ENI Service - WordPress - PHP

Paul Schuhmacher

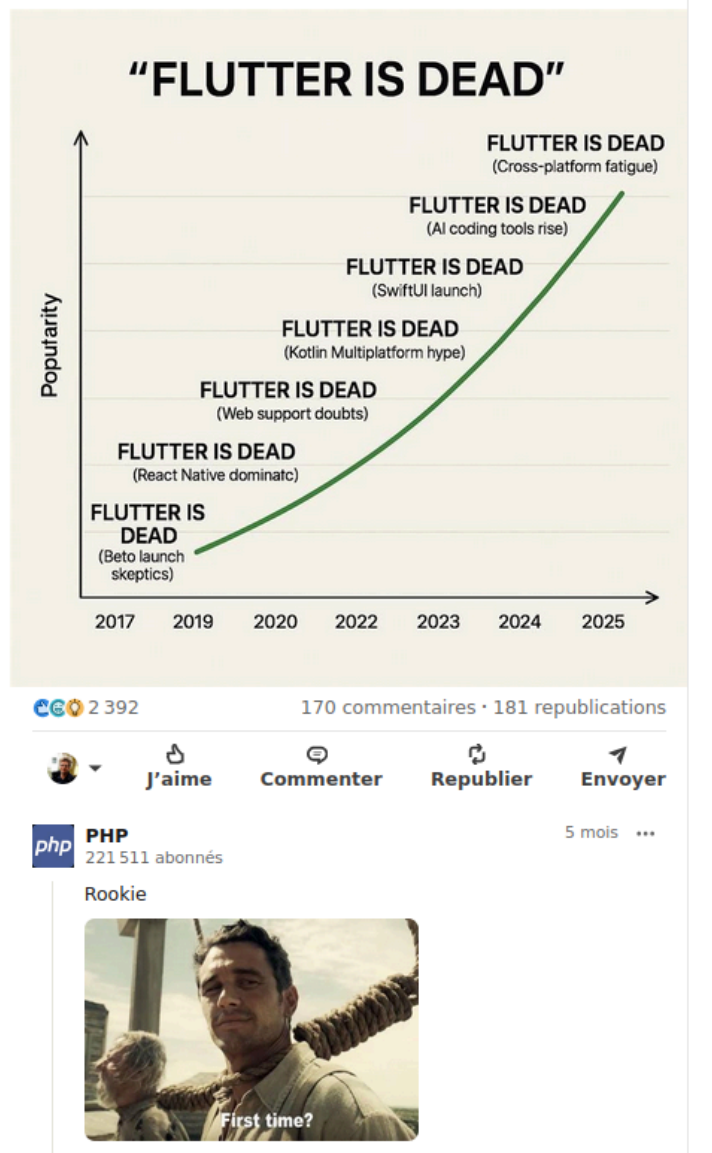
Durée : 28h

Novembre 2025

Partie 2 : PHP pour Wordpress

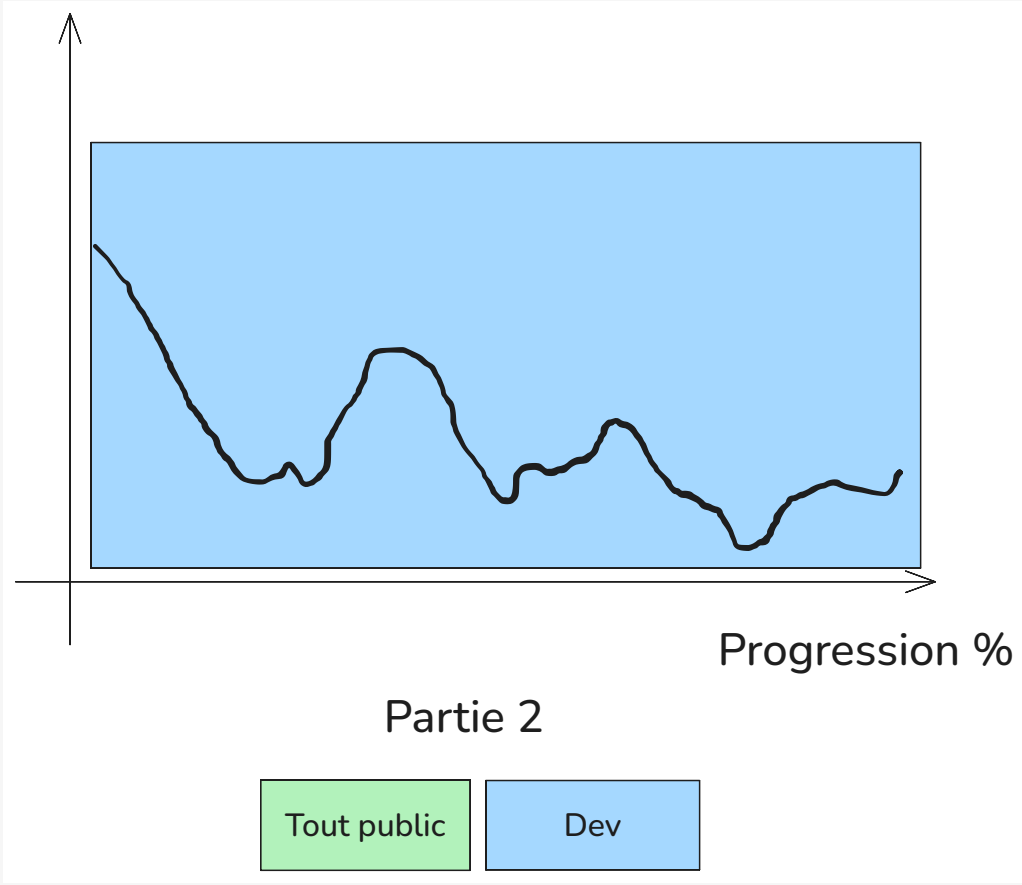
{.marp-bg-img}

> Source du post



Partie 2 : PHP pour Wordpress

{.marp-bg-img}



Partie 2 : PHP pour WordPress

Objectif : mise à niveau sur *PHP moderne*, orienté WordPress.

- **(Re)voir les bases** :
  - Fonctionnement de PHP (machine virtuelle), typage dynamique
  - Rappels PHP pour développeurs venant du .NET, les **primitives** de PHP : variables, tableaux, objets, fonctions
  - Programmation Orientée Objet en PHP
  - **Pratique**
- **PHP Moderne** (PHP >= 5)
  - Gestion des dépendances : namespaces et *autoloading* (norme PSR-4), **Composer** et Packagist, installer et utiliser des composants (paquets)
  - Nouveautés de PHP 8.4 et 8.5
  - **Pratique** : utiliser des composants dans un site WordPress
- **PHP appliqué à WordPress** :
  - Adopter les guidelines/standards/pratiques de WordPress
  - Loop, *Hooks*
  - **Pratique**
  - Ressources pour aller plus loin.

PHP (re)voir les bases

> À parcourir en fonction des compétences à consolider.

Portrait général de PHP

- PHP est utilisable sur tous les OS les plus courants (Windows, macOS, GNU/Linux, UNIX);
- PHP supporte de nombreux paradigmes de programmation : **procédural**, **orienté objet**, **fonctionnel**;
- PHP supporte **toutes les bases de données** (API `PHP Data Objects (PDO)`);
- PHP supporte et permet d'utiliser de nombreux **protocoles** de l'internet (HTTP, SMTP, FTP, etc.);
- Le moteur de PHP (*virtual machine*) dispose de nombreuses extensions pour étendre ses fonctionnalités;
- PHP dispose d'un gestionnaire de dépendances et de dépôts associés;
- PHP dispose de **plusieurs modes d'exécution configurables** (*SAPI*) pour le web : (mode CGI (cli), module Apache (`mod_php`, *déprécié*), PHP-FPM, FrankenPHP) le petit nouveau !

D'où vient PHP ?

- 1994 : PHP est crée par Rasmus Lerdorf (Groenland) comme une bibliothèque logicielle écrite en C pour faciliter le développement de son site web (*Personal Home Page*). PHP passe rapidement open source;
- 1997 : PHP3 est crée (refonte totale du moteur). Développé par Gutmans et Suraski, moteur Zend. Maintenu et supporté par la PHP Foundation et Zend;
- Novembre 2025 : PHP 8.5, version actuelle de PHP !

Plusieurs implémentations de PHP

- Zend Engine : une implémentation de PHP open source, par Zend Technologies. C'est celle que l'on va utiliser. Essayer `php -v` pour afficher la version de PHP et du Zend Engine;
- Hip Hop Virtual Machine for PHP (HHVM) : développé par Méta/Facebook, avec le langage Hack, un dialecte PHP fortement typé

PHP dans le monde réel

> PHP fait tourner le web mondial !

- En 2022, environ 80% des sites web recensés sont *powered* par PHP (avec Wordpress notamment);
- PHP est en constante évolution et maturation. Écosystème **riche**, communautés **actives** :
  - PHP Framework Inter Group (PHP-FIG),
  - PHP Foundation,
  - AFUP,
  - Nombreux frameworks : Symfony, Laravel, API Platform, MediaWiki, WordPress?, WooCommerce, bbpress, phpBB, etc.

Qui utilise PHP ?

- Wikipédia ! Avec le moteur de wiki MédiaWiki, écrit en PHP. La plupart des moteurs de Wiki sont implémentés en PHP (Dokuwiki);
- Meta (Facebook), Etsy, Spotify, Canva, Slack, etc;
- Quasiment tous les **moteurs de forum** (phpBB, bbpress, etc.);
- **Content Managment System** (CMS) ! **Wordpress**, **Drupal**, Joomla, Magento, etc.
- **Administration de base de données** (phpMyAdmin, Adminer, etc.);
- Etc.

Installer et tester PHP

Suivre les instructions données sur le site officiel en fonction de l'OS de votre machine.

Suivre le tutoriel officiel.

Votre premier script PHP

Créer un fichier `hello-world.php` :

```
hello, world
```

Exécuter :

```
php hello-world.php
```

Votre premier script PHP avec du code php

- Peu importe son environnement d'exécution, le code PHP est **toujours** inclus entre une balise de ouvrante `<?php` et une balise fermante `?>`.
- Voici le contenu d'un fichier `index.php` :

```
<html>
    <body>
        <?php echo "hello, world"; ?>
    </body>
</html>
```

- Toute instruction PHP se place entre les balises PHP et **se termine par un point-virgule** :

```
<?php echo "hello, world"; ?>
```

- **Exéxuter** le script `index.php` :

```
php index.php
```

- **Observer** la sortie.

Que s'est-il passé ?

Un script PHP s'exécute sur la machine virtuelle ou interpréteur PHP. Il analyse le fichier *byte par byte* :

- Tant qu'il ne rencontre aucune balise PHP, les caractères sont considérés comme des **données brutes** et sont écrits sur la sortie standard (*stdout*) **sans être interprétés**;
- Lorsque l'interpréteur rencontre une balise ouvrante (`<?php`), il passe en **mode interprétation**. Le code jusqu'à la balise fermante (`?>`) est alors :
  - **Analysé** (*Tokenizing, Parsing*): Le **code est vérifié** pour sa syntaxe et **transformé** en une séquence d'instructions appelée Abstract Syntax Tree (AST);
  - **Compilé** (en *bytecode*): L'AST est ensuite **compilé** en **bytecode** appelé **Opcode**;
  - **Exécuté**: Le moteur d'exécution Zend **exécute** l'**Opcode**. Résultat écrit sur *stdout*.

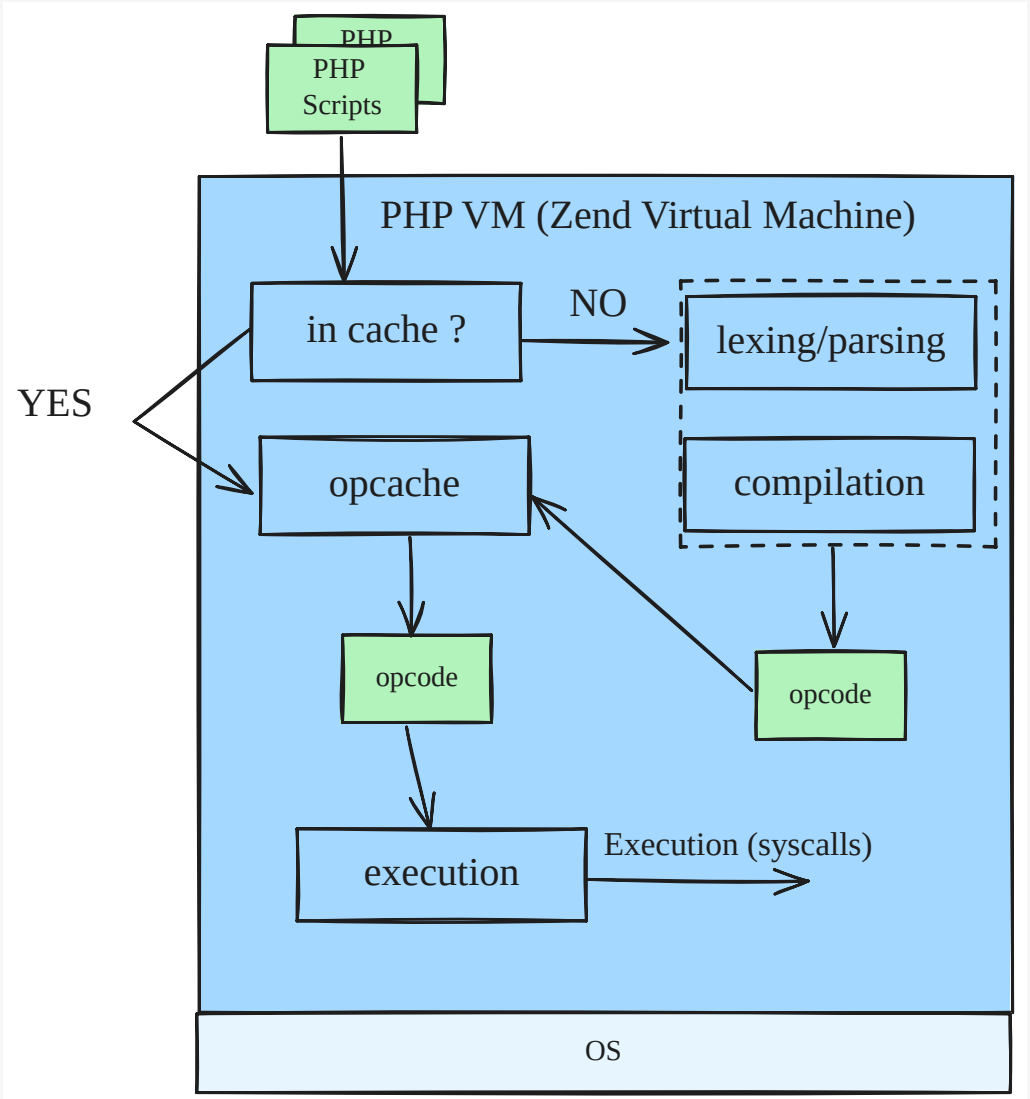
Grâce à son fonctionnement, PHP est parfaitement adapté pour **fabriquer des documents dynamiques** (contenu dynamique placé entre balises PHP) : **pages HTML**, fichiers texte ASCII, PDF, JSON, XML, email, etc.

Modèle d’exécution

{.marp-bg-img}

- Chaque **script** s'exécute dans son **propre processus ou contexte**.
- À la **fin** de **l’exécution**, toutes les variables, objets et **ressources allouées** sont automatiquement **libérées**.
- Il n’y a **pas de mémoire partagée entre les requêtes** (nécessite un cache, fichiers, session ou une base de données).

> Il existe des SAPI PHP comme PHP-FPM ou FrankenPHP qui permettent d'optimiser l'usage des processus PHP (pool de processus workers qui restent en mémoire pour traiter plusieurs requêtes successives), mais ces principes restent valables. Il est également possible de faire du multithread (CLI) avec les extensions pthreads ou parallel



PHP VM (Sans JIT)

Machine virtuelle PHP

La machine virtuelle PHP, dont le cœur est le Zend Engine, est conçue pour être à la fois hautement **configurable**, **modulaire** et **adaptable à différents contextes d'exécution**.

La machine virtuelle PHP :

- se **configure** via des **fichiers INI** (`php -i` pour afficher configuration actuelle);
- est **étendue** par un grand nombres d'**extensions** (modules) (`php -m` pour lister modules installés);
- possède **différentes API**, *Server Application Programming Interface (**SAPI**)*, interface d'intégration avec l'environnement hôte. Détermine l'architecture d'exécution, entrée/sorties, et la **gestion des processus** de la machine virtuelle PHP :
  - CLI/CGI,
  - Module Apache (`mod_php`), déprécié,
  - *FastCGI Process Manager* (PHP-FPM),
  - FrankenPHP, le petit nouveau (développé par Kevin Dunglas).
- On reviendra pus tard dans la formation sur la configuration de PHP (*dev* vs *prod*)

> - Quel fichier ini est utilisé par PHP ? Cela dépend justement de la SAPI !

Configurer PHP au runtime

Via des fichiers `php.ini` (format INI)

Dans un script, utiliser la fonction `phpinfo()` :

```php
<?php
//Afficher toute la configuration de PHP. Très pratique pour voir les fichiers ini chargés notamment.
phpinfo();
```

Ou directement depuis l'invite de commandes :

```
# Affiche toutes les informations (à grep)
php -i
# N'afficher que les fichiers de config chargés
php -i | grep -A15 "Configuration File (php.ini)"
# Avec PHP 8.5 ! Affiche directement les valeurs qui diffèrent des valeurs par défaut.
php --ini
php --ini=diff
```

Il existe de nombreuses directives pour configurer PHP à l'exécution, utiles notamment pour adapter le comportement en fonction des environnements.

PHP (SAPI CLI)

Exécuter directement le script dans le terminal :

```
php votre-script.php
```

La sortie (résultat) sera affichée dans le terminal.

PHP (SAPI CLI)

Exécuter directement le script dans le terminal, en chargeant une configuration personnalisée :

```
php -c mon-fichier.ini votre-script.php
```

Serveur web intégré de PHP (SAPI CLI)

PHP dispose d'un **serveur web intégré**, utilisant la SAPI CLI.

1. Ouvrir un terminal,
2. Se placer à la racine du projet, du fichier ou dossier à *servir comme un site web* (au même niveau) :

```
cd chemin/des/fichiers/php/a/servir
#Servir tout le contenu du dossier courant (attend un fichier index.php !)
php -S localhost:3000
#Servir uniquement le fichier 'monsite.php'
php -S localhost:3000 monsite.php
#Servir tout le dossier 'public'
php -S localhost:3000 -t public
```

> Le serveur web intégré de PHP **attend un fichier `index.php` par défaut**. Penser à toujours utiliser un fichier `index.php` comme point d'entrée de votre site web.

Exemple

1. **Créer** un dossier `public` ;
2. Dans le dossier `public`, **créer** un fichier `index.php` avec le contenu suivant :

```php
<?php
echo "<h1>Hello, world</h1>";
```

3. À la racine, **servir** tout le dossier avec le serveur intégré de PHP sur, par exemple, le port 8000 :

```
php -S localhost:8000 -t public
```

4. Avec votre navigateur favori, **se rendre** à l'URL `http://localhost:8000`. Vous devriez voir le résultat de l'execution du script `index.php`.

Syntaxe et features du langage

Les variables en PHP

- En PHP, les variables sont représentées par un signe dollar "$" suivi du nom de la variable ;
- **Le nom est sensible à la casse** ;
- Un nom de variable valide **commence par une lettre** (A-Z, a-z) ou par **un tiret bas**, suivi par une lettre ou un souligné (_), suivi de **lettres**, **chiffres** ou **soulignés** ;
- Il n'est **pas nécessaire de déclarer une variable** avant de lui affecter une valeur.

```php
//La variable $message contient la valeur 'hello, world' de type 'string'
$string = 'hello, world';
//Un tableau (collection de valeurs)
$array = [1, 2, 3];
$int = 12;
$float = -1.5;

$_nom_variable_ok = 'foo';
$_123 = 'ok';
$12 = 'NON'; //Erreur
$a = 1;
$A = 2; // $a != $A
```

PHP est un langage typé dynamiquement

En PHP, il n'est pas nécessaire de définir le type de données référencée par une variable, le type est *déduit* au *runtime* :

```php
$message = "hello, world";
//Imprimer la valeur de la variable sur la sortie
echo $message;
//Imprimer le type de la variable sur la sortie
echo gettype($message);
echo gettype(new stdClass());
//Afficher le contenu et le type de la variable $message
var_dump($message);
```

Types principaux : `boolean`, `int`, `double`, `string`, `array`, `object`, `NULL` (absence de valeur), `callable`.

Voir tous les types du langage PHP.

Inspecter

Pour **inspecter** le contenu d'une variable et/ou son type. Utile pour **débuger** et comprendre son programme et regarder les valeurs

```php
$name = "Jane Doe";
//Ecrire la valeur sur la sortie
echo $name;
$grades = [12, 10, 19, 8];
//Imprimer le contenu d'un tableau de manière lisible
print_r($grades);
//Imprimer le contenu de n'importe quelle variable et des types des données
var_dump($grades);
// Une variable non initialisée et non référencée (pas de contexte d'utilisation).
var_dump($unset_var);
```

Sur les opérateurs PHP

- Même règles de précédence et d'associativité que la plupart des langages;
- **Ne pas utiliser** `AND` et `OR` :
  - Pas les mêmes précédences que `&&` et `||` !
  - **Précédence plus faible que les opérateurs d'affectation** !

```php
$c = 8 > 5 AND 15 < 10 //$c vaut true ! car évalué comme ($c = (8 > 5)) AND (15 < 10)
```

- **Opérateur ternaire** :

```php
$action = (empty($_POST['action'])) ? 'default' : $_POST['action'];
```

Opérateurs Null coalescing

Null coalescing operator `??`

```php
// if it does not exist.
$username = $_GET['user'] ?? 'nobody';
// This is equivalent to:
$username = isset($_GET['user']) ? $_GET['user'] : 'nobody';
```

Opérateurs de comparaison

Comparaison :

- `==` : (*essaie* de convertir dans même type et) compare les valeurs;
- `===` : compare les *valeurs* **et** les *types*.

```php
<?php
//Opérateurs de comparaison :
//@link https://www.php.net/manual/en/language.operators.comparison.php

if(1 == '1') echo "1 == '1' est VRAI" . PHP_EOL;
if(1 === '1') echo "1 === '1' est FAUX" . PHP_EOL;
if(1 !== '1') echo "1 !== '1' est VRAI". PHP_EOL;
```

Opérateurs de comparaison et float

Comme dans tout language implémentant le standard IEEE 754 double precision format, **ne jamais comparer les nombres flottants directement**, définir et utiliser une **précision** :

```php
<?php
$a = 1.23456789;
$b = 1.23456780;
//Précision
$epsilon = 0.00001;

if (abs($a - $b) < $epsilon) {
    echo "$a est égal à $b";
}
```

Constantes

Pour créer des **constantes**, utiliser la primitive `define()` :

```php
define("CONSTANT", "Hello world.");
echo CONSTANT; // outputs "Hello world."

define('ANIMALS', [
    'dog',
    'cat',
    'bird'
]);

echo ANIMALS[1]; // outputs "cat"

//Afficher toutes les constantes définies et accessibles dans ce script
print_r(get_defined_constants());
```

> Par convention, écrire les constantes en `MAJUSCULES`. On utilisera souvent des constantes en WordPress, notamment pour la configuration.

Structures de contrôle : `for`

```php
<?php

//Boucle for
for($i = 0 ; $i < 100; $i = $i + 1){
    //Embranchement avec if/else (tests)
    if($i % 2 === 0){
        echo "$i est un nombre pair";
    }else{
        echo "$i est un nombre impair";
    }
}
```

Structures de contrôle : `for` syntaxe alternative

```php
<?php

//Boucle for
for($i = 0 ; $i < 100; $i = $i + 1) :
    //Embranchement avec if/else (tests)
    if($i % 2 === 0) :
        echo "$i est un nombre pair";
    else :
        echo "$i est un nombre impair";
    endif;
endfor;
```

> Sera utile dans les templates

Structures de contrôle : `while` et `do-while`

```php
<?php

$i = 1;
while ($i <= 10) {
    echo $i++;  /* La valeur affichée est $i avant l'incrémentation
                   (post-incrémentation)  */
}

$i = 0;
do {
    echo $i;
} while ($i > 0);
```

> En savoir plus sur while, sur do-while

Structures de contrôle : `while` et `do-while`, syntaxe alternative

```php
<?php

$i = 1;
while ($i <= 10) :
    echo $i++;
endwhile;
```

Structures de contrôle : `switch`

> Alternative à if/else si les cas sont énumérables et tests d'égalité uniquement

```php
<?php

$i=2;
switch ($i) {
    case 0:
        echo "i égal 0";
        break;
    case 1:
        echo "i égal 1";
        break;
    case 2:
        echo "i égal 2";
        break;
}
```

> En savoir plus sur switch

Syntaxe alternative

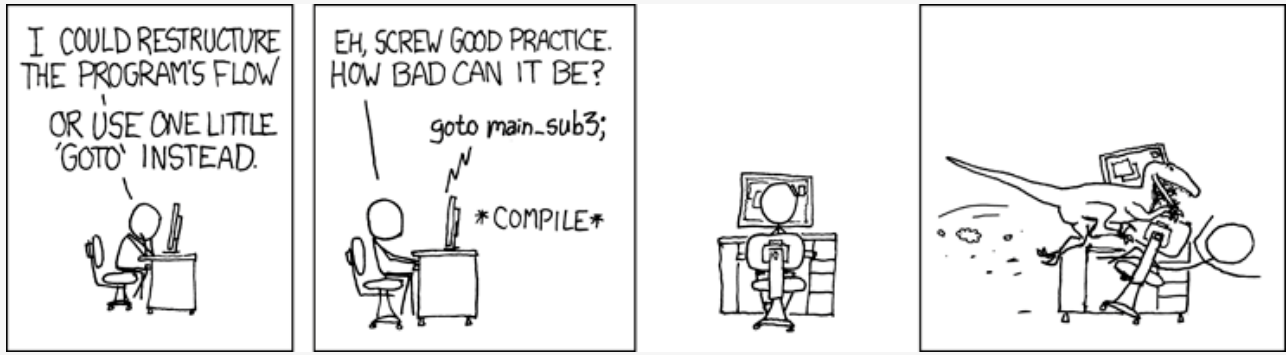
```php
$i=2;
switch ($i) :
    case 0:
        echo "i égal 0";
        break;
    case 1:
        echo "i égal 1";
        break;
    case 2:
        echo "i égal 2";
        break;
endswitch;
```

goto

{.marp-bg-img}

- **Pas de bonne raison de s'en servir dans du code haut niveau** : complexité (`goto` a du sens en bas niveau)
- **Ne pas s'en servir dans une application web PHP !**

```php
<?php
for ($i = 0, $j = 50; $i < 100; $i++) {
    while ($j--) {
        if ($j == 17) {
            goto hell;
        }
    }
}
echo "i = $i";
hell:
echo 'j hit 17';
```



> Le *classique* Edgar Dijkstra: Go To Statement Considered Harmful : *"I've been familiar with the observation that the quality of programmers is a decreasing function of the density of `goto` statements in the programs they produce".*
> xkcd: goto

Structures de contrôle : `foreach`

Syntaxe commode pour parcourir des collections d'items. Permet de se passer ou non de l'index.

```php
<?php
$arr = array(1, 2, 3, 4);

//Parcourir un tableau en parcourant uniquement ses valeurs
foreach ($arr as $value) {
    //$value est une variable locale à la boucle qui contiendra chaque valeur du tableau (1, puis 2, puis 3, puis 4)
    //a chaque itération
    echo $value . PHP_EOL;
}
//Parcourir un tableau en parcourant ses clefs ET ses valeurs
//Syntaxe: foreach(tableau as clef => valeur)
foreach($arr as $index => $value){
    echo "Index: {$index} - Value: {$value} \n";
}
```

> En savoir plus sur foreach

Structures de contrôle : `foreach` syntaxe alternative

Les **accolades** ouvrantes et fermantes de la boucle sont **remplacées par** le caractère `:` et par une instruction `endforeach;` :

```php
<?php
$arr = array(1, 2, 3, 4);

// foreach($arr as $index => $value){
//     echo "Index: {$index} - Value: {$value} \n";
// }

foreach($arr as $index => $value) :
    echo "Index: {$index} - Value: {$value} \n";
endforeach;
```

Syntaxes alternatives dans les templates

Lorsque l'on utilise PHP pour générer des pages HTML, les structures de contrôle usuelles offrent une syntaxe alternative afin d'améliorer (un peu) la lisibilité du code source de la page.

Prenons un exemple :

```php
//Tout le HTML est généré depuis un code PHP
$items = ['pomme', 'poire', 'raisin'];
echo "<ul>";
foreach($items as $item){
    echo "<li>$item</li>";
}
echo "</ul>";
```

Structures de contrôle, syntaxes alternatives dans les templates

Dans un template, le script PHP est essentiellement **constitué de texte** (code HTML), le contenu est donc imprimé sur la sortie sans modification. Les balises PHP ne sont ouvertes que pour insérer des données dynamiques :

```
//Le HTML est écrit directement, PHP est seulement utilisé pour les données dynamiques
<?php $items = ['pomme', 'poire', 'raisin']; ?>
<ul>
<?php foreach ($items as $item) { ?>
        <li><?php echo $item ?></li>
<?php } ?>
</ul>
```

> Syntaxe encore difficile à lire...

Syntaxes alternatives dans les templates (foreach)

Idem que précédemment mais avec la **syntaxe alternative** pour le `foreach` :

```
//Le HTML est écrit directement, PHP est seulement utilisé pour les données dynamiques
<?php $items = ['pomme', 'poire', 'raisin']; ?>
<ul>
<?php foreach ($items as $item) :?>
        <li><?php echo $item ?></li>
<?php endforeach; ?>
</ul>
```

Cette syntaxe alternative est plus lisible, notamment lorsque les pages à générer deviennent plus complexes.

> En savoir plus sur la syntaxe alternative

Syntaxes alternatives dans les templates (if/else)

La structure de contrôle `if/else` (embranchement) dispose également d'une syntaxe alternative.

```
<?php $items = ['pomme', 'poire', 'raisin']; ?>
//...
//Syntaxe alternative pour le if
<?php if(!empty($items)) : ?>
    <!-- Cette balise sera écrite dans la page web si la liste d'items n'est pas vide -->
    <p> Il y a des items dans la liste ! </p>
<?php else : ?>
    <!-- Cette balise sera écrite dans la page web sinon -->
    <p> La liste est vide ! </p>
<?php endif; ?>
```

Exemple concret d'usage de la syntaxe alternative dans les templates d'un thème Wordpress

Exemple d'usage de la syntaxe alternative pour la Loop Wordpress, un pattern au coeur du fonctionnement du CMS :

```
//Script PHP en charge d'afficher la liste des articles, d'un article, etc.
//S'il y a des posts (articles), parcourir les posts (while)
//Affiche le contenu du post (the_content()) et passer au post suivant (the_post())
 <?php if ( have_posts() ) : while ( have_posts() ) : the_post(); ?>
    <div class="entry">
        <?php the_content(); ?>
    </div>
 <?php endwhile;?>
```

Les chaînes de caractères

En interne, une chaîne PHP est stockée dans une structure (`zval`) qui contient :

- Un *pointeur* vers un *buffer* mémoire ;
- La taille du buffer (nombre d'octets/*bytes*) ;

Pour connaître la taille d'une chaîne, il faut utiliser la fonction `strlen()`.

Attention, `strlen` ne retourne pas le nombre de caractères mais le nombre de *bytes* !

```php
echo strlen('foo') . PHP_EOL; // 3
echo strlen('été') . PHP_EOL; // 5
```

Quoting : simple vs double

- **Simple quotes** :
  - n'échappent pas les caractères spéciaux, échappement manuel avec `\'`, `\\` :
  - **pas d'interpolation des variables** ;
- **Double quotes** (opérateur) :
  - Échappent les caractères spéciaux `,` `` ` `` ;
  - Interprètent les séquences `\n`, `\t`, `\"`, etc.
  - **Interpolation** des variables (`"$var"`, `"{$var}s"`).

Les chaînes de caractères : simple quotes

```php
<?php
echo 'ceci est une chaîne simple';
//Concaténation de chaîne avec l'opérateur .
//PHP_EOL est une constante fournie pour demander le caractère fin de ligne
echo 'ceci est une chaîne simple' . PHP_EOL;

echo 'Vous pouvez également ajouter des nouvelles lignes
dans vos chaînes
de cette façon';

// Échapper des caractères spéciaux avec l'anti-slash (ici la guillemet simple avec \') pour signifier qu'il fait partie de la chaîne
echo 'Arnold a dit : "I\'ll be back"';
echo 'Voulez-vous supprimer C:\*.*?';
echo 'Ceci n\'affichera pas \n de nouvelle ligne';
```

Les chaînes de caractères : double quotes

```php
$firstName = "Jane";
$lastName = "Doe";

//Les simples quotes ne permettent aucune interprétation de caractères spéciaux dans la chaîne
echo 'La variable $firstName ne sera pas interpolée ici (remplacée par sa valeur)\n';

//Les doubles quotes effectuent des interprétations de caractères spéciaux (interpolation, caractère de retour à la ligne, etc.)
echo "La variable $firstName sera interpolée ici (remplacée par sa valeur)\n";
echo "Les variables {$firstName}/{$lastName} seront interpolées ici (remplacée par sa valeur)\n";
```

Syntaxe Heredoc

Une autre façon de délimiter une chaîne de caractères est la syntaxe Heredoc : `<<<`.

Après cet opérateur, un identifiant est fourni (ici `END`), suivi d'une nouvelle ligne. La chaîne de caractères en elle-même vient ensuite, suivie du même identifiant pour fermer la notation.

```php
<?php
// no indentation
echo <<<END
$$$$$$$\  $$\   $$\ $$$$$$$\
$$  __$$\ $$ |  $$ |$$  __$$\
$$ |  $$ |$$ |  $$ |$$ |  $$ |
$$$$$$$  |$$$$$$$$ |$$$$$$$  |
$$  ____/ $$  __$$ |$$  ____/
$$ |      $$ |  $$ |$$ |
$$ |      $$ |  $$ |$$ |
\__|      \__|  \__|\__|
END;
```

> Peut être utile dans des templates !

Syntaxe Heredoc : stocker le résultat dans une variable

```
$text= <<<FOOBAR
À d’autres la satiété ! Toi, tu gardes ton désir, désir exubérant qui déborde toujours : moi qui te poursuis sans cesse, je viens par-dessus le marché faire addition à tes tendres caprices.

Toi, dont le désir est si large et si spacieux, ne daigneras-tu pas une fois absorber mon désir dans le tien ? Ton désir sera-t-il toujours si gracieux aux autres sans jeter sur mon désir un rayon de consentement ?

La mer, qui est toute eau, reçoit pourtant la pluie encore, et ajoute abondamment à ses réservoirs : ainsi toi, riche de désir, ajoute à tes désirs la goutte du mien, et élargis ton caprice.

Ne te laisse pas accabler par tant de tentations, bonnes ou mauvaises : confonds-les toutes en une, et aime Will dans ce désir unique (1).
FOOBAR;
echo $text;
```

Quelques fonctions utiles pour manipuler les chaînes de caractères

- implode;
- sprintf;
- fprintf;
- number_format;
- str_contains
- strtolower;
- strtoupper;
- ucwords;
- strpos;
- htmlentities.

> Voir la liste des fonctions natives de PHP pour manipuler les chaines

Exemple à copier/coller et parcourir

```php
<?php
$x = 'bOnJoUr eT bIeNvEnUe';
echo "<p><code>strtolower('$x')</code> = " . strtolower($x) . '<br>';
echo "<code>strtoupper('$x')</code> = " . strtoupper($x) . '<br>';
echo "<code>ucwords('$x')</code> = " . ucwords($x) . '<br>';
echo "<code>ucwords(strtolower('$x'))</code> = " . ucwords(strtolower($x)) . ' </p>';
echo '<h5>Mise en forme avec <a href="http://php.net/manual/fr/function.sprintf.php"><code>sprintf()</code></a></h5>';
echo '<p>Mise en forme d\'une date : ' . sprintf('%02d/%02d/%04d', 1, 1, 1981) . '</p>';
echo '<h5>Mise en forme avec <a href="http://php.net/manual/fr/function.number-format.php"><code>number_format()</code></a></h5>';
$x = 1234.567;
echo "<p><code>number_format($x)</code> = " . number_format($x) . '<br>';
echo "<code>number_format($x,1)</code> = " . number_format($x, 1) . '<br>';
echo "<code>number_format($x,2,',',' ')</code> = " . number_format($x, 2, ',', ' ') . '</p>';
$mail = "paul.schuhmacher@monmail.fr";
$position = strpos($mail, 'h');
if ($position === false) {
    echo "'h' est introuvable dans $mail<br>";
} else {
    echo "'h' est à la position $position dans $mail<br>";
}
echo '<p><code>mktime(0, 0, 0, 1, 1, 1970)</code> = ' . mktime(0, 0, 0, 1, 1, 1970) . '</p>';
echo '<p>Création du timestamp pour le 4 octobre 2017 à 11h45 et 30 secondes<br>';
echo '<code>mktime(11, 45, 30, 4, 10, 2017)</code> = ' . mktime(11, 45, 30, 4, 10, 2017) . '</p>';
echo '<p><code>date("d/m/Y H:i:s", mktime(11, 45, 30, 4, 10, 2017))</code> = ' . date("d/m/Y H:i:s", mktime(11, 45, 30, 4, 10, 2017)) . '<br>';
echo 'Unix a fêté sa milliardième seconde le ' . date("d/m/Y H:i:s", 1000000000) . '</p>';
echo '<p>Date du jour au format JJ/MM/AAAA : ' . date('d/m/Y') . '<p>';
```

Cas des chaînes multi-bytes et encodage

En PHP, une chaîne est une **séquence de bytes**, pas une séquence de caractères. Avec l'encodage UTF-8, certains caractères occupent plusieurs octets.

- `strlen()` : compte les **bytes**;
- `mb_strlen()` : compte les **caractères** (si l'encodage est bien défini).

```php
echo strlen("été") . PHP_EOL; // 5
echo mb_strlen("été") . PHP_EOL; //3
echo mb_internal_encoding(); // UTF-8, par défaut
```

> Encode par défaut UTF-8. Les chaines sont juste des bytes, il faut donc définir/vérifier l'encodage pour les interpréter correctement ! En savoir plus

Utiliser la documentation officielle

La documentation officielle de PHP est de qualité.

Par exemple, regardons la doc de la fonction mb_strlen :

```
//Signature
mb_strlen(string $string, ?string $encoding = null): int
```

> **Ignorer** la section `User Contributed Notes` ! Désuète, dépréciée, inutile. Va être supprimée dans la mise à jour prochaine du site web.

Dans VS Code, installer l'extension PHPIntelephense pour avoir accès à la doc directement depuis l'IDE !

Les tableaux PHP

Les tableaux PHP sont très (trop !) **puissants** et **versatiles** :

- "Tableaux" (collections de valeurs indexées). **Les éléments peuvent être de n'importe quelle type** (y compris des fonctions!);
- **Tableaux associatifs** (dictionnaire, *map*).
- Par défaut, les tableaux commencent à l'indice `0` par défaut et utilise des entiers comme index ;
- Pour connaître la taille d'un tableau, utiliser la fonction `count()`.

Les tableaux PHP, exemple

```php
<?php
$array = [1, 2, 3, ['a','b','c']];
//Retour à la ligne + nouvelle ligne (pour le terminal)
echo "Taille du tableau : " . count($array) . PHP_EOL;
//Ajouter un élément
$array[] = 4;
//Ajoute des éléments à la fin
array_push($array, 5, 6, 7);
//Ajoute des éléments au début
array_unshift($array, -1, 0);
print_r($array);
//Supprimer une clef et sa valeur !
unset($array[0]);
print_r($array);
```

> En savoir plus

Les tableaux associatifs en PHP : parcourir, ajouter, supprimer

En PHP, un tableau **est en réalité une *map***, collection de clé/valeurs, la fonction la plus fondamentale !

```php
<?php

//Map, la fonction la plus fondamentale (application)
$square = [
    0 => 0,
    1 => 1,
    2 => 4,
    3 => 9
];

//Tableau : on déclare les clefs et les valeurs (slug => title)
$posts = [
    'hello-world' => "Hello World !",
    'how-to-use-array' => "How to" ,
];
foreach($posts as $slug => $title){
    //Equivalent "string literals" avec les back tick en JS
    echo "Slug: {$slug} - Title: {$title} \n";
}

//Ajouter un élément
$posts['add-element'] = "Ajouter un élément à un tableau PHP";
//Supprimer un élément
unset($posts['hello-world']);
```

(Quelques) Fonctions utiles sur les tableaux

`in_array`, **`implode`**, **`explode`**, **`array_map`**, **`array_filter`**, `array_reduce`, `shuffle`, `sort`, array_find, array_first, array_last, array_column, array_diff, etc.

```php
$array = [1,2,3];
//implode: Construit une chaine à partir des éléments d'un tableau, avec un séparateur
$array = ['lastname', 'email', 'phone'];
//in_array : permet de tester si un élément est dans le tableau
if(in_array('email', $array)){
    echo "email est dans le tableau";
}
//shuffle : mélange les éléments du tableau de manière aléatoire
shuffle($array);
var_dump(implode(",", $array)); // string(20) "lastname,email,phone"
```

> Voir toutes les fonctions natives de PHP pour manipuler le tableaux

NULL

`NULL` est un type spécial et une valeur qui indique **l'absence de valeur**. La fonction isset() est parfaitement adaptée pour tester la présence d'une valeur ou d'une clef dans un tableau.

```php
$var = NULL;
//Test avec la fonction isset :
if(!isset($var)){
    echo "la variable ne contient aucune valeur !";
}
```

> En savoir plus

Déclarer et utiliser ses fonctions en PHP

Une fonction, *avant* d'être appelée (exécutée), **doit être déclarée** avec le mot clef `function`. Une fonction peut prendre des arguments (ou pas) et **retourne toujours quelque chose** (si pas d'instruction `return`, elle retourne `NULL`)

On indique le type des données des arguments et de retour de la fonction (*type hinting*)

```php
function sayHi(string $firstName = 'Jane', string $lastName = 'Doe') : string {
    $msg = "Hello $firstName $lastName !";
    return $msg;
}

//Appel de la fonction
echo sayHi();
echo sayHi('John');
echo sayHi('Rasmus', 'Lerdorf');
```

> En savoir plus sur les fonctions.

Hoisting (remontée en haut de la portée avant exécution)

Les fonctions n'ont pas besoin d'être définies avant d'être utilisées, SAUF lorsqu'une fonction est définie conditionnellement.

```php
//Appel de la fonction avant sa définition ok !
echo sayHi();

function sayHi(string $firstName = 'Jane', string $lastName = 'Doe') : string {
    $msg = "Hello $firstName $lastName !";
    return $msg;
}

foo(); //Erreur ! Le bloc ci dessous DOIT être exécuté pour que foo soit définie !

if ($makefoo) {
  function foo()
  {
    echo "Je n'existe pas tant que le programme n'est pas passé ici.\n";
  }
}
```

Portée des variables

- La portée d'une variable **est le contexte dans lequel elle est définie**
- PHP possède une portée :
  - **de fonction** (locale);
  - une portée **globale** (script et scripts inclus);

Portée locale vs globale

```php
<?php
$a = 1; // portée globale

function test()
{
    echo $a; // La variable $a est indéfinie car elle fait référence à une version locale de $a
}
test(); //Warning Undefined variable $a !
```

Portée globale avec `global`

Le mot-clé `global` est utilisé pour lier une variable de la portée globale dans une portée locale.

```php
<?php
$a = 1; // portée globale

function test()
{
    global $a; // lier $a local à $a global
    echo $a; // Ok
}
test(); //1
```

> Cette instruction sera utile car WordPress utilise intensivement plusieurs variables globales !

Functions *as first class citizens*

PHP supporte **le paradigme fonctionnel**. En PHP, une fonction est (peut être vue) *une donnée comme les autres*. On peut donc créer des *closures*, faire du *currying*, etc.

Une fonction peut :

- Prendre des fonctions en argument ;
- Retourner des fonctions.

> Pour cela, PHP définit le type callable (implémenté sous forme de classe native)

Exemple de closure

```php
<?php

$input = array(1, 2, 3, 4, 5, 6);

// Crée une nouvelle fonction anonyme et l’assigne à une variable
$filter_even = function($item) {
    return ($item % 2) == 0;
};

$output = array_filter($input, $filter_even);

// Inutile de l'assigner à une variable, on peut utiliser directement une fonction anonyme :
$output = array_filter($input, function($item) {
    return ($item % 2) == 0;
});
```

Exemple de closure

```php
<?php

/* Crée une fonction de filtre anonyme acceptant les éléments > $min
 * Retourne un filtre unique parmi une famille de filtres "supérieur à n"
 */
function greater_than($min)
{
    //Closure : accède à des variables fixées par son environnement ($min)
    //avec l'instruction 'use' pour capturer la valeur de $min à la déclaration
    return function($item) use ($min) {
        return $item > $min;
    };
}

$input = array(1, 2, 3, 4, 5, 6);

$output = array_filter($input, greater_than(3));
```

Fonctions dans les templates, alternance PHP / HTML

```php
<?php
function print_html(){
?>
<nav>
    <ul>
        <li><a href="/foo">foo</a></li>
        <li><a href="/bar">bar</a></li>
        <li><a href="/baz">baz</a></li>
    </ul>
</nav>
<?php
}

print_html();
```

> Syntaxe très propre au fonctionnement de PHP. Utile pour écrire des gros blocs de HTML sans échapper. Privilégier néanmoins l'include de templates (plus lisible).

Passage par valeur

En PHP, comme en C, **tout argument est passé à une fonction par valeur** (copie locale).

Passage par valeur

```php
<?php
function addToArray(array $arr){
  $arr[] = 'bar';
}

$array = ['foo'];

addToArray($array);

var_dump($array);  //?
```

Passage par valeur

```php
class Foo{
  public function __construct(public int $counter = 0){}
}

function incFoo(Foo $foo): void{
  $foo->counter++;
}

$foo = new Foo();

incFoo($foo);

echo $foo->counter . PHP_EOL; // ?
```

Passage par référence

On peut passer directement la référence d'une variable (chaîne, tableau, etc.) en utilisant l'opérateur `&`.

```php
//Passage par référence (explicite)
function addToArray(array &$arr){
  $arr[] = "bar";
}

$array = ['foo'];

addToArray($array);

var_dump($array);  //?
```

Modification *sur place*.

PHP et son environnement : les variables *Super Globales*

Les variables Super Globales sont des variables **fournies** aux scripts PHP *à l'exécution*, **contenant toutes les informations sur : l'environnement du script**, **la requête HTTP**, le client, le serveur, etc.

Ce sont des tableaux associatifs PHP, qui commencent, par convention, par un underscore (`_`).

```
#Un tableau associatif des valeurs passées au script courant via le protocole HTTP et la méthode POST
$_POST;
#Un tableau associatif des valeurs passées au script courant via le protocole HTTP et la méthode GET
$_GET;
#$_SERVER est un tableau contenant des informations telles que les en-têtes, les chemins et les emplacements de script. Crée par le serveur web (programme)
$_SERVER;
#Un tableau associatif des valeurs téléchargées au script courant via le protocole HTTP et la méthode POST
$_FILES;
#Un tableau associatif de variables, passé au script courant, via des cookies HTTP.
$_COOKIE;
#Un tableau associatif qui contient par défaut le contenu des variables $_GET, $_POST et $_COOKIE.
$_REQUEST;
```

> **Ne modifiez jamais vous-même ces tableaux**, accédez-y en lecture seulement. Dans WordPress, on n'aura rarement (jamais ?) besoin d'y accéder directement. Le *core* nous en fournira une abstraction.

Répartir le code PHP dans différents fichiers et l'importer au besoin

Pour utiliser le code PHP placé dans un autre fichier, on peut utiliser le construct `require_once` :

- Le contenu du fichier *required* est importé dans le script courant, une seule fois (pas de redondance en cas d'inclusion multiple du même fichier) ;
- Si le fichier n'existe pas, `require_once` lève une erreur fatale qui met fin a à l'execution du script.

```php
<?php
//Inclure tout le contenu (UNE FOIS) du fichier foo.php dans le script courant
require_once './foo.php';
//Inclure tout le contenu (UNE FOIS) du fichier foo.php dans le script courant
require_once './bar.php';
//N'aura aucun effet
require_once './bar.php';
```

> Il existe d'autres construct pour inclure des scripts : include, include_once, require, etc.. **Privilégier `require` et `require_once`**.

Précisions sur le mécanisme d'import de script PHP

Le `require_once` **n'agit pas comme un simple copier/coller textuel** brut du contenu du script importé (comme `#include` en C), **mais comme une évaluation du script**. Conséquences :

- Les variables définies avant le `require` sont visibles dans le fichier inclus ;
- Les variables ou fonctions définies dans le fichier inclus restent visibles après ;
- Si le fichier contient des instructions PHP, elles sont exécutées immédiatement ;
- Tout ce qui n'est pas du code PHP (hors balises) **est recopié directement sur la sortie**.

Précisions sur le mécanisme d'import de script PHP

Fichier `component.php` :

```
<?php $last_component = "p"; ?>
<p>Un composant</p>
```

Fichier `index.php` :

```
//require_once ouvre, compile et execute le script component.php comme si on avait fait 'php component.php'
require_once 'component.php';
echo $last_component;
```

Exécuter le script `index.php` produit la sortie suivante :

```
<p>Un composant</p>
p
```

Type hinting

PHP est un langage **typé dynamiquement**, ce qui apporte de la *flexibilité* mais peut conduire à *des comportements inattendus* ! (bugs).

Depuis **PHP 7**, il est possible (**et recommandé**) de déclarer les types des arguments et retours de fonction.

Type hinting en pratique

Cela **assure à l'execution** que la valeur passée en argument à une fonction est du type attendu, sinon une erreur fatale `TypeError` est lancée (si coercition désactivée) et interrompt l'exécution du programme.

```php
<?php
//Déclaration de fonction utilisant le *type hinting* : attend une chaîne et un entier, retourne une chaîne
function repeat(string $str, int $times): string {
    if($times == 0)
      return '';
    return $str . repeat($str, $times - 1) ;
}

echo repeat("ha", 13);
```

> **TOUJOURS utiliser le type hinting** !

Conversion automatique

Attention, concernant les valeurs **scalaires** ou **primitives** (`string` , `int` , `float` , etc.), PHP tente par défaut de **convertir** (*coerce*) vers le type attendu (défini par le *type hinting*)

```php
<?php
function sum(int $a, int $b): int {
    return $a + $b;
}

var_dump(sum(1.5,2.5)); // Affiche int(3)
```

Conversion automatique et typage `strict`

Pour empêcher PHP de faire cette conversion (qui peut provoquer des comportement inattendus!), on peut activer le mode *typage strict* **par fichier**, avec l'instruction `declare(strict_types=1);`

> Elle peut aussi éviter des problèmes non anticipés...

Dans ce mode, **seule une variable correspondant exactement au type attendu dans la déclaration sera acceptée** sinon une `TypeError` sera levée, **sauf si la conversion garantit l'intégrité** de la donnée (`int -> float`)

```php
<?php
declare(strict_types=1);

var_dump(sum(1.5, 2.5));
```

Sortie :

```
PHP Fatal error:  Uncaught TypeError: sum(): Argument #1 ($a) must be of type int, float given
```

Typage strict

- Ce mode ne s'applique qu'aux **types scalaires/primitifs**;
- Ce mode ne **s'applique qu'un fichier où il est activé**;
- **Le typage strict s'applique aux appels de fonction effectués depuis l'intérieur d'un fichier dont le typage strict est actif**, et non aux fonctions déclarées dans ce fichier;

En résumé

- **Toujours utiliser le *type hinting*** (documentation, intention, lever des erreurs à l'exécution avant que le système ne casse, etc.);
- **Toujours activer le mode `strict` dans les fichiers contenant de l’exécution de code** que **vous contrôlez**;

Programmation orientée objet en PHP

Les fondamentaux de la programmation orientée objet (POO) en PHP.

Sommaire

Mêmes concepts que dans tous les langages populaires supportant la POO (Java, C#, C++, Python, Kotlin, Dart, etc.)

- Classe et objets (`class`, `new`, `__construct()`, `__destruct()`)
- Visibilité (`public`, `private`, `protected`)
- Opérateur de résolution de portée `::`
- `static`
- Héritage (`extends`)
- Contrôle sur l'héritage avec le mot clef `final`
- Classe et méthode abstraite (`abstract`)
- **Polymorphisme**
- Concept général d'interface
- Quelques principes de *design*

Classes et objets : un programme composé d'objets

- Les programmes conçus en POO sont composés d'*objets* qui échangent des *messages*
- L'exécution du programme = échange de messages entre objets
- Un message est émis à un objet quand on demande à un objet d'exécuter une de ses méthodes.

```
$foo = new Foo();
$bar = new Bar();
//$foo reçoit le message 'doStuff' dont le contenu est donné par $bar
$foo->doStuff($bar->stuffToDo());
```

> La difficulté d'un système designé avec de la programmation orienté objet est de décomposer le système en objets tout en gardant le système compréhensible et ouvert aux changements (flexibilité).

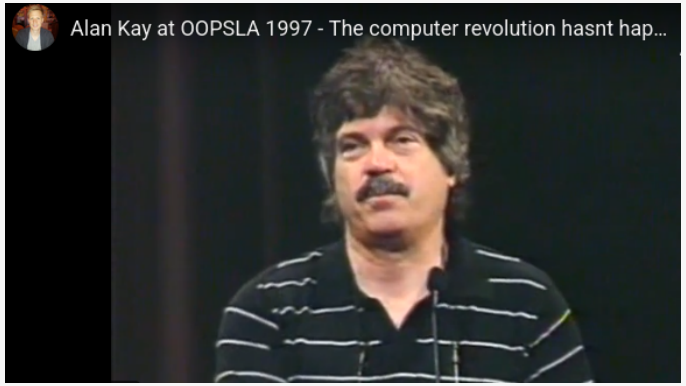
Programmation orientée objet (vision originale d'Alan Kay)

{.marp-bg-img}

Dr. Alan Kay on the Meaning of *Object-Oriented Programming* :

> "OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things"

- *Messaging* : Envoi de messages aux objets selon un protocole. Pattern *Publish-Subscribe*;
- *Isolation* : API. Cacher état interne (données), orienté *objet* = orienté *processus*. Objet = petit ordinateur
- *Extreme late binding* : Polymorphisme.



> Source de l'image : Alan Kay at OOPSLA 1997 - The computer revolution hasnt happened yet

Classes

La **classe** donne une définition (signature) des propriétés et des méthodes ainsi qu'une implémentation (ce que font les méthodes, le *corps* des méthodes). Une classe définit également un espace de nom.

```php
class Foo{
    //Une propriété (une variable de classe)
    string $proprerty;
    function __construct($property){
        //La pseudo variable $this (mise à disposition par PHP) est disponible lorsqu'une méthode
        //est appelée depuis un contexte objet. $this a pour valeur l'objet appelant.
        $this->property = $property;
    }
    //Une méthode (une fonction de classe)
    function doStuff(array $stuffToDo){
        //instructions
    }
}
```

> Une classe est une chose statique. Tous les langages orientés objets ne sont pas basés sur des classes (par ex : langages orientés prototypes comme JS, prototype est dynamique)

Classes et objets: instancier un objet `new` et accéder aux propriétés/méthodes

- Un *objet* est une instance de *classe*;
- Un objet est instancié, crée, avec l'opérateur `new`. Ce opérateur fait appel au constructeur `__construct()` pour initialiser l'objet.

```php
//Création d'un objet
$foo = new Foo('foo');
//Accéder à une propriété (l'encapsulation est brisée,
//les détails internes de l'objet sont exposés à sa surface et accessibles au monde entier)
$foo->property;
//Accéder à une méthode (i.e envoyer un message à l'objet)
$foo->doStuff(array(1,2,3))
```

Dans un langage *réellement POO*, **envoyer des messages** (appeler des méthodes) est la **seule manière** à notre disposition pour faire **agir un objet** (exécuter du code).

Les messages sont (et *devraient!*) être la seule manière de changer l'état interne d'un objet. Ainsi, l'état interne d'un objet est dit *encapsulé*, ses rouages internes (propriétés, méthodes privées, etc.) sont invisibles depuis l'extérieur.

Classes et objets: Signature d'une méthode

Signature d'une méthode :

- Une visibilité (`public` (API), `private`, `protected`) ;
- Comme les fonctions :
  - un type de retour (`void` indique que la méthode ne retourne rien);
  - un ou plusieurs arguments ayant chacun un type (`void` indique que la méthode ne prend rien en argument).

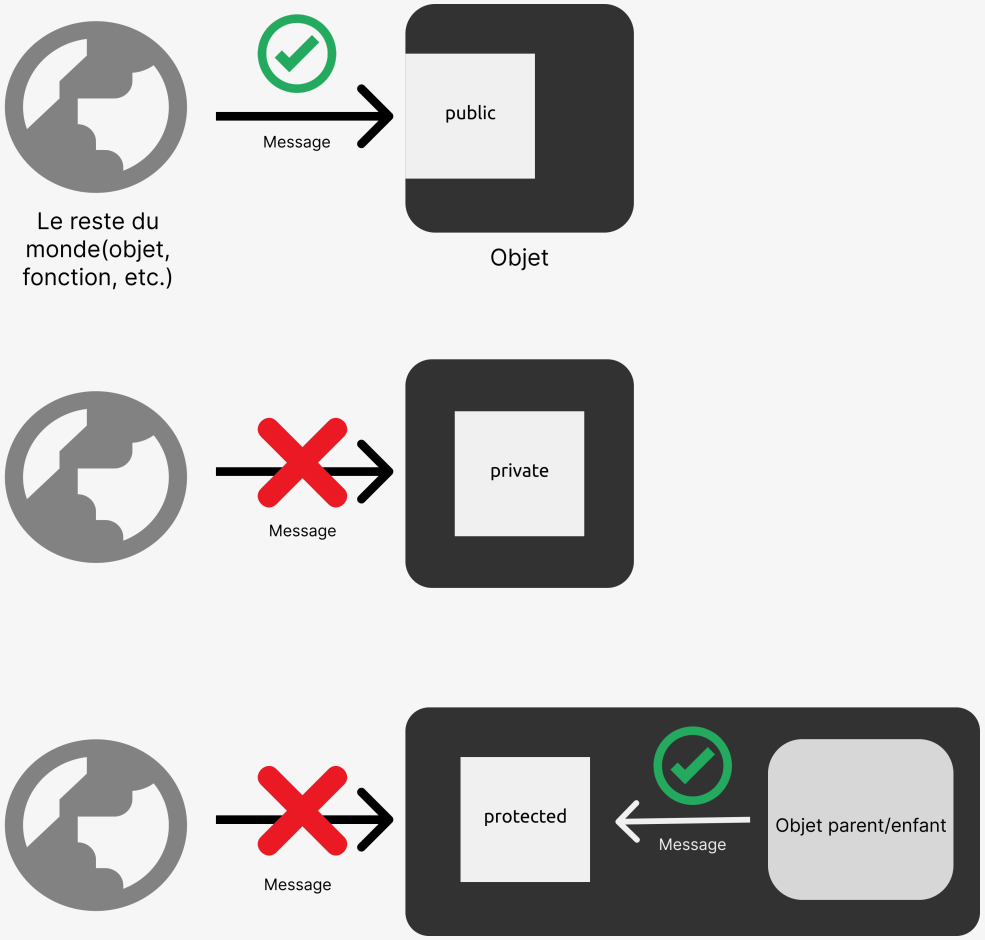
```
//La signature de la méthode `doStuff` de la classe Foo
public doStuff(array $stuffToDo):void
```

Classes et objets : Visibilité

La visibilité définit l'exposition des attributs et des méthodes d'un objet aux autres objets dans le programme. Il existe 3 modificateurs de visibilité :

- `public` : c'est la visibilité par défaut en PHP. Un attribut/une méthode `public` est accessible depuis l'extérieur de la classe, par tous les objets du programme.
- `private` : un attribut/une méthode `private` n'est pas accessible depuis l'extérieur de la classe. Aucun objet dans le programme ne peut y accéder, à part l'objet qui les porte. On ne peut pas redéfinir une propriété/méthode privée dans une classe enfant.
- `protected` : un attribut/une méthode `protected` n'est pas accessible depuis l'extérieur de la classe sauf par les classes parent ou enfant.

Classes et objets : Visibilité



Opérateur de résolution de portée `::`

L'opérateur de résolution de portée `::` est un moyen d'accéder aux membres `static`, `const` (constantes de classe) ainsi qu'aux méthodes surchargées :

- Depuis l'intérieur de la classe: `self::`, `parent::`, `static::`
- Depuis l'extérieur de la classe: `NomDeLaClasse::`

Opérateur de résolution de portée `::`

```php
class Foo
{
    public function __construct()
    {
        echo 'Call Foo constructor' . PHP_EOL;
    }
}

class Bar extends Foo
{
    const BAZ = 'baz';
    public function __construct()
    {
        //Appel au constructeur parent
        parent::__construct();
        echo 'Call Bar constructor' . PHP_EOL;
    }
}
$bar = new Bar();
echo Bar::BAZ . PHP_EOL;
```

Différence entre `self::` et `static::`

- self : résolut *at compile time*
- static : résolut *at run time*

```php
<?php

/**
 * Quelle est la différence entre static et self ?
 * Tous deux font référence à la classe où ces instructions sont utilisées.
 * Mais elles se comportent différemment, notamment dans le cadre de l'héritage.
 */

class Furniture {
    protected static $category = 'General';

    public static function describe() {

        //Changer static par self et observer le résultat

        //static::$category : résolu à l'execution, fait référence à la classe sur laquelle la méthode 'describe' est appelée, et donc sur sa valeur de $category(Chair ici).
        //Changement de comportement possible en héritage (late binding)

        //self::$category : résolu à la compilation, fait référence à la classe sur laquelle le mot clef est utilisé, non à  la classe sur laquelle la méthode est appelée (Furniture ici)
        //Changement de comportement/valeur impossible via l'héritage
        return 'This is a ' . self::$category . ' piece of furniture.' . PHP_EOL;
    }
}

class Chair extends Furniture {
    //Overriding de la valeur statique dans la classe enfant
    protected static $category = 'Chair';
}

echo Furniture::describe();
echo Chair::describe();
```

Attributs et méthodes `static`

Une propriété ou une méthode `static` peut être accédée sans avoir besoin d'instancier la classe (*statique,* car pas besoin pour y accéder d'utiliser l'opérateur `new` et une allocation mémoire, c'est un accès *à froid*)

```php
class Foo
{
    private static string $bar = 'bar';
    public static function bar()
    {
        return static::$bar;
    }
}
//Je peux appeler la méthode bar de la classe Foo sans instancier d'objet Foo
echo Foo::bar();
```

Null-safe operator

Introduit en PHP 8, le Null-safe operator apporte du sucre syntaxique pour tester l'existence d'une propriété/méthode sur un objet :

```
//Retourne NULL si l'une des expressions est NULL
$foo?->bar?->baz;
```

Exemple sans Null-safe operator

```php
class Customer {
 public function __construct(public ?Address $address = NULL){
 }
 public function getAddress(): ?Address {
  return $this->address;}
}
class Address {
    public function getCountry(): string {}
}

$customer = new Customer();
$address = $customer->getAddress();
if ($address) {
    $country = $address->getCountry();
}
else {
    $country = null;
}
```

Exemple avec Null-safe operator

```php
class Customer {
 public function __construct(public ?Address $address = NULL){
 }
 public function getAddress(): ?Address {
  return $this->address;}
}
class Address {
    public function getCountry(): string {}
}

$customer = new Customer();
//Silent NULL, et non une Fatal Error
$country = $customer->getAddress()?->getCountry();
```

Héritage : `extends`

Une classe qui *étend* une autre classe hérite à la fois :

- De ses propriétés,
- Dde ses méthodes;

Une classe enfant peut accéder et redéfinir les attributs/méthodes de sa classe parente (sauf si `private`). L'héritage est un *mécanisme de partage de code*.

```
class Foo{
    private $baz
    public function baz(){}
}
//Bar étend Foo, Bar est une classe enfant de Foo
class Bar extends Foo{
    //Bar dispose de la propriété $baz et de la méthode baz()
}
```

L'héritage est vu souvent comme une relation "*is-a*" entre deux classes, mais c'est surtout une relation "*behaves like*". Car la classe enfant hérite de l'**implémentation** (comportement) de sa classe parent.

> L'héritage crée un *couplage fort* entre deux classes, tout en exposant les détails internes de la classe parent à ses classes enfant. À utiliser avec précaution !

Héritage : Contrôle sur l'héritage avec le mot clef `final`

On peut contrôler l'héritage avec le mot clef `final`. Une classe `final` est une classe qui ne peut pas être étendue.

```php
final class Foo{}
//Erreur, Foo ne peut pas être étendue
class Bar extends Foo{}
```

Une méthode `final` est une méthode qui ne peut pas être redéfinie dans une classe enfant.

```php
class Foo(){
    public final function foo(){ echo 'Foo call foo';}
    public function bar(){ echo 'Foo call bar';}
}
class Baz extends Foo{
    //Erreur, impossible
    public function foo(){ echo 'Baz call foo';}
    //Valide
    public function bar(){ echo 'Baz call bar';}
}
```

Classes et méthodes abstraites: `abstract`

- Une classe abstraite ne peut pas être instanciée (car son implémentation est incomplète). Toute classe définissant une méthode abstraite est abstraite par définition.
- Les méthodes abstraites définissent simplement la signature de la méthode, non leur implémentation. Cette signature doit être respectée par l'implémentation.

```php
abstract class Foo{
    protected int $foo;
    abstract public function foo():int;
    public function bar(){echo 'bar' . PHP_EOL;}
}
class Bar extends Foo{
    //Bar définit l'implémentation de la méthode abstraite foo, en respectant sa signature
    public function foo():int {
        echo 'Implémentation de foo dans la classe Bar' . PHP_EOL;
        return 10;
    }
}
$bar = new Bar();
echo $bar->foo() . PHP_EOL;
$bar->bar();
```

> À la différence des `interface`, une classe abstraite peut être partiellement implémentée et disposer de propriétés comme n'importe quelle classe.

Les interfaces: `interface` et `implements`

- Les interfaces permet de définir des méthodes (*signatures*) qu'une classe *doit* implémenter (`implements`) sans avoir à définir *comment* ces méthodes fonctionnent.
- Une `interface` permet d'utiliser des objets de façon *interchangeable*.

```php
interface Foo{
    public function foo(string $value): string;
}
interface Bar{
    public function bar(string $value): string;
}
//Une classe peut implémenter plusieurs interfaces
class Baz implements Foo, Bar{
    public function foo(string $value): string{
    //instructions..
    }
 public function bar(string $value): string{
    //instructions..
    }
}
```

> Une `interface` est un cas particulier de classe abstraite *sans propriétés ni implémentations*.

Le polymorphisme

Le *polymorphisme* est la capacité, pour un même message, de correspondre à plusieurs formes de traitements selon l'objet auquel ce message est adressé.

```php
interface SomeBehavior{
    function doStuff();
}
class Foo implements SomeBehavior{
    function doStuff(){echo 'Foo do stuff this way';}
}
class Bar implements SomeBehavior{
    function doStuff(){echo 'Bar do stuff in another way';}
}
// Le message 'doStuff' est envoyé à $foo,
// il affiche 'Foo do stuff this way' (traitement de Foo)
$foo = (new Foo())->doStuff();

// Le MÊME message 'doStuff' est envoyé à bar,
// il affiche 'Bar do stuff in another way' (traitement de Bar)
$bar = (new Bar())->doStuff();
```

> Depuis PHP 8.4, on peut écrire `new Bar()->doStuff()`, sans les parenthèses (sucre syntaxique)

Interface et type

L'interface d'un objet (on ne parle pas ici du construct PHP `interface`) désigne l'ensemble des signatures de méthode d'un objet. N'importe quel message qui correspond à une signature de l'interface peut être envoyé à cet objet. **L'interface d'un objet est synonyme** de **type** de l'objet.

Dans un système OO *où l'encapsulation est bien respectée*, l'interface est *fondamentale,* car les objets sont connus (comment les utiliser) seulement *via* leurs interfaces (ce qu'ils exposent à leur surface).

En particulier :

- Un objet peut avoir plusieurs types (implémenter plusieurs interfaces) ;
- Un type est *sous-type* d'un autre si son interface contient l'interface du *super-type* ;
- Deux objets partageant la même interface peuvent agir de manière complètement différente (avoir des implémentations différentes).

Interface et type

```php
interface A{
    function a();
}
interface B{
    function b();
}
// Foo est de 'type' A, car il possède l'interface de A
class Foo implements A{function a(){}}
// Bar est de type A ET de type B, car il possède les deux interfaces.
// Je peux :
// - lui envoyer le message `b` (i.e appeler sa méthode b)
// - et le message `a`.
// On peut aussi dire que Bar est un sous-type de A (super-type),
// car son interface contient l'interface de A.
class Bar implements A, B{function a(){} function b(){} function bar(){}}
// Baz déclare une propriété de type A,
// une implémentation de A lui est fournie par injection de dépendance.
class Baz {private A $a; function construct(A $a){$this->a = $a;} function a(){$this->a->a();}}
```

> Un diagramme de classes UML permet de bien visualiser les **interfaces** des objets

Quelques principes de design

Objectifs

L'*architecture logicielle* est l'ensemble des techniques et concepts pour rendre le système :

- *Simple à comprendre* : je peux raisonner sur son fonctionnement sans avoir à penser à trop de choses à la fois;
- *Simple à modifier* : je peux changer, ajouter, retirer une partie de mon système sans que cela n'ait trop d'impact dessus.

> Fabriquer un système *simple* est une chose... *difficile* !

> Visioner cette excellente conférence "Simple Made Easy" (2011) de Rich Hickey (créateur de Clojure, Datomic), sur la différence entre *simplicité* et *facilité* (et la complexité logicielle et ses origines)

Quelques principes de design POO

Voyons quelques principes de design pour la POO.

Isoler ce qui change de ce qui ne change pas (trop)

Si l'un des aspects de votre code est susceptible de changer, vous savez que vous êtes face à un comportement qu'il faut isoler du reste de votre système. Vous pourrez ainsi les modifier plus tard sans affecter ce qui ne varie pas.

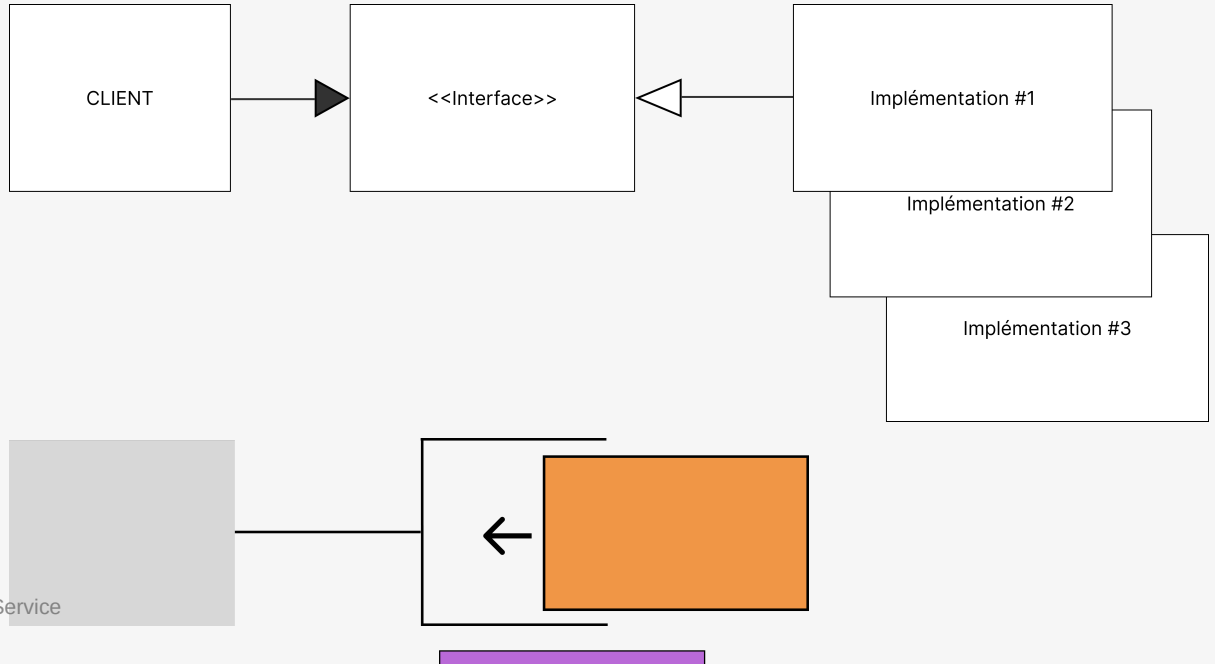
Programmez vers une interface, pas une implémentation (late binding)

Si vous **programmez vers une interface** (c'est-à-dire que votre code client consomme des interfaces et non des implémentations) vous pouvez envoyer le même message aux objets que le client manipule (appeler les mêmes méthodes) tout en obtenant un comportement différent.

Vous pouvez ainsi simplement *changer d'implémentation* en fonction de vos besoins (*plug-in* architecture)

Ce principe dit donc *utilisez le polymorphisme*.

> Le polymorphisme (d'avoir des primitives pour le mettre en place *facilement*) est **le véritable apport** de la Programmation Orientée Objet.



Préférez la composition à l'héritage (quand c'est possible)

Passez de la relation *est-un* et surtout *se comporte comme* (héritage) à la relation *a-un* (composition) entre deux classes.

- Beaucoup plus flexible, vous pouvez modifier les comportements au *run-time* (à l'éxecution), alors qu'avec l'héritage vous devez le faire *at compile-time* (avant exécution), d'où cette idée de couplage fort entre les deux classes. Un changement dans la classe parent amène souvent des changements dans la classe enfant.
- L'héritage vous fait hériter de l'interface, *mais aussi de son implémentation !* On parle alors de *White box visibilité* (détails internes partagés, encapsulation brisée)
- La **composition** (un objet garde une référence vers d'autres objets dans une propriété) offre une *Black box visibility* (aucun détail interne partagé). Un objet composé d'un autre objet n'a besoin de connaître que son interface pour s'en servir (encapsulation préservée).

POO et WordPress

- WordPress *core* a démarré sans POO et l'a intégré au fur et à mesure. Il est donc important de **savoir interroger et instancier des objets** (types `WPdb`, `WP_Query`, `WP_Post`, etc.);
- Ne pas en faire *pour en faire*, si mal utilisée la POO n'est pas la panacée : abstractions sur des *adresses* (**complexité**), explosion de code, de types et d'abstractions peu utiles;
- Utiliser la POO :
  - **Pour remplacer les tableaux associatifs. Utiliser des classes en `readonly`** (*immutable*) à la place pour vos modèles (**map documentées** ou *Value Objects*) + **fonctions** pour les manipuler;
  - Dans vos plugins si vous le souhaitez (POO orienté) ou souhaitez utiliser des *patterns*;
  - Pour développer des composants avec *autoloader* (très axé sur les classes namespacées) !
- *WordPress way* : garder des fonctions préfixées avec votre *vendor name* pour interagir avec le core (*hooks*), templates, etc.

PHP 8.4 (ou 8.5)

> Dernière version actuelle : 8.5 (12/25 !)

PHP 8.4, sortie en fin d'année 2024, est une mise à jour *majeure* du langage PHP.

> PHP 7 (2015) (Perfs, Exceptions, Type Hinting, Classe anonymes) et PHP 8 (2020) (arguments nommés, attributs, promotion constructeur, unions de types, NullSafe operator, throw, **JIT**) avaient également été des nouvelles versions majeures.

Quelques nouvelles features en vrac

Promotion du constructeur

```
class Point {
  public function __construct(
    public float $x = 0.0,
    public float $y = 0.0,
    public float $z = 0.0,
  ) {}
}
```

Classes readonly

Pour faire des object values immutables (approche fonctionnelle)

```php
readonly class Test {
    public string $prop;
}
```

> Très bon ! Pour faire des *Value Objects* (clé/valeur) immutables

Match

```
echo match (8.0) {
  '8.0' => "Oh no!",
  8.0 => "This is what I expected",
};
//> This is what I expected
```

Union

Type hinting permet de faire l'union de différents types de retour

```php
class Number {
  public function __construct(
    private int|float $number
  ) {}
}

new Number('NaN'); // TypeError
```

Enums et backed enums

> Il fallu attendre PHP 8 pour avoir des Enums propres en PHP...

```php
<?php
enum Suit: string
{
    case Hearts = 'H';
    case Diamonds = 'D';
    case Clubs = 'C';
    case Spades = 'S';
}
```

> En savoir plus sur les énumérations.

Nouveautés de PHP 8.4

- Nouvelles fonctions primitives pour manipuler les tableaux :
  - array_find()
  - array_find_key()
  - array_any()
  - array_all() sont désormais disponibles.
- Lazy Objects;
- Nouvelle méthodes pour les classes `DateTime` et `DateTimeImmutable` (API simplifiée et homogénéisée);
- *Sucre Syntaxique* : `new MyClass()->method()` sans parenthèses.
- Fonctions et API obsolètes;
- Etc.;

> Voir toutes les nouveautés de PHP 8.4

Nouveautés de PHP 8.5

> PHP 8.5 sort en novembre 2025 !

Voir toutes les nouveautés à venir de PHP 8.5

Quelques exemples.

Pipe operator `|>`

```php
$resultat = "Hello World"
    |> htmlentities(...)
    |> str_split(...)
    |> fn($x) => array_map(strtoupper(...), $x)
    |> fn($x) => array_filter($x, fn($v) => $v !== 'O');
```

Primitives pour les tableaux : array_first et array_last

```php
$panier = [
'SKU-902' => 'T-shirt PHP',
'SKU-123' => 'Mug Éléphant'
];

$premierProduit = array_first($panier); // "T-shirt PHP"
$premierProduit = array_last($panier); // "Mug Éléphant"
```

Autres nouveautés

- Nouvelle commande de debug : `php --ini=diff`. Affiche directement les valeurs qui diffèrent des valeurs par défaut;
- Stack trace complète en cas de *Fatal Error*;
- Nouvelle extension URI qui implémente, conforme aux standards (spécification URL WHATWG, RFC 1738 et RFC 3986). Fini `parse_url`

PHP Standards Recommandations (PSR)

Framework Interoperability Group (PHP-FIG), un groupe de travail réunissant la plupart des *vendor* PHP pour travailler sur la normalisation des pratiques de développement. Le but est d'assurer l'interopérabilité entre bibliothèques et frameworks PHP pour faciliter la maintenance, la lecture et la réutilisation du code.

- **PSR** : Ensemble de standards établis par le PHP-FIG;
- PSR-1/PSR-12 : Normes de style de code (nommage, fichiers, classes). **Non suivis par la communauté WordPress !**

> Les PSR-1 et PSR-12 sont des standards utiles, à ne pas suivre religieusement... La gouvernance PHP-FIG est néanmoins contestée...

PHP Standards Recommandations (PSR)

Extraits recommandés des PSR-1 et PSR-12 pour le développement avec WordPress :

- The closing `?>` tag MUST be **omitted** from **files** containing **only PHP**;
- (Class) **constants** MUST be **declared in all upper case with underscore separators**;
- **Files SHOULD** either **declare symbols** (classes, functions, constants, etc.) **or cause side-effects** (e.g. generate output, change .ini settings, etc.) but **SHOULD NOT do both**;
- Method and function **arguments with default values** MUST go **at the end of the argument list**.

PHP Moderne : auto-loading et composants

Pratique : Mini projet sans auto-loading

- **Pratiquer** les fondamentaux de PHP vus précédemment ;
- **Discuter** de quelques pratiques ;
- Préparer un petit système que l'on va :
  - **Migrer** vers l'*Autoloading PSR-4*;
  - Faire **utiliser des composants**;

Pratique : Mini projet sans auto-loading

Créer un site web qui **affiche une liste de publications sur sa page d'accueil**.

1. **Préparer** le projet avec la structure suivante :

```
demo_autoloading/
├── index.php
├── config.php
├── db.php (simulera une petite base de données)
└── src/
    ├── Model/
        └── User.php
        └── Post.php
```

Pratique : Mini projet sans auto-loading

2. **Créer** les modèles suivants :

    i. `User.php` définit une classe `User`. Un `User` est défini par :

        a. un **identifiant unique** (`int`),
        b. un **prénom** et un **nom**.

    Un `User` **peut être l'auteur·ice d'une ou plusieurs publications** (de type `Post`).

    ii. `Post.php` définit une classe `Post`. Une publication (*post*) est définie par :

        a. un **identifiant unique** (`int`),
        b. un **slug**,
        c. un **titre**,
        d. une **date de publication** (de type `DateTimeImmutable`),
        e. l'**identifiant de son auteur·ice** (de type `User`).

Les identifiants des `User` et `Post` **commencent à 1 et doivent être incrémentés de manière automatique**.

Pratique : Mini projet sans auto-loading

3. Les autres fichiers :
    i. Le fichier `index.php` est le **point d'entrée de l'application** (*core* et *controller*), produit la sortie,
    ii. Le fichier `config.php` contient des éléments de configuration du projet:
        a. Définit une constante `ABS_PATH` égale à `__DIR__ . '/'`. *Remarque : Cette constante vous sera utile pour importer vos scripts*;
        b. Définit une constante `POSTS_PER_PAGE` égale à `5`, indiquant **le nombre maximum de posts qui doivent être affichés** sur la page,
    iii. Le fichier `db.php` simulera un accès à une source de données (base de données, service web, etc.). Il définit *deux* collections:
        a. `$users` : une collection de 2 User,
        b. `$posts` : une collection de 6 Posts.

Pratique : Mini projet sans auto-loading

4. À sa racine (`/`), le site web doit **produire une page HTML** affichant la liste des publications **du plus récent au plus ancien**, avec le **template HTML** suivant :

```
...
<h1> Dernières publications</h1>
<ul>
    <li>
        <article id="{id article}">
            <h2>{titre article}</h2>
            <p>
                <a href="/{slug}">Lire l'article </a>
            </p>
            <footer>
                <p>
                    Publié le <time datetime="{date publication au format d:m:Y H:i}">{date en français. Ex : 16 mai}</time> par {prénom auteur·ice} {nom auteur·ice}.
                </p>
            </footer>
        </article>
    </li>
    ...
</ul>
...
```

Pour trier les posts, **trouver** la fonction primitive PHP la plus adaptée parmi cette liste.

> Les données dynamiques sont indiquées entre accolades. **Conseil** : pour la date en français, utiliser un formateur international de dates `IntlDateFormatter`

Pratique : Mini projet sans autoloading

{.marp-bg-img}

5. **Servir** le projet **avec le serveur intégré de PHP** sur l'URL `localhost:5001` ;
6. **Tester** : `curl localhost:5001` ou avec votre navigateur favori.

Pratique : Mini projet sans autoloading

Que remarquez-vous sur la gestion des dépendances entre scripts ? Est-ce simple ? Est-ce *fun* ?

L'*autoloading* a pour but de **régler tous ces problèmes**, une bonne fois pour toutes, *pour toujours* !

> **Garder ce projet**, nous le réutiliserons !

> Accéder à/Télécharger une proposition commentée

Les limites de l'import explicite

```php
<?php
include '../../path/to/file1.php';
include '../path/to/file2.php';
include __DIR__ .'/path/to/file3.php';
require ABS_PATH .'path/to/file3.php';
include ...
//Et si on doit inclure 100 fichiers ? Déplacer ce fichier dans un autre repertoire ?
```

Problème avec l'importation explicite :

- **Source d'erreurs** et de confusion !
- Se met **mal à l'échelle** quand le nombre de scripts augmente;
- **Pollue** le code source, illisible;
- Rend l'écriture et la **refactorisation du code extrêmement pénible** (changer le **path** de chaque fichier importé);
- **Intégrer** du code externe (lib, paquet), **distribuer** du code ?

Solution : L'*autoloading*

L'*autoloading* définit **une stratégie pour que l'interpréteur PHP trouve une classe, une interface, une fonction et la charger** à la demande, au moment de l'exécution.

Un *autoloader* est un composant qui permet de charger des *objets* (fonctions, classes, enum, etc.) *sans devoir **explicitement** dire quels fichiers inclure* (avec `require`, `require_once`, `include` ou `include_once`)

En PHP, l'autoloading standardisé (PSR-4) est **basé sur l'utilisation des espaces de noms** (*namespaces*).

Il convient donc d'aborder d'abord les *namespaces*.

Les espaces de noms (Namespaces)

- Introduit en PHP 5.3.0, les namespaces permettent de créer une *hiérarchie virtuelle*, comme un système de fichiers ;
- **Tout composant PHP ou framework moderne utilise les namespaces**. **À la base de tout l'écosystème actuel moderne de PHP**.
- Permet d'écrire du *code isolé* qui peut être utilisé dans n'importe quel projet, *sans conflits de noms*.
- Chaque personne distribue son code sous un namespace *vendor* unique. Par exemple, le namespace *vendor* du framework Symfony est `symfony`.
- Permet de standardiser l'*autoloading* de code-source en utilisant l'*autoloader* PSR-4.

Déclarer un namespace

Toute classe, interface, fonction ou constante vit dans un namespace (ou sous-namespace).

La déclaration d'un namespace se fait au début d'un fichier PHP sur une nouvelle ligne **juste après le tag d'ouverture** `<?php`

```php
<?php
namespace Foo;
echo __NAMESPACE__; //affiche le namespace courant
```

> Pour savoir dans quel namespace vous êtes, utiliser la constante **NAMESPACE**

Sous-namespaces et hiérarchie

On peut déclarer *un sous-namespace* à l'intérieur d'un namespace (comme on peut créer un sous-dossier dans un dossier) :

```php
<?php
namespace Foo\Bar;
```

Les sous-namespaces sont utiles pour *organiser* votre code.

Le namespace le plus important est votre namespace *vendor* (identifiant de votre marque ou de votre organisation). Il doit être **unique** pour distribuer votre code et permettre aux autres de l'utiliser **sans créer de conflits** avec ses propres sources ou dépendances.

Namespaces et fichiers

Les classes, interfaces, etc. d'un même namespace n'ont pas besoin d'être déclarées dans le même fichier.

Si vous déclarez un namespace dans votre fichier, tout ce que contient votre fichier appartiendra à ce namespace.

Namespaces et fichiers

Fichier `Baz.php` :

```php
<?php
namespace Foo

class Baz{}
```

Fichier `Bar.php` :

```php
<?php
namespace Foo

class Bar{}
```

Les classes `Bar` et `Baz` sont déclarées dans deux fichiers différents, mais elles appartiennent au même namespace `Foo`.

Avant les namespaces

Pour éviter *les problèmes de collisions de nom*, on utilisait avant une *convention*, appelée *Zend-style class names* (popularisée par le framework Zend) :

```
class Zend_Cloud_Document_Service_Adapter_WindowAzure_query{
  ...
}
```

- Le *vendor* name est inséré en *préfixe* de la classe. **Cette pratique est toujours utilisée dans la communauté WordPress !**
- Les *underscores*(`_`) permettent de retrouver le path du fichier contenant la classe => `Zend/Cloud/Document/Service/Adapter/WindowAzure/Query.php`

Utiliser les namespaces

Utiliser une classe placée dans un namespace : **Renseigner le nom complètement qualifié** de la classe (nom namespace + nom de la classe).

```php
<?php
//Je dis que j'utilise la classe Response du namespace Symfony\Component\HttpFoundation
$response = new \Symfony\Component\HttpFoundation\Response('Foo', 400);
```

Utiliser les namespaces: import et alias

- **Importer** (`use`): **Dire** à PHP quelle namespace, classe, interface, fonction ou constante je vais utiliser dans ce fichier.
- **Alias** (`as`): dire à PHP que je vais référencer une classe, interface, fonction ou constante importée avec un nom plus court (généralement son nom non qualifié);

```php
<?php
//Importer avec un alias par défaut
use Symfony\Component\HttpFoundation\Response;
//Équivalent à
use Symfony\Component\HttpFoundation\Response as Response;
$response = new Response('Foo', 400);
```

```php
<?php
//Namespace avec alias customisé (à éviter)
use Symfony\Component\HttpFoundation\Response as Res;
$response = new Res('Foo', 400);
```

Utiliser `use`

On **importe** du code avec le mot clef `use`.

Il doit être placé immédiatement **après** le tag ouvrant php `<?php` ou la déclaration du namespace

> Attention, "importer" avec `use` n'est pas équivalent à `require` ! C'est seulement une **déclaration**, pas une instruction pour l'interpréteur PHP. L'*autoloading* que nous verrons après permet de charger une classe (require) à partir d'un use.

`use` IS NOT `require`

Fichier `Foo.php` :

```php
namespace Bar;
class Foo{}
```

Fichier `index.php` :

```php
<?php
use Bar\Foo;
new Foo();// Fatal error ! index.php ne sait pas où trouver la classe Foo !
```

Namespace global

- Si on ne précise pas de namespace, la classe, interface, fonction ou constante existe dans le namespace global, le namespace par défaut.
- Si l'on fait référence à une classe, interface, fonction ou constante sans namespace, PHP **assume qu'elle existe dans le namespace courant** (le namespace *où* la référence est faite).
- Si vous voulez faire référence à une classe dans un namespace *depuis un autre namespace*, **vous devez utiliser le nom entièrement qualifié** = `\Namespace\Nom de la classe`. Par exemple, `\Bar\Foo`.
- Pour faire référence à quelque chose vivant dans le namespace global, vous devez ajouter un `\` au début du nom qualifié pour le rendre complet.

> Il n'est pas nécessaire d'ajouter le `\` devant le nom qualifié dans une déclaration avec le mot-clé `use` car PHP *assume* qu'il est entièrement qualifié.

Nom non qualifié, qualifié et entièrement qualifié

```php
namespace Bar;
class Foo{}
```

- Nom *non qualifié* : `Foo`

> PHP recherche dans le namespace courant, si non trouvé recherche dans le namespace global

- Nom *qualifié* (relatif) : `Bar\Foo`

> PHP ajoute le namespace courant comme préfixe

- Nom *entièrement qualifié* : `\Bar\Foo`

> PHP l’interprète comme Bar\Foo sans faire aucune supposition (aucune ambiguïté)

Exemple d'erreur classique

```php
<?php
namespace Bar;

class Foo{
  public function doSomething(){
    $exception = throw new Exception(); // Fatal error !
  }
}

new Foo()->doSomething();
```

Résultat :

```
PHP Fatal error:  Uncaught Error: Class "Bar\Exception" not found in ...
```

Le code crash, pourquoi ?

Solution

```php
<?php
namespace Bar;
class Foo{
  public function doSomething(){
    $exception = throw new \Exception(); // OK !
  }
}
```

Dans l'exemple précédent, PHP cherche la classe `Exception` dans le namespace courant `Foo`, or elle n'existe pas ici. Ici on veut bien utiliser la classe *built-in* `Exception` de PHP qui existe dans le namespace global. Il faut donc indiquer le nom qualifié complet avec le `\`.

Exercice : se familiariser avec les namespaces et la résolution des noms

Pour chaque appel, dites quelle classe ou fonction PHP essaie d'invoquer.

```php
<?php
namespace A;
use B\D, C\E as F;

foo();
\foo();
my\foo();
F();
new B();
new D();
new F();
new \B();
new \D();
new \F();
B\foo();
```

Solution

```php
<?php
namespace A;
use B\D, C\E as F;

foo();      // tente d'appeler la fonction "foo" dans l'espace de noms "A"
            // puis appelle la fonction globale "foo" (uniquement pour les fonctions)

\foo();     // appelle la fonction "foo" définie dans l'espace de noms global

my\foo();   // appelle la fonction "foo" définie dans l'espace de noms "A\my"

F();        // tente d'appeler la fonction "F" définie dans l'espace "A"
            // puis tente d'appeler la fonction globale "F"

new B();    // crée un objet de la classe "B" définie dans l'espace de noms  "A"

new D();    // crée un objet de la classe "D" définie dans l'espace de noms  "B"

new F();    // crée un objet de la classe "E" définie dans l'espace de noms  "C"

new \B();   // crée un objet de la classe "B" définie dans l'espace de noms global

new \D();   // crée un objet de la classe "D" définie dans l'espace de noms global

new \F();   // crée un objet de la classe "F" définie dans l'espace de noms global

B\foo();    // appelle la fonction "foo" de l'espace de noms "A\B"
```

Gérer les dépendances entre fichiers

- Comment *automatiser* et *standardiser* l'import de sources ?
- Comment utiliser les *namespaces* pour mieux gérer les dépendances entre fichiers ?

L'autoloading avant, avec `__autoload()`

PHP 5 avait introduit la fonction "magique" __autoload().

> Ne pas utiliser. **Elle est devenue obsolète et a été supprimée dans PHP 8.0.0 !**

Elle servait de *hook* pour implémenter une stratégie de chargement de classes :

```php
<?php
function __autoload($className) {
     include $className . '.php';
}
//__autoload() va automatiquement être appelée par l'interpréteur PHP car la classe Foo n'est pas connue
$foo = new Foo();
```

Problème : **vous ne pouviez la définir qu'une fois !**

L'autoloading *avant* : `spl_autoload_register()`

La fonction `spl_autoload_register()` permet d'enregistrer une implémentation d'`__autoload()` dans une *pile* d'appels (file d'attente)

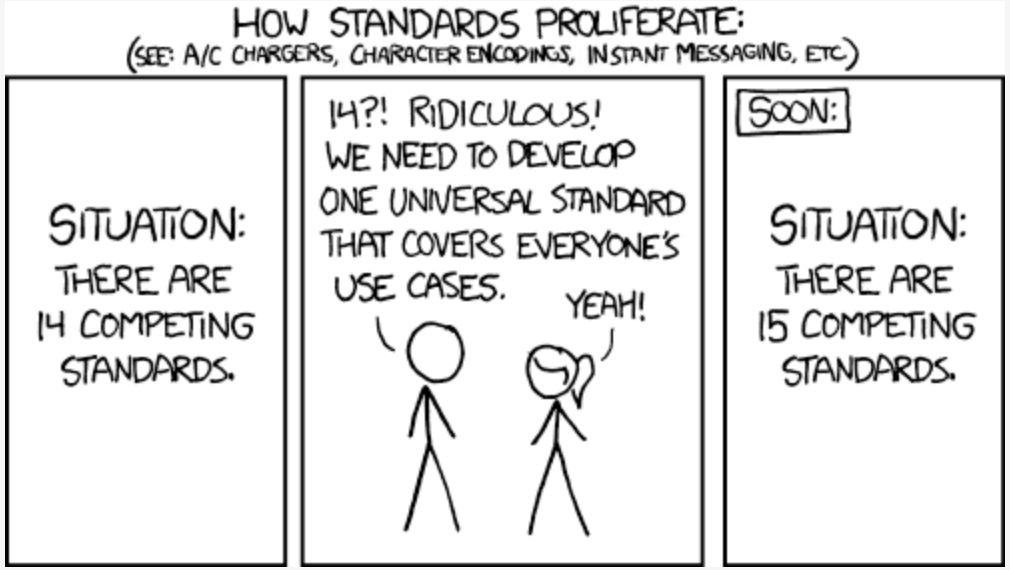
```php
function my_autoloader($class) {
    include 'classes/' . $class . '.class.php';
}

spl_autoload_register('my_autoloader');

// Ou, en utilisant une fonction anonyme
spl_autoload_register(function ($class) {
    include 'classes/' . $class . '.class.php';
});
```

> Si vous devez utiliser plusieurs fonctions d'autochargement, la fonction `spl_autoload_register()` est faite pour cela. Elle crée une file d'attente de fonctions d'autochargement, et les exécute les unes après les autres, dans l'ordre où elles ont été définies.

Problèmes

- Plusieurs *stratégies* d'*autoloading* possibles dans *le même projet* ;
- Chaque *vendor* va avoir sa propre stratégie. **Vous devez comprendre** comment chaque composant ou framework implémente **son** *autoloading*.

Solution ? Standardiser l'autoloading



> xkcd : Standards

L'autoloading *maintenant* : l'autoloader PSR-4

Le PHP-FIG a proposé une spécification d'*autoloader* pour standardiser la stratégie d'autoloading dans le PSR-4.

Avec le PSR-4 : **Autoloading standard et interopérable**.

N'importe qui peut utiliser des composants, frameworks **avec un et un seul autoloader** !

> Si vous écrivez et distribuez des composants PHP, **faites en sorte de respecter le PSR-4** sinon personne ne voudra utiliser votre code. La majeure partie des organisations le font : Symfony, Doctrine, Monolog, Twig, Guzzle, PHPUnit, Carbon, etc.

L'autoloader PSR-4 : une stratégie standardisée

L'**autoloader PSR-4** implémente une stratégie pour **trouver** et **charger** (`require`) automatiquement :

- Les **classes** ;
- Les Interfaces ;
- Les Traits ;
- Les Enums;

Cette stratégie est **basée sur les namespaces**.

> On ne peut **pas** autoloader de simple fonctions, variables ou constantes (define)

L'autoloader PSR-4 : une stratégie standardisée

Cette recommandation impose une **contrainte** sur :

- L'organisation de vos fichiers sources;
- Vos namespaces;

L'autoloader PSR-4 : une stratégie standardisée

Principe : un `use` est associé à un `require` Pour cela, **la hiérarchie des namespaces doit correspondre à la hiérarchie des fichiers sources**.

L'autoloader PSR-4 en pratique

1. **Point d'entrée:** Faire correspondre un namespace de haut niveau `ENI\App` à un répertoire, par exemple `src`.
2. A présent PHP sait (grâce à l'*autoloader*) que toute classe ou interface déclarée dans le namespace `ENI\App` se trouve dans le dossier `src`
3. **Organisation de votre code:** Faites *correspondre les sous-namespace aux sous-dossiers*.

> Par exemple `ENI\App\ModernPHP` correspond au dossier `src/ModernPHP`.
> La classe `ENI\App\ModernPHP\Foo` correspond au fichier `src/ModernPHP/Foo.php`

Vous n'avez pas besoin d'écrire votre propre autoloader, vous pouvez en utiliser un généré automatiquement par Composer, le gestionnaire de dépendances. Nous y reviendrons.

> L'autoloader PSR-4 utilise, évidemment, sous le capôt la fonction `spl_autoload_register()` !

Composer, le gestionnaire de paquets moderne de PHP

Composer

Composer est :

- Un gestionnaire de composants (de dépendances) **puissant** et (vraiment) **simple** à utiliser !
- Gestion de l'autoloading (PSR-4)

Packagist

Packagist est le dépôt public principal où sont publiés des composants PHP. On y retrouve tous les *vendor* majeurs.

> À ne pas confondre avec PECL, le dépôt des extensions de la machine virtuelle PHP !

Vous dites à Composer quel composant vous voulez et Composer :

- Le **télécharge** ;
- L'**autoload** dans votre projet ;
- Télécharge également les **dépendances** de votre composant (et leurs dépendances etc.)

Installer Composer

Suivre les instructions ici pour installer et ici pour le télécharger

```
php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');"
php -r "if (hash_file('sha384', 'composer-setup.php') === '55ce33d7678c5a611085589f1f3ddf8b3c52d662cd01d4ba75c0ee0459970c2200a51f492d557530c71c15d8dba01eae')
{ echo 'Installer verified'; } else { echo 'Installer corrupt'; unlink('composer-setup.php'); } echo PHP_EOL;"
php composer-setup.php
php -r "unlink('composer-setup.php');"
```

Pour l'installer globalement (Ubuntu/Debian) :

```
sudo mv composer.phar /usr/local/bin/composer
```

Dans un terminal :

```
composer -V
Composer version 2.3.8 2022-07-01 12:10:47
```

Utiliser Composer : installer et gérer ses dépendances

Nous allons installer le composant guzzlehttp/guzzle, un client HTTP puissant et performant.

Les noms du composant

- Vendor name (unique): `guzzlehttp`
- Package name : `guzzle`
- Nom complet: `guzzlehttp/guzzle`

> Packagist utilise la convention `vendor/package` pour éviter les collisions de nom.

Installer un composant: `composer require`

```
composer require guzzlehttp/guzzle
```

Composer crée deux fichiers :

- `composer.json` : composants installés du projet (prod, dev);
- `composer.lock` : liste toutes les dépendances avec leur version utilisée (permet de cloner le projet dans le même état).

> Versionner ces deux fichiers dans votre contrôle des sources (Git) !

`composer install` et `composer update`

```
composer //liste toutes les commandes
```

- `composer install` : **installe toutes les dépendances** du projet en respectant les versions déclarées dans le `composer.lock` ;
- `composer update` : **télécharge les dernières versions des composants** (dans les limites de votre politique de maj *semver*), et met à jour `composer.lock`

L'autoloading

Comment utiliser les composants ?

Dans votre code

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

Et *voilà* !

Exemple: utiliser le composant `guzzle`

On a installé `guzzlehttp/guzzle`. Dans notre projet :

```php
<?php
require 'vendor/autoload.php';
$client = new \GuzzleHttp\Client();
```

ou

```php
<?php
require 'vendor/autoload.php';
use GuzzleHttp\Client;
$client = new Client();
```

> La classe `Client` existe bien dans le fichier `src/Client.php`

Créer un composant

Pourquoi faire de son projet *un composant* ?

- Partager et distribuer son code ;
- Créer des **modules** réutilisables ;
- Standardiser : Suivre les mêmes standards pour son application que pour ses dépendances (PSR-4) ;
- Maintenabilité : Utiliser l'*autoloading* et la norme PSR-4 ! (plus de `require` ni de path !);

L'*autoloading* de fonctions

L'écosystème actuel de PHP et son système de gestion de dépendances *via* composer est **fortement basé sur l'orienté objet (classes, interfaces, traits, enums)**. Une classe peut être vu ici comme un *namespace* regroupant un ensemble de fonctions.

> Amené à changer dans le futur... On a pas toujours envie d'utiliser la POO !

Si l'on ne souhaite **pas utiliser de POO**, il est possible d'"autoloader" de simple fonctions. Pour cela, utiliser la clef `files` de l'`autoload` dans `composer.json` :

```
"autoload": {
        "psr-4": {
            "Vendor\\Package\\": "src/"
        },
        "files": ["src/functions_include.php"] <==== ICI
    },
```

Tout fichier listé ici sera *inclus* (`require`) automatiquement au démarrage (utile pour des constantes également).

Remarques sur le versionnement

Pensez à :

- **Versionner** vos fichiers `composer.json` et `composer.lock` ;
- **Ignorer** le répertoire `vendor` en l'ajoutant au `.gitignore` .

> Vous ne voulez pas pousser le code source de vos dépendances avec votre projet. Les fichiers `composer.json` et `composer.lock` sont là pour éviter cela !

Récapitulatif : Création d'un projet PHP *moderne*

1. Initialiser votre projet avec `composer init` (création de `composer.json` et `composer.lock`);
2. Renseigner un namespace (votre namespace `<vendor>/<nom application>`) mappé au dossier `src` dans le `composer.json`;
3. Placer tout votre code source sous le namespace mappé au dossier `src` (étape 3);
4. Versionner votre projet avec Git en incluant les fichiers `composer.json` et `composer.lock`, et **en excluant** **`vendor`** dans un fichier `.gitignore`;

> Le namespace renseigné à l'étape 3 ne doit pas nécessairement correspondre à votre *vendor/package* name. Ce nom sert seulement à Composer et Packagist pour identifier votre paquet parmi tous les paquets présents sur Packagist et dans votre projet.

Toutes vos classes, interfaces doivent vivre dans le dossier `src` et sous le namespace renseigné dans le fichier `composer.json`

Pratique

1. **Refactoriser** le projet réalisé précédemment `demo_autoloading` afin qu'il **utilise l'autoloading PSR-4 et le gestionnaire de dépendances Composer** :
    i. Initialiser le projet avec `composer init` ;
    ii. **Appliquer les modifications nécessaires** (retirer les `require`);
    iii. Déplacer le fichier `congig.php` dans `src` pour qu'il soit chargé automatiquement;
    iv. Commentaires :
        a. `index.php` joue le rôle de *client* du contenu `src` . Il invoquera l'*autoloader* (`require ./vendor/autoload.php`).
        b. Laisser `db.php` en dehors de la stratégie d'*autoloading*.

Voici la structure finale du projet :

```
├── composer.json
├── db.php
├── index.php
├── README.md
├── src
│   ├── config.php
│   └── Models
│       ├── Post.php
│       └── User.php
└── vendor
```

Pratique

2. **Installer** le composant var_dumper
3. L'**utiliser** dans `index.php` pour afficher la collection de posts.
4. **Créer** un fichier `.gitignore` correcte pour versionner le projet.

En résumé

- Vous avez découvert les nouveautés de PHP 8 ;
- Vous êtes familier·e avec la notion de namespace;
- Vous savez comment utiliser l'écosystème moderne de PHP avec `Composer`, `Packagist` et l'`autoloading` ;
- Vous pouvez à présent créer des applications ou modules **utiles**, faciles à versionner, **maintenir** et **distribuer**;
- Écosystème PHP riche et bien vivant ! (435 402 paquets sur Packagist au 09/11/25);

La gestion des erreurs en PHP

Une erreur se déclenche lorsque votre script PHP ne peut pas s’exécuter comme prévu (par exemple un script à charger n'existe pas, une autorisation de lecture de fichier est refusée, l'espace disque est insuffisant, une classe n'existe pas, etc.).

Une erreur est caractérisée par 4 valeurs :

- Un **niveau**. Chaque niveau est défini par une constante PHP dont le nom commence par `E_`. Par exemple, une erreur de niveau 1 est une **erreur fatale** définie par la constante `E_ERROR`.
- Un **message**;
- Le **nom du fichier PHP** dans lequel l'erreur s'est produit ;
- Le **numéro de ligne** du fichier où l'erreur s'est produite.

En cas d'erreur

- Lorsqu'une erreur est émise, **un rapport est généré** par le système de gestion d'erreurs de PHP;
- **Par défaut**, lorsqu' aucune politique de gestion n'est mise en place, **le rapport est simplement affiché sur la sortie standard**;
- Chaque rapport fournit les quatre informations mentionnées ci-dessus, suivi de la trace d'appels (*stack trace*);
- Une fois la génération du rapport terminée, si le niveau d'erreur le permet, l’exécution du programme continue. Sinon, le programme s'arrête.

Niveaux d'erreur et comportements

- Erreur est **fatale** (*Fatal Error*, de niveau 1): le programme reporte l'erreur (par défaut sur la sortie standard) et **le programme s'arrête immédiatement** (*crash*) ;
- Erreur est une **alerte** (*Warning*, de niveau 2): le programme reporte l'erreur et **continue** son exécution ;
- Soit l'erreur n'en est peut-être pas une, c'est une simple **remarque** (*Notice*, de niveau 8) : le programme reporte l'erreur et continue son exécution.

Exemple

```php
<?php
//Le fichier foo.php n'existe pas
require 'foo.php';
echo 'Tout va bien' . PHP_EOL;
```

Que va-t-il se passer ? Comment cela va-t-il mal se passer ? Et si on remplace `require` par `include` ?

La gestion des erreurs en PHP, *what a mess...*

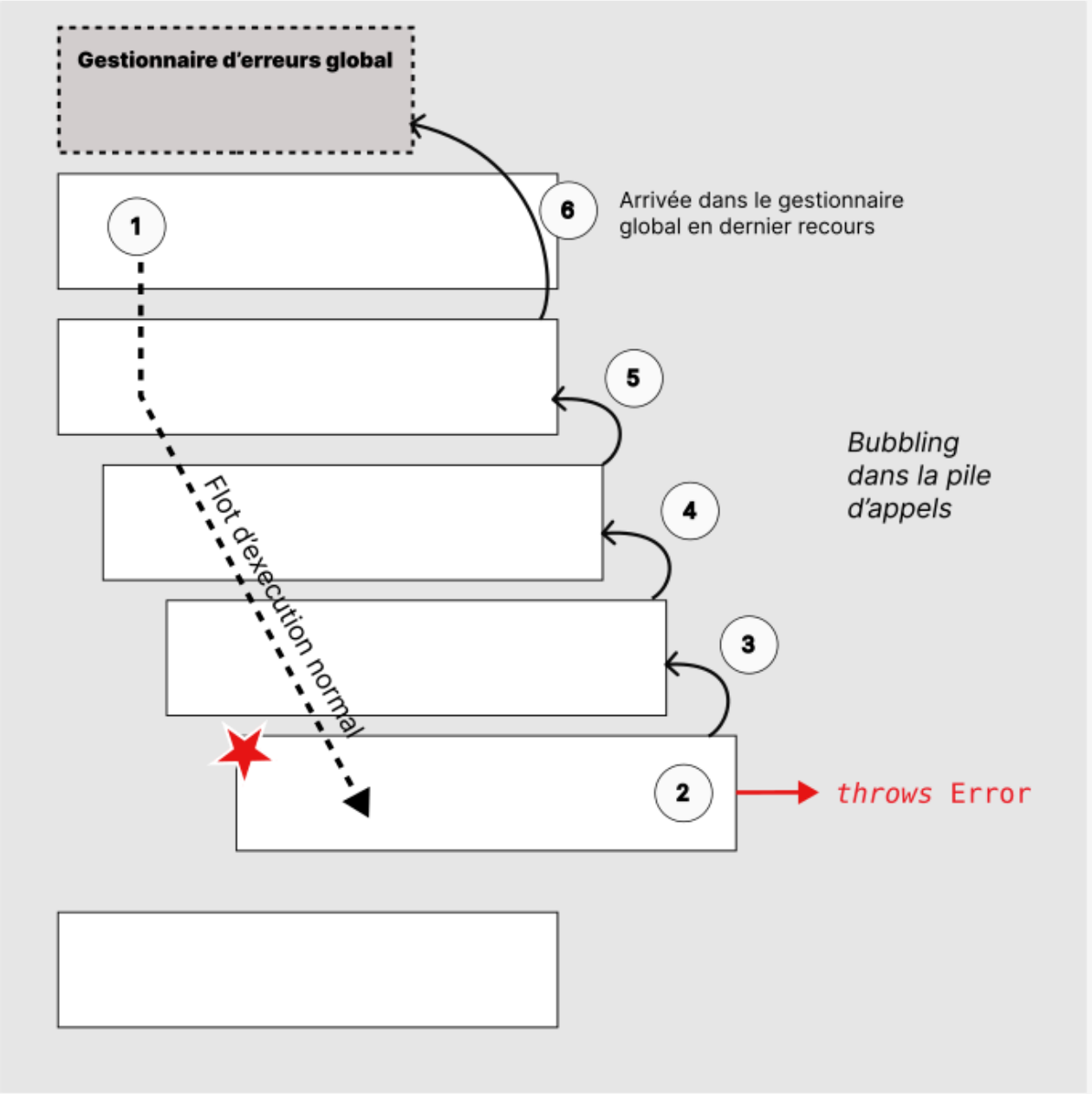
- PHP < 5 : erreurs uniquement *via* le système de reporting d'erreurs classique (*warnings*, *notices*, *fatals*), gérées par le **gestionnaire d'erreurs global** ;
- PHP 5 : **introduction des Exceptions**, mais les erreurs du moteur restent non attrapables;
- PHP 7 : introduction de la hiérarchie `Error` + interface `Throwable`. **La majorité des erreurs internes** (mais pas toutes, comme fopen...) **deviennent des objets de type** `Error` (`TypeError`, `ArithmeticError`, etc.), **attrapables**, comme les exceptions.

Bubbling des erreurs

{.marp-bg-img}

Une erreur, une fois émise (*thrown*), **remonte la pile d'exécution** (bubbling de l'erreur) jusqu'à être attrapée:

- **Localement** par un bloc `try/catch(Error)` ;
- Par **un gestionnaire d'erreurs global**, qui agit comme un **garde-fou de dernier recours**.



Bubbling d'une `Error` et Gestionnaire d'erreurs global

Gestionnaire d'erreurs global

Toujours en enregistrer un, si vous manquez de `catch` une `Error` localement.

```php
set_error_handler(function ($errno, $errstr, $errfile, $errline) {
    echo "Nous sommes désolés, un problème vient de survenir :/ \n Nous vous invitons à revenir plus tard." . PHP_EOL;
    //Si c'est une erreur de niveau 2 (avertissement), l'exécution peut "en principe" continuer
    // mais on décide, par précaution, d'arrêter là les frais
 if ($errno === E_WARNING)
  die;
});
```

Gestionnaire d'erreurs local, avec un bloc `try/catch`

```php
<?php
try {
    //Le fichier foo.php n'existe TOUJOURS pas
 require 'foo.php';
 echo 'Cette instruction ne sera pas executée si require émet une erreur' . PHP_EOL;
} catch (Error $e) {
 echo "L'erreur " . $e->getMessage() . " est gérée localement" . PHP_EOL;
}
echo 'Le programme continue ici' . PHP_EOL;
```

La gestion des erreurs en PHP, *what a mess...*

```php
<?php
set_error_handler(function ($errno, $errstr, $errfile, $errline) {
 echo 'Par contre, fopen, comme tout le monde, reporte bien ses erreurs auprès du gestionnaire d\'erreurs global.' . PHP_EOL;
});
try {
    //On essaie d'ouvrir un fichier foo.php qui n'existe pas. Fatal Error
 fopen('foo.php', 'r');
} catch (Error $e) {
 echo 'Vous ne verrez jamais ce message car fopen n\'émet pas d\'erreurs de type Error. Cette fonction utilise toujours l\'ancien système de reporting' . PHP_EOL;
}
```

Règles de gestion des erreurs

Voici les règles que vous devriez suivre :

- Ne **désactivez jamais votre rapport d'erreur**. Certaines fonctions natives sont encore sur l'ancien système de reporting;
- **Afficher les erreurs dans l'environnement de développement mais n'affichez pas les erreurs dans un environnement de production**
- **Faites toujours un log** (enregistrement horodaté dans un fichier dédié) **de vos erreurs**, **en développement comme en production**. Cela vous fournira des informations précieuses pour comprendre ce qu'il s'est mal passé et déboguer votre code

Exemple de configuration des logs

Script `error.php` :

```php
<?php
set_error_handler(function ($errno, $errstr, $errfile, $errline) {
 error_log("[$errno] $errstr in $errfile:$errline");
 return false;
});
try {
    //On essaie d'ouvrir un fichier foo.php qui n'existe pas. Fatal Error
 require 'foo.php';
} catch (Error $e) {
 echo 'Impossible de charger foo.php. Ce n\'est pas grave, on continue.'. PHP_EOL;
}
```

Configuration du reporting (DEV)

Fichier de configuration `php-dev.ini` :

```
;Afficher les erreurs
display_startup_errors = On
display_errors = On
;Rapporter toutes les erreurs
error_reporting = -1
;Activer le log d'erreurs
log_errors = On
;Placé dans le repertoire courant pour la démo
error_log = ./php-cli.log
```

```
php -c php-dev.ini error.php
```

Voir les directives pour configurer PHP au runtime

Configuration du reporting (PROD)

Fichier de configuration `php-prod.ini` :

```
;N'afficher SURTOUT PAS les erreurs
display_startup_errors = Off
display_errors = Off
;Rapporter toutes les erreurs SAUF les notice (pour ne pas polluer vos logs)
error_reporting = E_ALL & ~E_NOTICE
;Activer le log d'erreurs
log_errors = On
;Placé dans le repertoire courant pour la démo
error_log = ./php-cli-prod.log
```

```
php -c php-prod.ini error.php
```

Ne lancez pas des erreurs, gérez-les

Il n'est pas recommandé d'utiliser les erreurs PHP lorsque l'on souhaite émettre une erreur (mais plutôt les exceptions). **Les erreurs doivent être uniquement gérées** pour les fonctions natives de PHP.

- Le système `Error` est **réservé à PHP lui-même** ;
- Votre code applicatif ne doit pas lancer d'objets de type `Error` (il doit lancer des objets de type `Exception`)

Gestion des erreurs avec les exceptions

{.marp-bg-img}

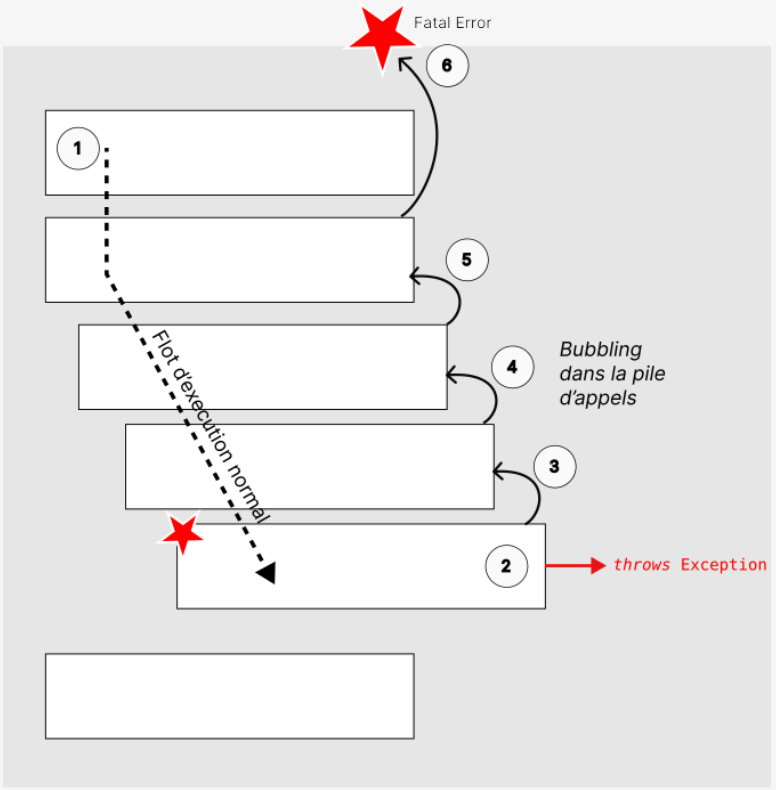
Aujourd'hui, **toute la communauté utilise les exceptions dans le code applicatif**.

Une exception non traitée **provoque une erreur fatale et l'arrêt du programme** (*crash*)

```php
throw new Exception("Une exception"); //Si non attrapée, crash le programme.
```

- Vous pouvez traiter localement une exception dans un bloc `try/catch` ;
- Vous pouvez enregistrer un **gestionnaire global d'exceptions** (filet de sécurité)

> En savoir plus sur le système d'exceptions de PHP



Bubbling d'une `Exception` dans la pile

Créer ses propres exceptions

Vous pouvez **définir vos propres exceptions**, en étendant la classe `Exception` ;

```
class MyException extends Exception { }
```

Gestion locale des Exceptions

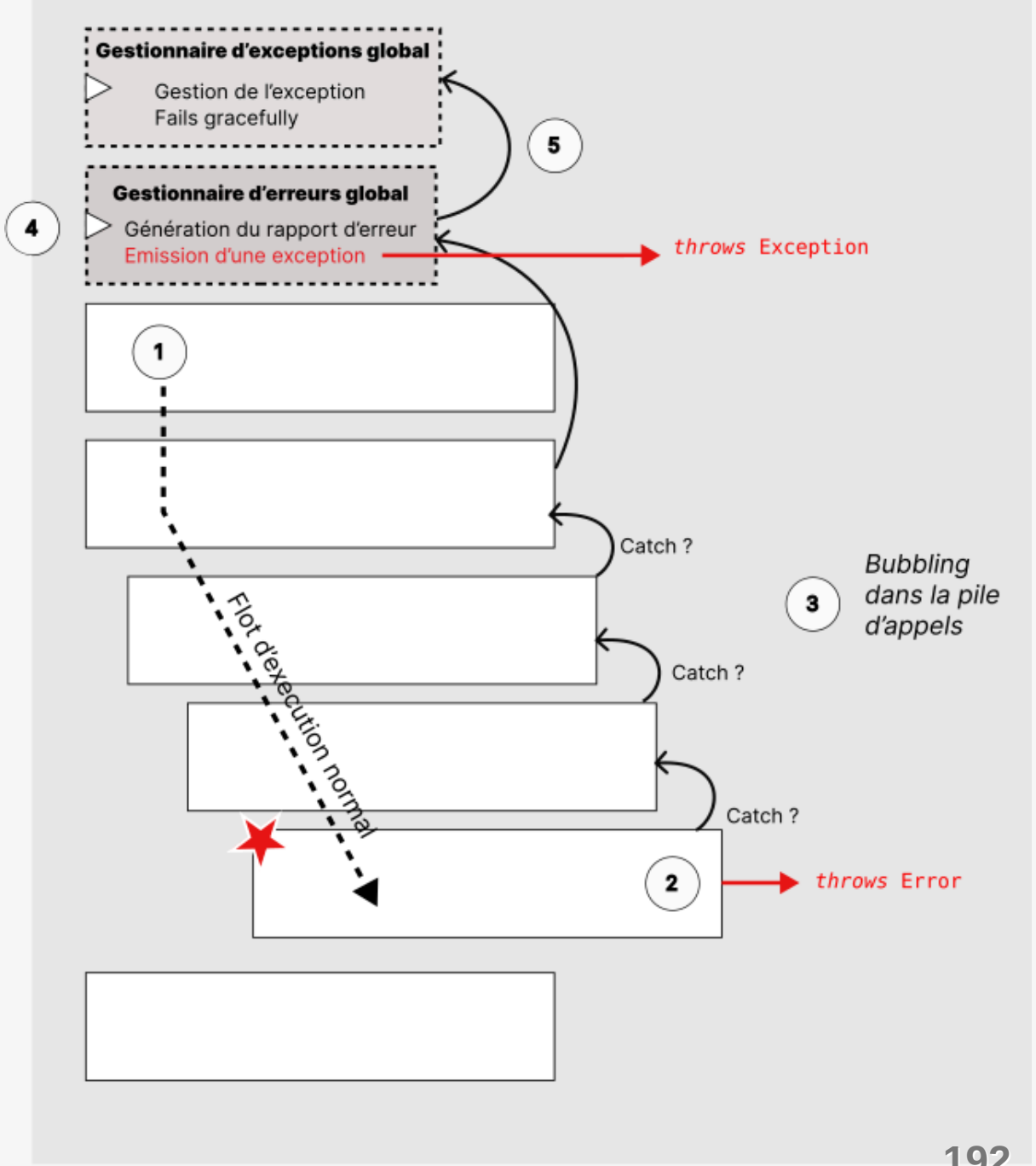
```php
try {
    $pdo = new PDO('mysql://host=foo;dbname=bar');
} catch (Exception $e) {
    //Inspecter l'exception
    $message = $e->getMessage();
    echo "Mmmm, impossible d'accéder à la base de données. Veuillez rééssayer plus tard.";
}
```

Gestion complète des erreurs

{.marp-bg-img}

Pour palier à toutes les éventualités :

- **Convertir** toutes les erreurs (ancien et nouveau système) en exception;
- **Enregistrer un gestionnaire global** d'exceptions;
- **Traiter localement** les exceptions, sur la pile d'appels.



Transformer une `Error` en `Exception`

Gestion complète des erreurs, exemple

```php
//Définition d'un gestionnaire d'erreurs global
//Remarque: certaine fonctions PHP utilisent encore le système de reporting, comme fopen
set_error_handler(function ($errno, $errstr, $errfile, $errline) {
    //Transformer l'erreur en Exception avec la classe dédiée ErrorException
    throw new ErrorException($errstr, $errno, 0, $errfile, $errline);
});

//Définition d'un gestionnaire d'exceptions global
set_exception_handler(function (\Throwable $e) {
    echo 'Une Exception a été détectée. Nous mettons fin au programme.' . PHP_EOL;
    var_dump($e->getMessage());
});

try {

    //On essaie d'ouvrir un fichier foo.php qui n'existe pas. Fatal Error (ancien système)
    fopen('foo.php', 'r');
} catch (Error $e) {
 echo 'Vous ne verrez jamais ce message car fopen n\'émet pas d\'erreurs de type Error. Cette fonction utilise toujours l\'ancien système de reporting' . PHP_EOL;
}
```

Tooling et qualité du code en PHP

PHP est un langage :

- Interprété, non compilé ;
- typé dynamiquement, non statiquement.

Sans compilateur, les **erreurs arrivent** donc... à l'**execution** !

S'équiper des *bons* outils :

- **Linter** (PHP_CodeSniffer);
- **Analyse statique de code** (PHPStan);
- **Générateur de documentation** (PHP Documentor);
- **Tests** (PHPUnit)

> PHP, grâce à sa longévité et communauté dispose de nombreux outils très puissants et matures pour assister le développement ! Voir d'autres outils utiles

Configurer son IDE, avec VS Code

- **Autocomplete les balises PHP**, créer ces *snippets* dans `Preferences/Configure Snippets` (`php.json`):

```
{
    "php": {
  "prefix": "php",
  "body": [
   "<?php $0 ?>"
  ],
  "description": "Open and close a php tag"
 },
    "ph": {
  "prefix": "ph",
  "body": [
   "<?php",
   "$0"
  ],
  "description": "Open a php tag (without closing it)"
 },
}
```

Configurer son IDE, avec VS Code

- **Installer les extensions suivantes** :
  - Intelephense, **la base** (doc au survol, autocomplete, etc.)
  - Auto Rename Tag, renommer les balises HTML correspondantes;
  - Format HTML in PHP, formater le HTML dans des scripts PHP
  - PHP DocBlocker, snippets de Doc Block PHP
  - Hooks IntelliSense for WordPress, doc WordPress core intégrée, auto complétion, etc.

PHP_CodeSniffer et coding standards

Utiliser un **linter**, comme PHP_CodeSniffer. **Outil essentiel de développement** pour assurer un code propre et cohérent, pour le versionnement (même règles pour toute l'équipe).

PHP_CodeSniffer est composé de **deux programmes** :

- `phpcs` : **détecte** la violation de standards dans les fichiers **PHP**, CSS et JS (***c**ode **s**niffer*)
- `phpcbf` : **corrige** automatiquement les violations détéctées (***c**ode **b**eautifier and **f**ixer*)

Installer PHP_CodeSniffer (*via* Composer)

> Remarque : Le projet a récemment changé de mainteneur officiel !

Installation globale

```
composer global require "squizlabs/php_codesniffer=*"
```

ou locale au projet :

```
composer require --dev "squizlabs/php_codesniffer=*"
```

> Le mot clef `global` dit à Composer d'installer le module globalement pour qu'il soit accessible à tous les projets sur la machine hôte.

Tester l'installation de `PHP_CodeSniffer`

Si installé **globalement**, s'assurer que le répertoire `composer` global est sur le `PATH`, puis :

```
phpcs -h
phpcbf -h
```

Si installé **localement**, à la racine du projet :

```
./vendor/phpcs -h
./vendor/phpcbf -h
```

Utilisation de `PHP_CodeSniffer`

Avec les standards par défaut (standard PEAR) :

```
phpcs /path/to/code-directory
```

Avec d'autres standards, par exemple PSR12 :

```
phpcs --standard=PSR12 /path/to/code-directory
```

Appliquer un standard par défaut :

```
phpcs --config-set default_standard PSR12
```

> À intégrer dans une pipeline CI ! `phpcs` retourne un code non zéro en cas d'erreur (`echo $?`)

Pratique : Appliquer les *codings standards* PSR12 au projet

```
# Lister les standards installés
phpcs -i
phpcs --standard=PSR12 src
```

Puis, pour **corriger** les violations qui peuvent l'être de manière automatique :

```
phpcbf --standard=PSR12 src
```

Corriger les autres manuellement.

Vérifier jusqu'à ce que phpcs retourne un code égal à 0 (aucune erreur) : `echo $?`

Utiliser les standards de WordPress (*core*)

1. Installer les codings standards utilisés par WordPress avec composer;
2. Utiliser les coding standards avec `phpcs` et `phpcbf` :

```
# Vérifier que le standard WordPress est bien installé
phpcs -i
# Vérifier que la config utilisée
phpcs --config-show
# "Sniffer" avec progress bar
phpcs -ps --standard=WordPress src
```

Utilisation dans VS Code

1. Installer l'extension PHP Sniffer & Beautifier

> L'extension **fournit une interface dans VS Code** aux programmes `phpcs` et `phpcbf` (installés localement ou globalement), **elles ne les installent pas** !

2. Choisir `PHP Sniffer & Beautifier` comme **formateur par défaut**.

Analyse statique de code avec PHPStan

Analyse statique de code pour détecter des bugs, erreurs *avant* l'execution.

- Installer PHPStan.
- Analyser au niveau 8;

```
./vendor/bin/phpstan analyze -l8 src
```

> À intégrer dans une pipeline CI ! `phpcs` retourne un code non zéro en cas d'erreur (`echo $?`)

Générer de documentation à partir des Docblocks avec PHP Documentor

Utiliser PHP Documentor pour générer automatiquement la documentantion de référence du projet. Basé sur les DocBlock

> DocBlock n'est pas natif au PHP. Format de commentaire.

Version *standalone* avec Docker :

```
#télécharger et éxecuter l'image docker phpdoc
docker run --rm -v "$(pwd):/data" "phpdoc/phpdoc:3"
#créer un alias pour l'usage
alias phpdoc="docker run --rm -v $(pwd):/data phpdoc/phpdoc:3"
```

Pratique

Générer la doc du dossier `src` :

```
docker run --rm -v "$(pwd):/data" phpdoc/phpdoc:3 -d src -t docs
```

Tests unitaires avec PHPUnit

Tests et suites de test avec PHPUnit :

```
#installer
composer require --dev phpunit/phpunit ^12
#tester
./vendor/bin/phpunit --version
```

Suivre le guide

Pratique

Créer un test unitaire dans le fichier `tests/PostTest.php` :

```php
<?php

use PHPUnit\Framework\TestCase;
use Ps\App\Models\Post;

final class PostTest extends TestCase
{
    protected function setUp(): void
    {
        // Réinitialiser $last_id avant chaque test
        $reflection = new \ReflectionProperty(Post::class, 'last_id');
        $reflection->setAccessible(true);
        $reflection->setValue(null, 1);
    }

    public function testAutomaticIdIncrement(): void
    {
        $post1 = new Post(slug: 'first', title: 'First Post', date_publication: new \DateTimeImmutable(), author: 1);
        $post2 = new Post(slug: 'second', title: 'Second Post', date_publication: new \DateTimeImmutable(), author: 2);

        $this->assertSame(1, $post1->id);
        $this->assertSame(2, $post2->id);
    }
}
```

Lancer les tests :

```
./vendor/bin/phpunit --testdox tests
```

Pratique

Si ce n'est pas encore le cas, **tester** ces outils sur le projet `demo_autoloading`.

En résumé

- Vous savez configurer votre IDE (VS Code) pour travailler efficacement sur des projets PHP et WordPress;
- Vous savez utiliser un **linter** et un **coding standard** et l'intégrer à l'IDE;
- Vous savez utiliser un **analyseur statique de code**;
- Vous savez couvrir votre code de **tests unitaires** avec PHPUnit;

PHP et WordPress

PHP et WordPress

- Écrire du PHP *"WordPress core"* (conventions, pratiques usuelles);
- Quelques cas pratiques;
- Écrire du PHP moderne *dans ses modules*;
- Organiser son projet.

PHP et WordPress, une histoire à part

WordPress 1.0 date de 2004. A cette époque, PHP était à la version 4.X !

- Pas de namespaces ;
- POO balbutiante ;
- Pas de gestionnaire de dépendances;

WordPress a **construit son écosystème et ses conventions** pour compenser les limitations de PHP 4, tout en voulant rester **rétrocompatible** avec les anciennes versions de PHP :

- **Fonctions globales** plutôt que des classes;
- **Hooks** plutot que polymorphisme;
- Templates procéduraux **sans moteur moderne**;

> "PHP is dead" = "PHP 4 is dead" ! PHP moderne commence avec PHP 5 et parachevé avec PHP 7.

Conventions dans WordPress 1/3

Quelques conventions à appliquer pour développer sous WordPress :

- **REGLE IMPORTANTE : Préfixer toutes les fonctions, classes, variables globales et hooks avec un préfixe unique** pour éviter les collisions avec le cœur de WordPress ou d'autres plugins (*vendor*) :

```
function vendor_afficher_message() {}
class Vendor_Admin {}
```

- **Toujours utiliser un préfixe unique** ;
- Utiliser uniquement les balises `<?php` et `?>` (pas de `<?`, ni `<?=`);
- **Utiliser des guillemets simples et doubles selon le contexte**. Si rien n'est évalué dans la chaîne, utiliser des guillemets simples.

Conventions dans WordPress 2/3

- Utiliser uniquement des **lettres minuscules pour les variables, les actions/filtres et les fonctions** (ne jamais utiliser le `camelCase`) :

```
function some_name( $some_variable ) {}
```

- Les fichiers doivent être **nommés de manière descriptive** en utilisant des **lettres minuscules**. Des **tirets** doivent séparer les mots (`mon-plugin-nom.php`);

Conventions dans WordPress 3/3

- Les classes, interfaces, traits, enums écrits avec **des noms capitalisés** et séparés par des **underscores** :

```
class Walker_Category extends Walker {}
class WP_HTTP {}

interface Mailer_Interface {}
trait Forbid_Dynamic_Properties {}
enum Post_Status {}
```

- Les fichiers contenant des classes préfixées : `class-*.php`
- Les tableaux doivent être déclarés en utilisant **la syntaxe longue** :

```
$arr = [ 1, 2, 3 ]; //Non (sort array syntax)
$arr = array(1, 2, 3); //Oui (long array syntax)
```

- **Utiliser le type hinting**;
- En général, la **lisibilité** prime sur l'ingéniosité ou la concision (**pas de code "clever"**);
- **Pas de commandes shell** (no backtick operator)

> La plupart de ces conventions de syntaxe peuvent être appliquées *via* les coding standards+linter !

Sécurisation des données

Les formulaires exposées par un service web permettent à n'importe quel client de soumettre des données. Le client peut envoyer n'importe quelle donnée, autant de fois qu'il le désire et peut essayer, s'il est mal intentionné, de briser ou de pénétrer dans votre service web.

Chaque formulaire, requête auprès d'une base de données, d'un service web, etc. agrandit *la surface d'attaque* de votre service web.

Règle d'or : **NE JAMAIS FAIRE CONFIANCE AU CLIENT, NI À AUCUNE DONNEE VENANT DE L'EXTERIEUR**.

Toute donnée provenant de l'extérieur **doit être validée**.

Sécurisation des données en PHP

Pour sécuriser les données *entrantes* dans le site web, il existe plusieurs stratégies :

- **La validation** : la valeur soumise doit être une valeur présente dans une liste de valeurs acceptées (exemple : select). **Le cas idéal**
- **La *sanitization*/le filtrage** : altérer, modifier les données entrantes pour leur retirer tout aspect dangereux (caractères spéciaux). A ses limites
- **L'échappement** : si la donné doit être écrite quelque part, sur un document HTML, dans une base de données, le code s'assure de neutraliser tous les caractères spéciaux pour rendre la donnée inoffensive, sans la modifier

Échappement des caractères spéciaux

L'échappement est, avec la validation, le meilleur mécanisme de défense pour votre application. Contrairement au filtrage (*sanitization*), l'échappement **ne modifie pas les données entrantes** mais s'assure que, **dans un contexte donné**, cette donnée soit utilisée de manière sécurisée en *échappant* (rendant inoffensif) les caractères jugés dangereux **pour ce contexte**.

Par exemple :

- Dans un contexte de page HTML (ce qui nous intéresse ici), il faut échapper tous les caractères permettant d'injecter des balises : `<`, `>`, `/>`, etc. Pour cela, **utiliser toujours la fonction PHP `htmlentities()` quand vous générer une page HTML avec des données provenant de l'extérieur du script** (données envoyées par le client, données récupérées depuis une base de données).
- Dans un contexte de requête SQL, il faut échapper les caractères qui essayent de modifier la structure de la requête SQL: `'`, `--`, `"`, `_`, etc. Cette attaque est connue sous le nom d'injection SQL. Pour échapper ces caractères, on utilise des requêtes préparées.

On verra les abstractions que nous offre WordPress pour cela (santitize, escape)

> Recommandation/Mantra : **Don't sanitize input. Escape output.**

> Pour l'échappement, on pourra utiliser un moteur de Templates Twig

Sécurisation des données : Validation

Identifier clairement les données dans votre script PHP qui proviennent de l'*extérieur* (la base de données est aussi l'exterieur !), et sont potentiellement dangereuses pour votre service, et celles qui ont été validées.

La validation est le cas idéal et doit **être utilisée dès que possible**. Cela consiste à comparer la valeur fournie à une liste de valeurs acceptées. Si la valeur fournie n'est pas dans la liste, elle est rejetée (*liste blanche*)

```php
//Liste des valeurs acceptées
$acceptedColors = ['red', 'white','blue'];
//Si la donnée 'color' soumise est dans la liste, on l'accepte et on poursuit le traitement
if(isset($_POST['color']) && in_array($_POST['color'], $acceptedColors)){
    $clean['color'] = $_POST['color'];
}else{
    //Indiquer une erreur
    echo "Couleur invalide";
}
```

Sécurisation des données : Validation dans WordPress

WordPress nous fournit quelques fonctions utiles (`formatting.php`) pour valider les données :

```
is_email()
term_exists()
username_exists()
validate_file()
etc...
// Voir toutes les fonctions de la forme *_exists(), *_validate(), and is_*()
```

À utiliser au maximum !

Sécurisation des données : Échappement des caractères spéciaux

WordPress nous fournit un ensemble de fonctions utiles dans `formatting.php` pour échapper dans le contexte HTML (templates) :

```
esc_html__(): string //  Retrieves the translation of $text and escapes it for safe use in HTML output.
esc_html_e() : void //   Displays translated text that has been escaped for safe use in HTML output.
esc_attr__() : string // Retrieves the translation of $text and escapes it for safe use in an attribute.
esc_attr_e() : void //   Displays translated text that has been escaped for safe use in an attribute.
```

- **A utiliser** toujours !
- **À utiliser à la place de** `htmlentities()` !

> e === 'echo' pour "display".

Sanitization/Filtrage

Lorsqu'il n'est pas possible de valider des données (un nom, un prénom par exemple), i.e quand le nombre possible de réponses est impossible à déterminer à l'avance, on peut ajouter une couche de *filtrage* des données.

Filtrer (*sanitize*) les données consister **à modifier les données entrantes pour les rendre inoffensives, généralement en supprimant ou remplaçant des caractères**. Par exemple, en retirant tous les caractères spéciaux qui pourraient être utilisés pour former des balises HTML, à savoir `<`, `>` et `/>`.

Pour cela, en PHP, on peut utiliser les fonctions filter_var, filter_input, filter_input_array

Sécurisation des données: Sanitization

WordPress nous fournit tout un ensemble de fonctions utiles dans `formatting.php` *sanitize* des données :

```
sanitize_email()
sanitize_file_name()
sanitize_hex_color()
sanitize_hex_color_no_hash()
sanitize_html_class()
sanitize_key()
sanitize_meta()
sanitize_mime_type()
sanitize_option()
sanitize_sql_orderby()
sanitize_term()
sanitize_term_field()
sanitize_text_field()
sanitize_textarea_field()
sanitize_title()
sanitize_title_for_query()
sanitize_title_with_dashes()
sanitize_user()
sanitize_url()
```

Exemple sanitize_text_field

Regardons sanitize_text_field :

- Checks for invalid UTF-8,
- Converts single < characters to entities
- Strips all tags
- Removes line breaks, tabs, and extra whitespace
- Strips percent-encoded characters

En résumé

Utiliser les conventions de WordPress pour toute le code interaggisant avec le core de WordPress :

- **Préfixer** toutes les fonctions, classes, variables globales, options et hooks avec un **préfixe unique** pour éviter les collisions ;
- Suivre le style de `snake_case` pour les fonctions et hooks, `Pascal_Case` pour les classes.

Appliquer les mêmes règles dans tout votre projet (cohérence), **appliquer la plupart des règles** du *core* a des bénéfices :

- **Continuité/Homogénéité** entre votre code et les sources du *core* (*smooth transition*);
- **Mieux comprendre** le code source du *core*;
- **Conventions** utilisées (généralement...) par les autres dev WordPress;
- Penser **à sécuriser les données** (**validation**, **échappement** et sanitization) avec les primitives fournies soit par WordPress, le cas échéant par PHP.

Pratique : PHP et WordPress

> Utiliser le kit de développement et le thème de départ.

1. **Coder** la *Loop WordPress* dans le fichier `index.php` du thème pour **afficher la liste des posts**.

```
//fichier web/wp-content/themes/mon-theme/index.php
<?php get_header(); ?>

//Votre code ici...

<?php get_footer(); ?>
```

Utiliser le *template* HTML suivant pour chaque post:

```
...
<article>
    <h2>{titre}</h2>
    <p>{résumé (excerpt)}</p>
    <a href="{permalien}">Lire la suite</a>
</article>
...
```

S'il n'y pas de posts publiés, la page doit afficher `"Aucun contenu trouvé"`. Vérifier que dans `Settings/Reading` l'option `Your Home page displays` soit égale à `Your Latest Posts`.

Pratique : PHP et WordPress

2. **Réécrire** le code précédent en utilisant les *template tags* suivantes : `the_title()`, `the_exceprt()` et `the_permalink()`. Les *template tags* (vue à la prochaine séance) sont des fonctions WordPress qui permettent de récupérer ou d'afficher des contenus dynamiques, principalement dans la Loop WordPress.

3. Que se passe-t-il si, dans `Settings/Reading`, vous changez l'option `Your Home page displays` à `A static page` (choisir une page) ? Pourquoi ?
4. **Mettre** des balises HTML dans le titre d'un article. Par exemple `<em>Hello !</em>`. **Observez** le résultat sur la page d'accueil. Est-ce que le *template tag* `the_title` échappe le contenu ? **Est-ce normal** ? Consultez la documentation pour le savoir. Que faut-il en déduire ?
5. **Listez** quelques autres *template* tags permettant d'afficher ou de récupérer les données d'un post.

Pratique : PHP et WordPress

Créer un plugin de développement dédié au debug. Ce plugin servira à centraliser les outils de debug et pourra embarquer des composants installés via Composer. Il devra fournir :

- Le composant VarDumper pour utiliser les fonctions dump() et dd() ;
- Une fonction de journalisation `write_log` améliorée, plus pratique que la fonction native `error_log` :

```
if ( ! function_exists( 'write_log' ) ) {
 function write_log( $log ) {
  if ( true === WP_DEBUG ) {
   if ( is_array( $log ) || is_object( $log ) ) {
    error_log( print_r( $log, true ) );
   } else {
    error_log( $log );
   }
  }
 }
}
```

Pratique : PHP et WordPress

1. Créer un dossier `mon-plugin-debug-tools` dans le dossier `wp-content/plugins` avec la structure suivante :

```
├─ mon-plugin-debug-tools.php
└─ src/
   └─ functions.php
```

2. Dans ce dossier, **initialiser** un projet avec `composer init` ;
3. **Installer** le composant VarDumper;
4. Ajouter le code de la fonction `write_log` à `functions.php` ;
5. Regarder dans le guide comment créer un plugin. Utilisez les fonctions `register_activation_hook` et `register_deactivation_hook` ;
6. **Tester** votre plugin dans votre thème.

> C'est l'occasion de découvrir les *hooks* et le développement de plugins. Nous y reviendras plus tard dans la formation.

Annexe : *Templating* avec Twig

Optionnelle, pour aller plus loin, donner à réfléchir...

Templating avec Twig

Pour écrire des *templates* :

- Composables et réutilisables ;
- Plus simples à écrire et à lire ;
- Sécurisés par défaut (échappement automatique);

Templating avec Twig

Le principe d'un template est d'élaborer des modèles statiques de pages HTML dans lesquels viendront s'insérer des données dynamiques, résultat de l'exécution d'un programme.

Le *templating* permet de faire **la séparation entre données statiques (présentation) et données dynamiques (résultat du traitement)**.

> N'est-ce pas une définition possible de PHP et l'origine de son succès ?!

Twig

Twig (3.X) est un moteur de templates *fast* (compilé), *secure* (échappement par défaut) and *flexible* (on va voir pourquoi).

- Crée en 2008 ;
- Inspiré du moteur de templates Jinja (Python) ;
- Maintenu par la *core team* de Symfony, moteur de templates par défaut de Symfony ;
- **Utilisable dans n'importe quel projet**, comme n'importe quelle dépendance (composant à installer via Composer)

```
composer require "twig/twig:^3.0"
```

Pourquoi utiliser Twig et non PHP directement ?

PHP est **par définition** un moteur de templates. Alors, pourquoi ne pas utiliser directement PHP ?

Fabien Potencier, créateur (entre autres) du framework Symfony, a écrit un long billet intéressant à ce sujet.

Inconvénients d'utiliser *vanilla* PHP comme moteur de template :

- Trop verbeux ;
- Mauvaise lisibilité, car langage non orienté *template* ;
- Mauvaise réutilisabilité : pas de mécanisme de **réutilisation** de **fragments** de template (composants) ;
- **Sécurité : Pas d'échappement par défaut !**

Exemple Twig vs PHP : échappement des caractères spéciaux

En PHP, pour *échapper* les caractères spéciaux (et dangereux dans le contexte d'une page web !), il faut utiliser la fonction htmlspecialchars, qui convertit les caractères spéciaux en entités HTML, inoffensives.

Template en PHP :

```php
<?php echo $var ?>
<?php echo htmlspecialchars($var, ENT_QUOTES, 'UTF-8') ?>
```

Template Twig :

```twig
{# Par défaut Twig échappe les caractères spéciaux #}
{{ var }}
```

Langage orienté template : Boucles et conditions

PHP :

```
<?php if ($products): ?>
  <?php foreach ($products as $pdocut): ?>
  <?php if($product->isAvailable()):>
    * <?php echo $item ?>
  <?php endif;>
  <?php endforeach; ?>
<?php else: ?>
    No product has been found.
<?php endif; ?>
```

Twig :

```
{% for product in products | product.isAvailable %}
  * {{ product }}
{% else %}
  No product has been found.
{% endfor %}
```

Un mot sur l'échappement, sécurité côté client

Démonstration de l'échappement et des risques à oublier (souvent) de le faire.

> Don't try to sanitize input. Escape output

Langage Twig

Twig vient avec son propre langage et son système de balises

Principalement **trois systèmes** de balise à retenir:

- `{# ... #}` : un commentaire ;
- `{% ... %}` : executer une instruction ;
- `{{ ... }}` : écrire quelque chose sur la sortie standard.

```
{# Execute une instruction: block, extends, function #}
{% ... %}
{# Print quelque chose}
{{ ... }}
```

Pour en savoir plus consulter la documentation.

Étendre et réécrire un template existant: extends et block

Templating : héritage et surcharge, vers une approche composants

Twig permet d'écrire des templates de manière plus sécurisée et plus lisible que PHP.

Il permet surtout d'**étendre** et de **réécrire** une partie d'un template existant.

Vous pouvez **réutiliser** des templates (avec l'instruction `extends`) et les surcharger (en redéfinir des sections) via le système d'héritage et de `block`.

Templating : héritage et surcharge

Template parent/principal:

```
{# Fichier parent.html.twig #}
{# Définition d'un block: un espace réinscriptible dans le template #}
{% block main %}
Contenu de parent.html.twig
{% endblock %}
```

Template réutilisant le template parent:

```
{# Fichier child.html.twig #}
{% extends 'parent.html.twig' %}

{%block main %}
Ce contenu écrase le contenu du template parent
{% endblock %}
```

Ce système, simple et puissant, **permet de développer une approche composant des templates et de mutualiser les éléments communs**.

Templating : héritage et surcharge

Template parent/principal:

```
{# Fichier parent.html.twig #}
{# Définition d'un block: un espace réinscriptible dans le template #}
{% block main %}
Contenu de parent.html.twig
{% endblock %}
```

Template réutilisant le template parent **et le contenu du bloc main parent** (fonction `parent()`):

```
{# Fichier child.html.twig #}
{% extends 'parent.html.twig' %}

{%block main %}
{# Utiliser le contenu du bloc parent #}
{{parent()}}
Ajouter le contenu spécifique au template
{% endblock %}
```

Inclure des templates ou fragments dans d'autres templates

Inclure des templates avec `include` :

```
{# Fichier child.html.twig #}
{% extends 'parent.html.twig' %}
{%block main %}
  {{include ('publicite.html.twig')}}
{% endblock %}
```

> Le path des fichiers `.twig` inclus est relatif au path du dossier `templates`.

Déclarer et manipuler des variables locales au template

Déclarer et assigner des variables dans un template.

```
{% set foo = 'bar'%}
{% set bar = {'key': 'value'} %}
```

Manipuler les données avec les filtres

Twig propose tout **un mécanisme de filtres très puissant** (*built-in* et que l'on peut développer soi-même), permettant d'écrire des templates concis et efficaces pour présenter les données.

À l'instar de la philosophie des programmes Unix, les filtres utilisent le système de *pipe* (`|`), pour pouvoir se servir d'une sortie d'un filtre comme entrée d'un autre filtre.

```
{# Exemples d'utilisation de filtre: sans paramètre, avec paramètres, enchaînés via le pipe}

{{ variable | nom_du_filtre }}
{{ variable | nom_du_filtre('param1','param2') }}
{{ variable | filtre1 | filtre2 | filtre3 }}
```

Exemple d'usage de filtres

Exercice : Générer une page web qui affiche 1000 nombres aléatoires compris entre 0 et 100. Les présenter dans un template Twig, puis **afficher uniquement les nombres supérieurs à 50 et impairs**.

```
{# On a donné à notre template Tiwg le tableau contenant les nombres aléatoires sous la variable numbers #}

<h2>Random Numbers !</h2>

{% for number in numbers %}
 {{ number }}
{% endfor %}

<h2>Nombres impairs supérieurs à 50</h2>

{% for number in numbers | filter (number => number > 50 && is odd) %}
 {{ number }}
{% endfor %}
```

Fonctions

Twig propose un ensemble de fonctions.

- `path()` : retourne une URL relative
- `url():` retourne une url absolue (avec le nom de domaine)
- `form():` permet de présenter un formulaire HTML
- `dump():` print pour le debug
- `asset(path)` : permet de charger un asset (fichier CSS, JS, image, font, etc.) en fournissant son chemin
- etc.

Organiser ses templates : quelques conventions, recommandations

- Nom des templates en `snake_case`, par exemple `navbar_offre_speciale.html.twig` ;
- On place les templates réutilisés (`extends`) a la racine du dossier `templates` ;
- On place les fragments/composants (`include`) dans un dossier `template/parts` ;
- Templates spécifiques à un contrôleur: `template/nom_controleur/nom_du_template.html.twig` ;

Outils utiles ?

Pour VS Code:

- Ajouter **auto-complétion des balises HTML** dans les fichiers `.twig` : *File* > *Preferences* > *Settings* > *Emmet* > *Include Languages* : ajouter **Item**: `twig` **Value**: `html` ;
- Extension Twig Language 2, TWIG pack : syntaxe highlighting, formatting, snippets, etc.

Utiliser Twig dans une application WordPress

Timber est un composant crée pour faire l'interface entre le *core* WordPress (template hierarchy, hooks) et Twig.

- Timber est la solution standard pour intégrer Twig dans WordPress :
  - Séparation propre présentation / logique (pattern MVC léger mais adapté à WP) ;
  - Templates Twig et tous ses avantages par rapport à un template PHP !
  - Repose sur la hiérarchie de templates WordPress (*template hierarchy*), *plug-in*.

Utiliser Twig dans une application WordPress. Principe

Installer Timber comme une dépendance :

```
composer require timber/timber
```

Charger les dépendances, par exemple dans `wp-config.php` :

```
require_once __DIR__ . '/vendor/autoload.php';
```

Structure du thème :

```
mon-theme/
  functions.php
  style.css
  views/
    index.twig <= template Twig associé (map un à un)
  index.php <= template WordPress classique
```

Utiliser Twig dans une application WordPress. Principe

Initialiser Timber :

```
//Dans functions.php
<?php
use Timber\Timber;
new Timber();
```

Utiliser Twig dans une application WordPress. Principe

Template WordPress (`index.php`) :

```php
//index.php
<?php
//Contient tout le contexte ($wp_query)
//Ajouter données (logique métier) au context ici
$context = Timber::context();
Timber::render('index.twig', $context);
```

Template Twig associé (`views/index.twig`) :

```twig
<!DOCTYPE html>
<html>
  <body>
    <h1>{{ site.name }}</h1>
    {% for post in posts %}
      <article>
        <h2>{{ post.title }}</h2>
        {{ post.content }}
      </article>
    {% endfor %}
  </body>
</html>
```

Utiliser Twig dans une application WordPress, Warning

Attention, Timber n'est pas forcément la solution la plus adaptée en fonction de la nature et de l'état de votre projet :

- Projet déjà très avancé avec **beaucoup de templates PHP** (contenant de la logique) : migration coûteuse (et non sans risques) à prévoir !
- **Block Themes (Full-Site Editing)**, opposé au principe ! Retour aux thèmes classiques;
- Site utilisant de nombreux plugins ou **pilotés par un plugin dominant** (WooCommerce, Elementor, etc.);
- Equipe/projet *WordPress native*, Timber impose un minimum de discipline et de nouvelles contraintes.
- **Deux dépendances supplémentaires** à gérer !

Timber/Twig est **idéal pour les thèmes custom maîtrisés**, où l'on **contrôle le markup et l'architecture**.

> Il est toujours possible d'utiliser Timber/Twig. Seulement il faut mesurer le coût/bénéfices en fonction du projet. Si le projet repose sur un thème existant, FSE, un page builder, ou un plugin imposant son rendu, la bascule vers Twig a un coût élevé et doit être pesée soigneusement.

Ressources

Liste de ressources utiles pour PHP et pour continuer à se former.

Général

- Manuel PHP, la documentation officielle de PHP **traduite** en français, plutôt **complète** (explications, exemples)
- Dépôt dédié à la veille PHP, WordPress encourage l'utilisation des dernières versions de PHP. Utile de **rester à jour**. Liste articles et **actualités**, **tooling**, **conférences**, communautés, etc.
- Nouveautés de PHP 8.4
- Nouveautés de PHP 8.5 (11/25), non publiée au moment de la publication de ce support

Namespaces et Auto-loading

- Auto-chargement de classes, chargement automatique de classes et d'implémentations à l'exécution
- Documentation PHP sur les namespaces
- Documentation PHP sur l'autoloading
- PSR-4: autoloading, le standard défini par le PHP-FIG

Composants

- Composer, site officiel du gestionnaire de dépendances de PHP
- Packagist, dépôt principal de composants PHP
- Awesome PHP: un dépôt qui maintient une liste filtrée de *bons* composants

Coding standards, style guides

- PSR-1: Basic Coding Standards
- PSR-12: Extended Coding Style [en anglais], les standards avancés de la communauté PHP, défini par le PHP-FIG (Framework Interoperability Group)

Design (POO)

- Solid Principles, *Uncle Bob* [en anglais], Robert C. Martin, Uncle Bob, discute des principes SOLID notamment de l'architecture plugin et du polymorphisme
- Chapitre *How Design Patterns Solve Design Problem* de *Design Patterns: Elements of Reusable Object-Oriented Softwares*,p24-36 [en anglais]. Cette section est très bien faite et décrit ce qu'est une interface (sous-section *Specifying Object Interfaces*) ainsi que les fondamentaux de la programmation orientée objet.
- Design Patterns: Elements of Reusable Object-Oriented Software, la source
- Le catalogue des patrons de conception (de Refactoring guru), bon catalogue en ligne interactif avec des exemples de code

Templating, échappement et sécurisation des données

- Site officiel du moteur de templates Twig
- Don’t try to sanitize input. Escape output., lecture recommandée sur les bonnes stratégies à employer pour sécuriser un contexte web;
- htmlspecialchars(), fonction native de PHP. **À utiliser pour échapper les caractères spéciaux** dans un template de page HTML;

WordPress et PHP

Ouvrages

- *Modern PHP*, Josh Lockhart publié chez O'REILLY (2015)