Module 03 - PHP Moderne (8+)

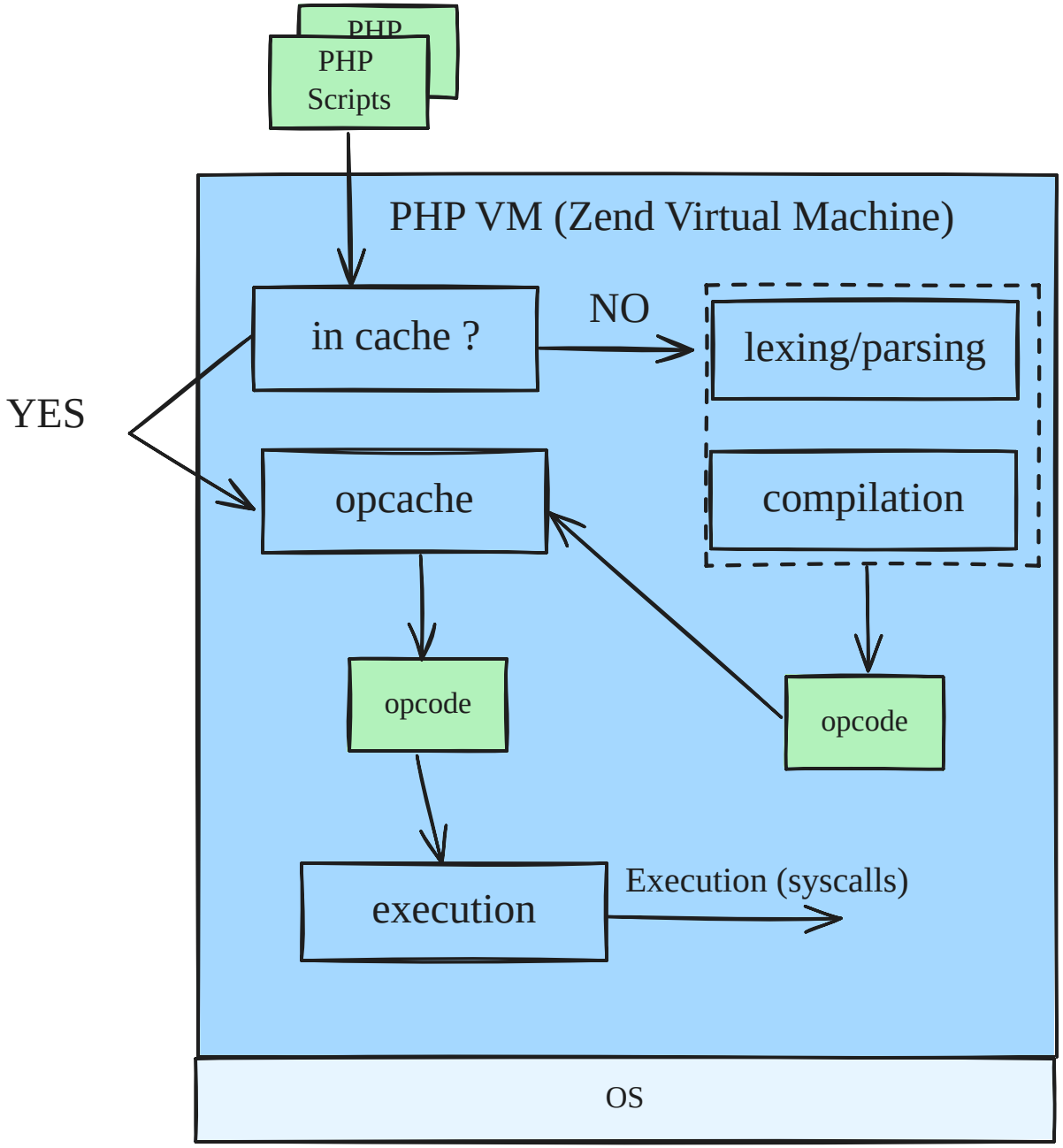
Développement **moderne** avec PHP.

PHP

Revoir quelques bases *à la volée* :

- Fonctionnement de PHP (VM);
- Types, opérateurs, variables;
- Fonctions;
- Tableaux;
- Classes;
- Etc.;

Fonctionnement de PHP



PHP VM (Sans JIT)

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 de déclarer les types des arguments et retours de fonction (*type hinting*). Cela assure que la valeur **à l'exécution** est du type attendu, sinon une erreur `TypeError` est lancée.

```php
//Quelques exemples de type hinting
interface AnimalShelter
{
    //Méthode attend un argument de type string et retourne une valeur de type Animal
    public function adopt(string $name): Animal;
}

//Méthode attend 2 entiers et retourne un entier
function sum(int $a, int $b): int {
    return $a + $b;
}
```

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);`

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

La morale de cette histoire est :

- **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** (pas de la déclaration);

PHP 8.4

> Dernière version actuelle : 8.4.2

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.

Nouveautés de PHP 8.4

- Les nouvelles fonctions array_find(), array_find_key(), array_any() et array_all() sont désormais disponibles.
- Lazy Objects;
- Nouvelle méthodes pour les classes DateTime et DateTimeImmutable (API simplifiée et homogénéisée);
- Syntaxe: `new MyClass()->method()` sans parenthèses.
- Fonctions et API obsolètes;
- Etc.;

> Voir toutes les nouveautés de PHP 8.4

Namespaces (Espaces de noms)

- 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 distributeur de code le distribue 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

`Baz.php`

```php
<?php
namespace Foo

class Baz{}
```

`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` != `require`

Foo.php

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

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 par défaut, le namespace global.

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 `\`.

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"
```

TD: Namespacer une démo (ensemble de scripts PHP) faites ensemble

Prenez le code source de la démo `FactoryPattern` et refactoriser le pour le placer dans des espaces de noms.

1. Placer les implémentations dans un dossier `src` ;
2. Placer les interfaces dans un dossier `src/Interfaces` ;
3. Créer un namespace `MDS\DemoNamespace` et placez les implémentations et les interfaces dans les sous namespaces correspondant;
4. **Executer le programme** dans le code client `index.php` en utilisant le mot clef `use` ;

> Laisser les `require` en place ! Nous n'avons pas encore mis en place l'*autoloading* !

Gérer les dépendances entre fichiers

Les limites de l'import explicite

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

Problème avec l'importation explicite:

- Se met mal à l'échelle quand le nombre de scripts augmente;
- Pollue le code source, illisible;
- Rend la refactorisation du code extrêmement pénible (changer le *path* de chaque fichier importé);

Solution : l'autoloading

L'*autoloading* définit une stratégie pour trouver une classe, une interface et la charger dans l'interpréteur PHP à la demande, au moment de l'execution.

Un autoloader permet de charger des classes *sans devoir* ***explicitement*** *dire quels fichiers inclure* (avec `require`, `require_once`, `include` ou `include_once`)

L'autoloading avant: `__autoload()`

PHP avait introduit depuis PHP 5 la fonction magique __autoload(). Elle est devenue obselète 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 pouvez 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.

> 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 organisation va avoir sa propre stratégie. **Vous devez comprendre** comment chaque composant ou framework implémente son autoloading.

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 définit une stratégie pour trouver et charger les classes, interfaces à l’exécution, **basée sur les namespaces**.

Cette recommandation impose une contrainte sur :

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

Le but est qu'un `use` puisse être associé à un `require` automatiquement ! 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 à un répertoire.

> Par exemple, `\MDS\B3\MVC` correspond à mon répertoire `src`.

2. A présent PHP sait que toute classe ou interface déclarée dans le namespace `\MDS\B3\MVC` se trouve dans le dossier `src`
3. **Organisation de votre code:** Faites *correspondre les sous-namespace aux sous-dossiers*.

> Par exemple `\MDS\B3\MVC\ModernPhp` correspond au dossier `src/ModernPhp`.
> La classe `\MDS\B3\MVC\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. Nous y reviendrons. L'autoloader PSR-4 utilise sous le capôt la fonction `spl_autoload_register()`

Composer, le gestionnaire de paquets moderne de PHP

Composer

- un gestionnaire de composants (de dépendances) **puissant** et **simple** à utiliser
- gestion de l'autoloading (PSR-4)

Packagist

Le dépôt où sont publiés des composants PHP.

> A ne pas confondre avec PECL, le dépôt des extensions du core de PHP

Vous dites à Composer quel composant vous voulez et Composer le télécharge et l'autoload dans votre projet ! Il 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 dép du projet en respectant les versions déclarées dans le `composer.lock`

`composer update` : télécharger les dernières versions des composants, 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;
- 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 !);

TD (suite): transformer la démo en composant avec l'autoloading

Transformer le code source de la démo pour qu'il puisse utiliser l'*autoloader* PSR-4.

1. Retirer tous les `require`, `require_once`, `include`, `include_once` des sources;
2. Placer tout le code source (sauf le `index.php` qui ne fait pas partie du paquet mais sert de point d'entrée) dans un dossier `src`;
3. A la racine du projet, exécutez `composer init` pour créer un fichier `package.json` et un autoloader. Suivez les instructions. Utiliser comme *vendor name* ; `mds`, et comme *package name* `monpackage`;
4. Charger l'*autoloader* avec `require 'vendor/autoloader.php'` dans `index.php`;
5. Exécutez `php index.php` ... *et voilà !*

L'autoloading de fonctions

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

Si l'on ne souhaite pas utiliser de classes, il est possible d'autoloader de simple fonctions. Utiliser la clef `files` de l'`autoload` dans `composer.json` :

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

Remarques sur le versionnement

Pensez à

- Versionner vos fichiers `composer.json` et `composer.lock`;
- Ajouter le répertoire `vendor` à votre `.gitignore` (ignorer).

> 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

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`

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 très intéressantes et faciles à versionner, maintenir et distribuer;
- Écosystème PHP riche et bien vivant ! (413 404 paquets sur Packagist au 05/24);

`PHP_CodeSniffer` et coding standards

PHP_CodeSniffer : 2 scripts

- `phpcs` : detecte 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*)

Outil essentiel de développement pour assurer un code propre et cohérent, pour le versionnement (même règles pour toute l'équipe)

Installer `PHP_CodeSniffer` (via `Composer`)

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

Plusieurs méthodes d'installation

Via Composer

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

ou en tant que dépendance de dev uniquement

```
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

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
```

Voir plus d'options ici.

Test

`src/index.php`

```php
<?php

echo "Hello World!";
class foo{};
?>
```

```
phpcs --standard=PSR12 src
```

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

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

Utilisation dans VS Code

Installer l'extension PHP Sniffer & Beautifier

> Les extensions permettent fournissent une interface dans VS Code à `phpcs` et `phpcbf`, elles ne les installent pas !

Installer les coding standards de Symfony.

Choisir `phpcbf` comme formateur par défaut.

Suivre le guide fourni

Utiliser le serveur local intégré de PHP

PHP dispose d'un serveur local *built-in* !

Très pratique pour le dev local (pas que pour PHP, servir des fichiers HTML/CSS statiques etc.)

Servir le dossier courant sur un serveur local HTTP qui écoute sur le port `5001`

```
php -S localhost:5001
```

Servir le dossier `src`, avec un `php.ini` local

```
php -S localhost:5001 -t src -c php.ini
```

> Pour en savoir plus, `man php` chercher la chaine `-S`. Quelques options pour le serveur intégré.
> Pour arrêter le serveur: `Ctrl+C`
> Ne pas utiliser en production !

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 -l6 src
```

En résumé

A présent vous êtes bien armé·e pour faire du PHP moderne en conditions réelles !

Vous savez :

- Utiliser le serveur intégré de PHP (en cas de doute: `man php`) pour développer et tester son projet en local;
- Utiliser `PHP_CodeSniffer` pour garder des projets maintenables, propres et travailler en équipe;
- Intégrer `PHP_CodeSniffer` à VS Code;
- Installer et utiliser des standards propres à un framework (ici Symfony);
- Installer et utiliser PHPStan pour analyser statiquement votre code avant de le commit;

Ressources

- Une introduction à PHP, tutoriel en français du site officiel de PHP. Le suivre et suivre les liens. Explorer.
- PHP : Le tutoriel pour grands débutants pressés, chapitres 1 à 5.
- Nouveautés de PHP 8.4
- Auto-chargement de classes, chargement automatique de classes et d'implémentations à l'exécution
- Documentation PHP (fr) sur les namespace
- Documentation PHP (fr) sur l'autoloading

Ressources

- PHP-FIG (Framework Interoperability Group)
- PSR-1: Basic Coding Standards
- PSR-4: autoloading
- PSR-12: Extended Coding Style [en anglais], les standards avancés de la communauté PHP, défini par le PHP-FIG (Framework Interoperability Group)
- Symfony Coding Standards
- 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
- *Modern PHP*, Josh Lockhart publié chez O'REILLY (2015)