25         $sigPag = $crawler->selectLink('next')->link();
26         $crawler = $client->click($sigPag); // hacemos click
27     } else {
28         // ya no hay enlace next
29         $salir = true;
30     }
31 }
32
33 $precioTotal = array_sum($precios);
34 echo $precioTotal;
```

## 6. Pruebas con PHPUnit

El curso pasado, dentro del módulo de Entornos de Desarrollo, estudiamos la importancia de la realización de pruebas, así como las pruebas unitarias mediante [JUnit](#).

A día de hoy es de gran importancia seguir una buena metodología de pruebas, siendo el desarrollo dirigido por las pruebas (Test Driven Development / TDD) uno de los enfoques más empleados, el cual consiste en:

1. Escribir el test, y como no hay código implementado, la prueba falle (rojo).
2. Escribir el código de aplicación para que la prueba funcione (verde).
3. Refactorizar el código de la aplicación con la ayuda de la prueba para comprobar que no rompemos nada (refactor).



En el caso de PHP, la herramienta que se utiliza es PHPUnit (<https://phpunit.de/>), que como su nombre indica, está basada en JUnit. La versión actual es la 9.0

Se recomienda consultar su documentación en <https://phpunit.readthedocs.io/es/latest/index.html>.

### 6.1. puesta en marcha

Vamos a colocar todas las pruebas en una carpeta tests en el raíz de nuestro proyecto.

En el archivo composer.json, añadimos:

```
1  "require-dev": {
2      "phpunit/phpunit": "^9"
3  },
4  "scripts": {
5      "test": "phpunit --testdox --colors tests"
6  }
```

Si quisiéramos añadir la librería desde un comando del terminal, también podríamos ejecutar:

```
1  composer require --dev phpunit/phpunit ^9
```

## librerías de desarrollo

Las librerías que se colocan en `require-dev` son las de desarrollo y *testing*, de manera que no se instalarán en un entorno de producción.

Como hemos creado un script, podemos lanzar las pruebas mediante:

```
1 composer test
```

Vamos a realizar nuestra primera prueba:

```
1 <?php
2     use PHPUnit\Framework\TestCase;
3
4     class PilaTest extends TestCase
5     {
6         public function testPushAndPop()
7         {
8             $pila = [];
9             $this->assertSame(0, count($pila));
10
11             array_push($pila, 'batman');
12             $this->assertSame('batman', $pila[count($pila)-1]);
13             $this->assertSame(1, count($pila));
14
15             $this->assertSame('batman', array_pop($pila));
16             $this->assertSame(0, count($pila));
17         }
18     }
```

Tenemos diferentes formas de ejecutar una prueba:

```
1 ./vendor/bin/phpunit tests/PilaTest.php
2 ./vendor/bin/phpunit tests
3 ./vendor/bin/phpunit --testdox tests
4 ./vendor/bin/phpunit --testdox --colors tests
```

## 6.2. diseñando pruebas

Tal como hemos visto en el ejemplo, la clase de prueba debe heredar de `TestCase`, y el nombre de la clase debe acabar en `Test`, de ahí que hayamos llamado la clase de prueba como `PilaTest`.

Una prueba implica un método de prueba (público) por cada funcionalidad a probar. Cada uno de los métodos se les asocia un caso de prueba.

Los métodos deben nombrarse con el prefijo `test`, por ejemplo, `testPushAndPop`. Es muy importante que el nombre sea muy claro y descriptivo del propósito de la prueba. (*camelCase*).

En los casos de prueba prepararemos varias aserciones para toda la casuística: rangos de valores, tipos de datos, excepciones, etc...

## 6.3. aserciones

Las aserciones permiten comprobar el resultado de los métodos que queremos probar. Las aserciones esperan que el predicado siempre sea verdadero.

PHPUnit ofrece las siguiente aserciones:

- `assertTrue` / `assertFalse`: Comprueba que la condición dada sea evaluada como *true* / *false*.
- `assertEquals` / `assertSame`: Comprueba que dos variables sean iguales.
- `assertNotEquals` / `assertNotSame`: Comprueba que dos variables NO sean iguales.
  - `Same` → comprueba los tipos. Si no coinciden los tipos y los valores, la aserción fallará.
  - `Equals` → sin comprobación estricta.
- `assertArrayHasKey` / `assertArrayNotHasKey`: Comprueba que un array posea un key determinado / o NO lo posea
- `assertArraySubset`: Comprueba que un array posea otro array como subset del mismo
- `assertAttributeContains` / `assertAttributeNotContains`: Comprueba que un atributo de una clase contenga una variable determinada / o NO contenga una variable determinada
- `assertAttributeEquals`: Comprueba que un atributo de una clase sea igual a una variable determinada.

## 6.4. comparando la salida

Si los métodos a probar generan contenido mediante `echo` o una instrucción similar, disponemos de las siguiente expectativas:

- `expectOutputString(salidaEsperada)`
- `expectOutputRegex(expresionRegularEsperada)`

Las expectativas difieren de las aserciones que informan del resultado que se espera antes de invocar al método. Tras definir la expectativa, se invoca al método que realiza el `echo` / `print`.

```

1  <?php
2      namespace Dwes\Videoclub\Model;
3
4      use PHPUnit\Framework\TestCase;
5      use Dwes\Videoclub\Model\CintaVideo;
6
7      class CintaVideoTest extends TestCase {
8          public function testConstructor()
9          {
10             $cinta = new CintaVideo("Los cazafantasmas", 23, 3.5, 107);
11             $this->assertSame( $cinta->getNumero(), 23);
12         }
13
14         public function testMuestraResumen()
15         {
16             $cinta = new CintaVideo("Los cazafantasmas", 23, 3.5, 107);
17             $resultado = "<br>Película en VHS:";

```

```

18     $resultado .= "<br>Los cazafantasmas<br>3.5 (IVA no incluido)";
19     $resultado .= "<br>Duración: 107 minutos";
20     // definimos la expectativa
21     $this->expectOutputString($resultado);
22     // invocamos al método que hará echo
23     $cinta->muestraResumen();
24 }
25 }

```

## 6.5. proveedores de datos

Cuando tenemos pruebas que solo cambian respecto a los datos de entrada y de salida, es útil utilizar proveedores de datos.

Se declaran en el docblock mediante `@dataProvider nombreMetodo`, donde se indica el nombre de un método público que devuelve un array de arrays, donde cada elemento es un caso de prueba.

La clase de prueba recibe como parámetros los datos a probar y el resultado de la prueba como último parámetro.

El siguiente ejemplo comprueba con diferentes datos el funcionamiento de `muestraResumen`:

```

1  <?php
2  /*
3   * @dataProvider cintasProvider
4   */
5  public function testMuestraResumenConProvider($titulo, $id, $precio,
6  $duracion, $esperado)
7  {
8      $cinta = new CintaVideo($titulo, $id, $precio, $duracion);
9      $this->expectOutputString($esperado);
10     $cinta->muestraResumen();
11 }
12
13 public function cintasProvider() {
14     return [
15         "cazafantasmas" => ["Los cazafantasmas", 23, 3.5, 107, "<br>VHS:
16         <br>Los cazafantasmas<br>3.5 €(IVA no incluido)<br>Duración: 107 min"],
17         "superman" => ["Superman", 24, 3, 188, "<br>VHS:<br>Superman<br>3 €
18         (IVA no incluido)<br>Duración: 188 min"],
19     ];
20 }

```

## 6.6. probando excepciones

Las pruebas además de comprobar que las clases funcionan como se espera, han de cubrir todos los casos posibles. Así pues, debemos poder hacer pruebas que esperen que se lance una excepción (y que el mensaje contenga cierta información):

Para ello, se utilizan las siguiente expectativas:

- `expectException(Exception::class)`
- `expectExceptionCode(codigoExcepcion)`
- `expectExceptionMessage(mensaje)`

Del mismo modo que antes, primero se pone la expectativa, y luego se provoca que se lance la excepción:

```

1  <?php
2      public function testAlquilarCupoLleno() {
3          $soporte1 = new CintaVideo("Los cazafantasmas", 23, 3.5, 107);
4          $soporte2 = new Juego("The Last of Us Part II", 26, 49.99, "PS4", 1, 1);
5          $soporte3 = new Dvd("Origen", 24, 15, "es,en,fr", "16:9");
6          $soporte4 = new Dvd("El Imperio Contraataca", 4, 3, "es,en", "16:9");
7
8          $cliente1 = new Cliente("Bruce Wayne", 23);
9          $cliente1->alquilar($soporte1);
10         $cliente1->alquilar($soporte2);
11         $cliente1->alquilar($soporte3);
12
13         $this->expectException(CupoSuperadoException::class);
14         $cliente1->alquilar($soporte4);
15     }

```

## 6.7. cobertura de código

La cobertura de pruebas indica la cantidad de código que las pruebas cubren, siendo recomendable que cubran entre el 95 y el 100%.

Una de las métricas asociadas a los informes de cobertura es el CRAP (Análisis y Predicciones sobre el Riesgo en Cambios), el cual mide la cantidad de esfuerzo, dolor y tiempo requerido para mantener una porción de código. Esta métrica debe mantenerse con un valor inferior a 5.

### requisito xdebug

Aunque ya viene instalado dentro de PHPUnit, para que funcione la cobertura del código, es necesario que el código PHP se ejecute con XDEBUG, y se le indique a Apache que así es (colocando en el archivo de configuración `php.ini` la directiva `xdebug.mode=coverage`).

Añadimos en `composer.json` un nuevo script:











```
1  "coverage": "phpunit --coverage-html coverage --coverage-filter app tests"
```

Y posteriormente ejecutamos

```
1  composer coverage
```

Por ejemplo, si accedemos a la clase `CintaVideo` con la prueba que habíamos realizado anteriormente, podemos observar la cobertura que tiene al 100% y que su CRAP es 2.

/var/www/html/app/Dwes/Videoclub / Model / CintaVideo.php

	Code Coverage									
	Classes and Traits			Functions and Methods				Lines		
Total		100.00%	1 / 1		100.00%	2 / 2	<u>CRAP</u>		100.00%	7 / 7
CintaVideo		100.00%	1 / 1		100.00%	2 / 2	2		100.00%	7 / 7
__construct					100.00%	1 / 1	1		100.00%	3 / 3
muestraResumen					100.00%	1 / 1	1		100.00%	4 / 4

```

1  <?php
2
3  namespace Dwes\Videoclub\Model;
4
5  class CintaVideo extends Soporte {
6      private $duracion;
7
8      function __construct($tit,$num,$precio,$duracion){
9          parent::__construct($tit,$num,$precio);
10         $this->duracion = $duracion;
11     }
12
13     public function muestraResumen(){
14         echo "<br>Película en VHS:";
15         parent::muestraResumen();
16         echo "<br>Duración: ".$this->duracion." minutos";
17     }
18 }
```

## Legend

Executed Not Executed Dead Code

Generated by [php-code-coverage 9.2.7](#) using [PHP 8.0.11](#) and [PHPUnit 9.5.10](#) at Sat Oct 23 18:10:26 CEST 2021.

## temas pendientes

- Dependencia entre casos de prueba con el atributo `@depends`
- Completamente configurable mediante el archivo `phpxml.xml`: <https://phpunit.readthedocs.io/es/latest/configuration.html>
- Preparando las pruebas con `setUpBeforeClass()` y `tearDownAfterClass()`
- Objetos y pruebas `Mock` (dobles) con `createMock()`



## 7. 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.