

1/ Comprendre l'espace de nom avec PHP



Un namespace (ou *espace de nom*) est une façon de simplifier et organiser l'accès aux classes dans un projet. Avec un code classique, nous devons charger la classe, puis l'appeler comme ceci :

- Exemple de classe nommée **FirstClass.php** :

```
<?php
class FirstClass {
    public function hello($msg) : string {
        return $msg;
    }
}
```

- Exemple d'un fichier **index.php** :

```
<?php
require 'FirstClass.php';
$instance = new FirstClass();
echo $instance->hello('bonjour');
?>
```

Si nous avons dans notre projet deux classes nommées **FirstClass**, il y aura un conflit, et donc une erreur fatale va s'afficher sur PHP. Pour remédier à ceci nous utiliserons l'espace de nom déclaré **namespace**. Ceci afin d'organiser et identifier correctement chacune de nos classes. Un exemple :

- Le même fichier **FirstClass.php**, l'ensemble de la classe seront enveloppés dans un "champ" nommé **App** :

```
<?php
namespace App;
class FirstClass {
    public function hello($msg) : string {
        return $msg;
    }
}
```

REMARQUE : la déclaration **namespace** se fait toujours au début du script.

- Sur le fichier **index.php**, on ajoute le préfixe **App** qui englobe la classe **FirstClass()**

```
<?php
require ' FirstClass.php ';
$instance = new App\FirstClass();
echo $instance->hello('bonjour');
?>
```

ATTENTION : le séparateur entre le nom d'espace et la classe est l'antislash [\]

Il n'y aura aucun changement sur le comportement du code, mais le fait d'isoler la classe **FirstClass** dans un champ **App** permet éventuellement d'en créer un autre, par exemple :

```
$instance1 = new App\FirstClass(); // une classe avec un comportement spécifique
$instance2 = new Core\FirstClass(); // une autre classe avec d'autres comportements spécifiques
```

Il y a bien deux classes ayant le même nom, elles sont classées et chargées différemment grâce à leur **namespace**.

2/ Charger automatiquement toutes les classes d'un projet

Le fait d'accumuler des classes, le nombre de `require()` augmente aussi !

Pour éviter cette surcharge, PHP intègre une fonction d'auto chargement grâce à la fonction interne [spl_autoload_register\(\)](#). Celle-ci permet d'identifier le nom de la classe lorsqu'elle est déclarée.

- Un exemple sur le fichier **index.php** :

```
<?php

require 'FirstClass.php';

spl_autoload_register(function ($className) {
    die($className);
});

$instance = new App\FirstClass();

?>
```



Testez ce code. Que constatez-vous ?

Commentez la ligne : `$instance (...)`. Que constatez-vous ?

Le but est maintenant d'améliorer le code afin de charger automatiquement toutes les classes déclarées dans notre projet :

```
<?php

/**
 * AUTOLOADER
 * La fonction spl_autoload_register va rechercher TOUS les namespaces déclarés,
 * par exemple : $instance = new App\FirstClass;
 * La variable $className sera le nom du namespace chargé, par exemple : App\FirstClass
 * Sachant le nom du namespace, si l'on a choisi des répertoires du même nom, par exemple
 * un répertoire App dans lequel un fichier FirstClass.php EXISTE alors on peut le charger
 * avec REQUIRE
 */

spl_autoload_register( function ($className) {
    //on remplace l'antislash par un slash adaptée à l'os (windows ou linux) :
    $className = str_replace('\\', DIRECTORY_SEPARATOR, $className);
    //on redirige le chemin absolu à un niveau supérieur si le fichier index.php est absent :
    $absPath = __DIR__ . '/';
    if(!file_exists($absPath.'index.php'))$absPath=dirname($absPath);
    //on charge le fichier et donc la classe associée :
    require $absPath . DIRECTORY_SEPARATOR . $className . '.php';
});
```



- Créez un nouveau dossier nommé **Setup**. Déposez dans ce dossier l'exemple de code ci-dessus. Nommez le fichier : **Setup/autoload.php**
- Créez un nouveau dossier nommé **App**. Déposez dans ce dossier le fichier **FirstClass.php**
- Modifier le fichier **index.php** →
- Testez vos modifications.
- Que constatez-vous ?

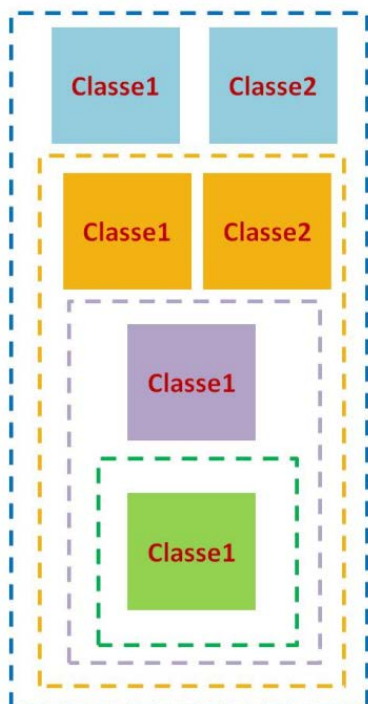
```
<?php

require __DIR__ . '/Setup/autoload.php';

$instance = new App\FirstClass();
echo $instance->hello('Salut ');

?>
```

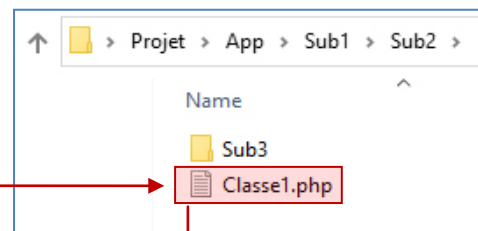
3/ Comprendre la structuration du namespace et pour l'autoloader



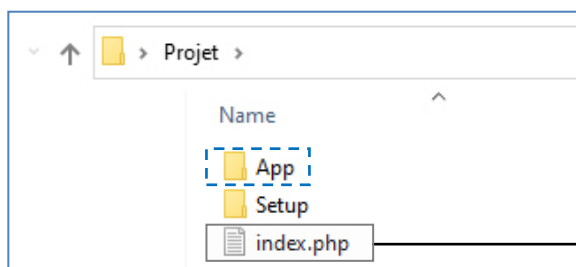
Le fait de déclarer une classe va maintenant produire un chargement du fichier PHP associé à sa classe. Dans l'exemple précédant le dossier **App** représente le namespace **App**. Si le nom du dossier change, il faudra alors modifier le nom du namespace ! Idem pour le nom de la classe, et le nom du fichier PHP (ex: *MaClasse.php*).

Ci-contre la représentation d'un namespace qui permet d'illustrer la structure d'un projet :

App\Classe1
App\Classe2
App\Sub1\Classe1
App\Sub1\Classe2
App\Sub1\Sub2\Classe1
App\Sub1\Sub2\Sub3\Classe1
(etc.)



```
<?php
namespace App\Sub1\Sub2;
class Classe1 {
    public function test() : string {
        return 'hello';
    }
}
```



```
<?php

require __DIR__ . '/Setup/autoload.php';

$instance = new App\Sub1\Sub2\Classe1();
echo $instance; //hello

?>
```

Le nom du répertoire (App) contenant les classes est arbitraire. Par ailleurs, la structure n'est pas limitée à un seul répertoire, d'autres peuvent s'implémenter à la racine du projet (ex: App, Core, System, etc...).

Exception : une classe native de PHP, tel que [DateTime\(\)](#) ou [PDO\(\)](#) dans une classe nommée va retourner une erreur si elle n'est pas placée au premier niveau (N) dans le namespace, par exemple :

```
<?php
namespace App\Sub1\Sub2;
class Classe1 {
    public function getTimestamp() : string {
        $now = new \DateTime();
        return $now->getTimestamp();
    }
}
```

Ce qu'il faut retenir : l'espace de nom est identique à la structure d'un dossier. Avec l'autoloader on doit faire coïncider le nom du namespace avec le nom du répertoire. La classe se charge automatiquement dans ce répertoire - sans *require()*, et doit avoir le même nom sur le namespace.

4/ Utilisation de USE et AS : les raccourcis et alias du namespace

Afin de raccourcir et donc utiliser plus simplement les espaces de noms, la méthode **USE** est utilisée.

- Par exemple sur un fichier **index.php** :

```
<?php
use App\Sub1\Class1;

require __DIR__ . '/Setup/autoload.php';

$instance1 = new Class1('hello');
$instance2 = new Class1('world');

?>
```

IMPORTANT : la déclaration de USE se fait toujours en début de script, après la balise PHP.

- Version **sans USE**, le namespace se répète :

```
<?php
require __DIR__ . '/Setup/autoload.php';

$instance1 = new App\Sub1\Class1('hello');
$instance2 = new App\Sub1\Class1('world');

?>
```

- L'opérateur **AS** permet de simplifier et nommer différemment le namespace. Il permet donc de créer un alias :

```
<?php
use App\Sub1\Class1 as Welcome;

require __DIR__ . '/Setup/autoload.php';

$instance1 = new Welcome('hello');
$instance2 = new Welcome('world');

?>
```

5/ Déclarations possibles du namespace

Sur PHP un namespace peut se déclarer aussi par des accolades :

```
<?php
namespace App\Personne {
    class Direction {
        public function batimentA() : array {
            return $responsable[];
        }
    }
}
```

La bonne pratique veut qu'un namespace contienne une seule et unique classe.
Cependant un namespace peut contenir plusieurs classes :

```
<?php
namespace App\Personne {
    class Direction {
        public function batimentA() : array {
            return $responsable[];
        }
    }
    class Comptable {
        public function batimentB() : array {
            return $agent[];
        }
    }
}
```

Dans ce cas, plusieurs namespace sont également possibles dans un seul et unique fichier PHP.
Exemple :

```
<?php
namespace App\Personne {
    class Formateur {
        public function batimentA() : array {
            return $referent[];
        }
    }
    class Etudiant {
        public function batimentB() : array {
            return $stagiaire[];
        }
    }
}

namespace App\Matériel {
    class Ordinateur {
        public function portable() : array {
            return $linux[];
        }
    }
    class Imprimante {
        public function consommable() : array {
            return $papier[];
        }
    }
}
```

6/ Dans la pratique ...

Reprendre les exemples de la [page 2](#) de ce document. Nous avons :

- un dossier **Setup** qui contient un fichier **autoload.php**
- un dossier **App** qui contient un fichier **FirstClass.php**
- un fichier **index.php** à la racine du projet.

À la racine :

- créez un nouveau dossier nommé **Option**.
- dans le dossier **Option**, créez un dossier **Defo**

Dans le dossier **App**

- créez un dossier nommé **Local**.

Ce qu'il faut implémenter :

- Dans le dossier **App** ajoutez une classe nommée **SecondClass.php**
- Adaptez le code source ci-dessous afin d'ajouter le namespace approprié :

```
<?php
class SecondClass {
    public function all($msg) : string {
        return $msg;
    }
}
```

- Dans le dossier **App** -> **Local** ajoutez une classe nommée **ThirdClass.php**
- Adaptez le code source ci-dessous afin d'ajouter le namespace approprié :

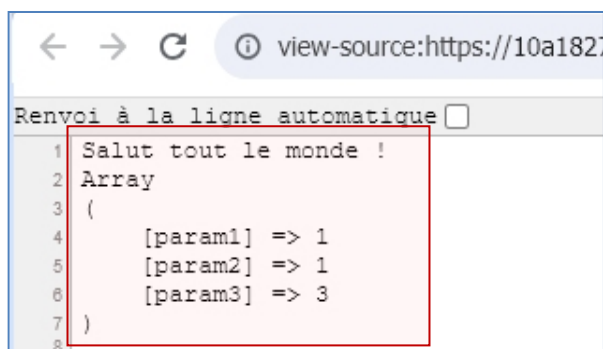
```
<?php
class ThirdClass {
    public function world($msg) : string {
        return $msg;
    }
}
```

- Dans le dossier **Option** -> **Defo** ajoutez une classe nommée **Param.php**
- Adaptez le code source ci-dessous afin d'ajouter le namespace approprié :

```
<?php
class Param {
    public function enum() : array {
        return ['param1'=>1, 'param2'=>1, 'param3'=>3];
    }
}
```

À la racine du projet, modifiez et testez le fichier **index.php**

- Depuis la classe **SecondClass()** utilisez la méthode **all** afin d'afficher la chaîne de caractère suivante : **tout le**
- Depuis la classe **ThirdClass()** utilisez la méthode **world** afin d'afficher la chaîne de caractère suivante : **monde**
- Depuis la classe **Param()** utilisez la méthode **enum** afin d'afficher le tableau avec **print_r()**
- Utilisez **USE** et **AS** dans la déclaration du namespace associé à la classe **Param()**. L'alias du namespace est : **DefoParam**



```
1 Salut tout le monde !
2 Array
3 (
4     [param1] => 1
5     [param2] => 1
6     [param3] => 3
7 )
8
```

