Chapitre 4
Écrire des fonctions et des classes PHP

1. Fonctions

1.1 Introduction

À l'instar des différents langages de développement, PHP offre la possibilité de définir ses propres fonctions (appelées fonctions "utilisateur") avec tous les avantages associés (modularité, capitalisation...). Une fonction est un ensemble d'instructions identifiées par un nom, dont l'exécution retourne une valeur et dont l'appel peut être utilisé comme opérande dans une expression. Une procédure est un ensemble d'instructions identifiées par un nom qui peut être appelé comme une instruction.

1.2 Déclaration et appel

Le mot-clé `function` permet d'introduire la définition d'une fonction.

Syntaxe

```
function nom_fonction([paramètre]) [: type]{
  instructions;
}
```

`nom_fonction` Nom de la fonction (doit respecter les règles de nommage présentées dans le chapitre Introduction à PHP - Structure de base d'une page PHP). Ce nom n'est pas sensible à la casse (pour PHP, les fonctions `unefonction` et `UneFonction` sont les mêmes).

`paramètre` Paramètres éventuels de la fonction exprimés sous forme d'une liste de variables (cf. section Fonctions - Paramètres) : `$paramètre1, $paramètre2, ...`

`type` Déclaration du type de données retourné par la fonction. Valeurs possibles : `int`, `float`, `string`, `bool`, `array`, `callable`, `iterable`, `object`, `mixed`, `void`, un nom de classe ou d'interface (cf. dans ce chapitre la section Classes), ou une union de types. Le nom du type peut être précédé d'un point d'interrogation (`?`) qui indique que la fonction peut retourner une valeur `NULL`. Voir le chapitre Les bases du langage PHP pour la définition des types de données (section Les bases du langage PHP - Types de données).

`instructions` Ensemble des instructions qui composent la fonction.

Le nom de la fonction ne doit pas être un mot réservé PHP (nom de fonction native, d'instruction) ni être égal au nom d'une autre fonction préalablement définie.

Une fonction utilisateur peut être appelée comme une fonction native de PHP : dans une affectation, dans une comparaison, etc.

Si la fonction retourne une valeur, il est possible d'utiliser l'instruction `return` pour définir la valeur de retour de la fonction.

Syntaxe

```
return expression;
```

`expression` Expression dont le résultat constitue la valeur de retour de la fonction (`NULL` par défaut).

Le résultat d'une fonction peut être de n'importe quel type (chaîne, nombre, tableau, etc.).

L'instruction `return` stoppe l'exécution de la fonction et retourne le résultat de `expression` à l'appelant. Si plusieurs instructions `return` sont présentes dans la fonction, c'est la première rencontrée dans le déroulement des instructions qui définit la valeur de retour et provoque l'interruption de la fonction. Si la fonction ne comporte aucune instruction `return` (ou si aucune instruction `return` n'est exécutée), la valeur de retour de la fonction est `NULL`.

Exemple

```
<?php
// Fonction sans paramètre qui affiche "Bonjour !"
// Pas de valeur de retour.
function afficher_bonjour() {
  echo 'Bonjour !<br />';
}
// Fonction avec deux paramètres qui retourne le produit
// des deux paramètres.
function produit($valeur1,$valeur2) {
  return $valeur1 * $valeur2;
}
// Appel de la fonction afficher_bonjour.
afficher_bonjour();
// Utilisations de la fonction produit :
// - dans une affectation
$résultat = produit(2,4);
echo "2 x 4 = $résultat<br />";
// - dans une comparaison
if (produit(10,12) > 100) {
  echo '10 x 12 est supérieur à 100.<br />';
}
?>
```

Résultat

```
Bonjour !
2 x 4 = 8
10 x 12 est supérieur à 100.
```

Remarque

Dans le langage PHP, il n'existe pas à proprement parler de procédure. Pour définir quelque chose d'équivalent à une procédure, il suffit de définir une fonction qui ne retourne pas de valeur et d'appeler la fonction comme s'il s'agissait d'une instruction (comme la fonction `afficher_bonjour` par exemple). Une fonction qui ne retourne rien peut explicitement être déclarée avec le type de retour `void`.

Comme nous l'avons déjà évoqué, le contenu d'un tableau peut être transformé en liste de paramètres dans un appel de fonction grâce à l'opérateur `...` (points de suspension).

Exemple

```
<?php
// Fonction avec trois paramètres qui retourne la somme
// des trois paramètres.
function somme($valeur1,$valeur2,$valeur3) {
  return $valeur1 + $valeur2 + $valeur3;
}
// Transformation du contenu d'un tableau en
// liste de paramètres.
$valeurs = [1,2,3];
echo '1 + 2 + 3 = ',somme(...$valeurs),'<br />';
// La même chose pour une partie seulement des paramètres
// avec un tableau défini directement dans l'appel.
echo '1 + 2 + 4 = ',somme(1,...[2,4]),'<br />';
?>
```

Résultat

```
1 + 2 + 3 = 6
1 + 2 + 4 = 7
```

Lorsqu'une fonction retourne un tableau, il est possible d'accéder directement à un élément du tableau lors de l'appel à la fonction avec une syntaxe du type `fonction(...)[clé]`.

Exemple

```
<?php
// Définition d'une fonction qui retourne un tableau.
function qui() {
  return ['Olivier','Heurtel'];
}
// Appel de la fonction et récupération directe du prénom stocké
// à l'indice 0 du tableau retourné.
$prénom = qui()[0];
echo "qui()[0] = $prénom<br />";
?>
```

Résultat

```
qui()[0] = Olivier
```

Cette technique fonctionne aussi lorsque la fonction retourne un tableau multidimensionnel avec une syntaxe du type `fonction(...)[clé1][clé2]`.

Il est possible d'utiliser une fonction avant de la définir.

Exemple

```
<?php
// Utilisation de la fonction produit.
echo produit(5,5);
// Définition de la fonction produit.
function produit($valeur1,$valeur2) {
  return $valeur1 * $valeur2;
}
?>
```

Résultat

```
25
```

Il n'y a donc aucun problème pour définir des fonctions qui s'appellent entre elles.

Remarque

Une fonction est utilisable uniquement dans le script où elle est définie. Pour l'employer dans plusieurs scripts, il faut, soit recopier sa définition dans les différents scripts (vous perdez l'intérêt de définir une fonction), soit la définir dans un fichier inclus partout où la fonction est nécessaire.

Exemple

– Fichier `fonctions.inc` contenant des définitions de fonctions :

```
<?php
// Définition de la fonction produit.
function produit($valeur1,$valeur2) {
  return $valeur1 * $valeur2;
}
?>
```

– Script utilisant les fonctions définies dans `fonctions.inc` :

```
<?php
// Inclusion du fichier contenant la définition des fonctions.
include('fonctions.inc');
// Utilisation de la fonction produit.
echo produit(5,5);
?>
```

Déclaration du type de retour

Il est possible de définir le type de données retourné par une fonction.

Lorsque c'est le cas, dans le mode de fonctionnement par défaut (à opposer au mode strict présenté ci-dessous), PHP effectue si besoin une conversion automatique de la valeur retournée dans le type de données déclaré.

Exemple

```
<?php
// Déclaration de deux fonctions qui retournent le produit
// des deux paramètres, la deuxième spécifiant un type
// de données "entier" pour la valeur de retour.
function produit1($valeur1,$valeur2) {
  return $valeur1 * $valeur2;
}
function produit2($valeur1,$valeur2) : int {
  return $valeur1 * $valeur2;
}
// Appel des deux fonctions avec les mêmes paramètres
echo 'produit1(20,1/7) => ',var_dump(produit1(20,1/7)),'<br />';
echo 'produit2(20,1/7) => <b>',var_dump(produit2(20,1/7)),'</b><br />';
?>
```

Résultat

```
produit1(20,1/7) => float(2.8571428571428568)
produit2(20,1/7) => int(2)
```

Sur cet exemple, nous voyons bien que la valeur retournée par la deuxième fonction a été convertie en entier par PHP (avec les règles de conversion évoquées dans le chapitre Introduction à PHP - Les bases du langage PHP - Types de données).

Si PHP n'est pas en mesure d'effectuer la conversion (types de données non convertibles entre eux), une exception `TypeError` est levée ; cette exception interrompt le script si elle n'est pas gérée (cf. dans ce chapitre la section Classes - Exceptions).

Exemple

```
<?php
// Déclaration et appel d'une fonction qui doit retourner un
// tableau mais qui retourne une chaîne de caractères.
function qui() : array {
  return 'Olivier Heurtel';
}
echo 'qui()[0] = ',qui()[0];
?>
```

Résultat

```
qui()[0] =
Fatal error: Uncaught TypeError: Return value of qui() must be of the
type array, string returned in /app/scripts/index.php:5 Stack trace: #0
/ app/scripts/index.php(7): qui() #1 {main} thrown in/app/scripts/
index.php on
line 5
```

Une fonction déclarée avec un type de retour autre que `void` doit retourner une valeur non `NULL`. Si ce n'est pas le cas, une erreur est retournée, différente selon les cas :

Absence d'instruction `return`

```
Fatal error: Uncaught TypeError: Return value of MaFonction() must be of
the type int, none returned in ...
```

Instruction `return` vide

```
Fatal error: A function with return type must return a value in ...
```

Instruction `return NULL`

```
Fatal error: Uncaught TypeError: Return value of MaFonction() must be of
type int, null returned in ...
```

Pour autoriser une fonction à retourner une valeur `NULL`, il faut faire précéder le nom du type (autre que `void`) d'un point d'interrogation (`?`).

```
<?php
// Déclaration et appel d'une fonction qui spécifie un
// type de donnée de retour qui peut être NULL
function cube($valeur) : ?int {
  if (is_null($valeur)) {
    return NULL;
  } else {
    return $valeur ** 3 ;
  }
}
echo 'cube(2) => <b>',var_dump(cube(2)),'</b><br />';
echo 'cube(NULL) => <b>',var_dump(cube(NULL)),'</b><br />';
?>
```

Résultat

```
cube(2) => int(8)
cube(NULL) => NULL
```

Même avec cette option, la fonction doit avoir une instruction `return` non vide. Si ce n'est pas le cas, une erreur est retournée, différente selon les cas :

Absence d'instruction `return`

```
Fatal error: Uncaught TypeError: Return value of MaFonction() must be of
type ?int, none returned in ...
```

Instruction `return` vide

```
Fatal error: A function with return type must return a value (did you mean
"return null;" instead of "return;"?) in ...
```

À l'inverse, il est possible de déclarer qu'une fonction ne retourne rien, en utilisant `void` comme nom de type. Dans ce cas, la fonction doit omettre l'instruction `return` ou mettre une instruction `return` vide, sans valeur (même pas `NULL`).

Chapitre 4
Routage et contrôleur

1. Fonctionnement du routage dans Symfony

1.1 Définition

Comme nous l'avions vu dans le chapitre sur l'architecture du framework, Symfony n'utilise pas la mise en relation entre l'URL et le système de fichiers pour servir les pages web.

Les URL sont gérées par le routage (composant **Router** du framework). Le rôle de ce composant est de trouver l'action à exécuter pour une requête donnée et pour cela, il s'appuie sur un ensemble de règles de routage définies par le développeur.

Remarque

*Si vous êtes familier des systèmes de réseaux, vous avez probablement déjà entendu parler de ce terme, voire du **routeur**. Dans un contexte applicatif, cela correspond à l'**action** sélectionnée pour une **requête** donnée ; les actions sont contenues dans des contrôleurs sous forme de méthodes.*

1.2 Le répertoire public et le contrôleur frontal

À la racine de votre projet, vous avez un répertoire nommé **public**. Il contient tous vos fichiers publics. Ce sont typiquement des images, des feuilles de style CSS, des fichiers de scripts JavaScript, et, plus largement, tous les fichiers destinés à être servis directement par le serveur web.

Pour l'URL **http://monjournal.local/robots.txt** (ou bien **http://localhost:8000/robots.txt** si vous utilisez le serveur web intégré de PHP), le fichier **robots.txt** du répertoire **public** sera servi.

Tandis que pour **http://monjournal.local/hello/world**, comme le dossier **public** ne contient pas de sous-dossier **hello** avec, à l'intérieur un fichier **world**, le contrôleur frontal est invoqué et c'est votre application qui, après avoir analysé la requête HTTP, peut décider de retourner une réponse, dont le corps serait « Hello world! ».

Pour ce faire, le Kernel fait appel au composant Router (cf. Architecture du framework - Architecture de Symfony pour un schéma explicatif).

1.3 Une requête, une action

Le rôle du routage est donc de sélectionner l'action à invoquer selon un ensemble de règles. Ces règles sont en rapport avec la requête HTTP envoyée par le client, requête qui est la seule entité permettant au routage de pouvoir faire son choix.

Les principales règles de routage sont :

– la méthode de la requête (GET, POST, etc.)

– le *path* de l'URL (http://www.website.com**/hello/world**)

– l'hôte de l'URL (http://**www.website.com**/hello/world)

Pour le *path* et l'hôte, un système similaire aux REGEX (expressions régulières) est disponible et permet de définir des URL dynamiques.

Nous allons maintenant aborder les différentes règles de routage au travers d'exemples.

2. Définition des routes

Une route est une règle de routage. Chaque route est constituée de différentes règles, et pointe vers une action donnée.

Par défaut, le fichier de routage de l'application est **config/routes.yaml**, celui-ci étant complété par les fichiers se trouvant dans **config/routes/** ; dans ce dernier, on trouve notamment le fichier **annotations.yaml** permettant d'activer la configuration du routage via des annotations :

```
controllers:
   resource: ../../src/Controller/
   type: annotation

kernel:
   resource: ../../src/Kernel.php
   type: annotation
```

2.1 Les différents formats de définition

Comme évoqué dans le chapitre Mise en place d'un projet Symfony, section Structure de l'application, il existe différentes manières de définir la configuration d'une application Symfony : via des **annotations** ou les **attributs PHP** mais également par des **fichiers de configuration**. Plusieurs formats sont disponibles pour les fichiers de configuration : **YAML** (*YAML Ain't Markup Language*), **XML** (*eXtensible Markup Language*) ou PHP.

En ce qui concerne le routage, ce sont principalement les **annotations** ou les **attributs PHP** (**si l'application fonctionne sur PHP 8**) et les fichiers au format **YAML** qui sont utilisés, les formats **XML** et **PHP** étant, de manière générale dans Symfony, de plus en plus délaissés.

Les informations obligatoires pour la définition des routes sont :

– le path : il correspond à l'URI de l'application sollicitée ;
– l'action : c'est la méthode d'un contrôleur à laquelle sera associé le path ; l'exécution sera donc déclenchée par la réception d'une requête à ce path.

Pour la configuration avec des annotations, on utilise **@Route** sur les actions à exécuter ; la valeur définie dans cette annotation est le path :

```
/**
 * @Route("/")
 */
public function index(): Response
{
    ...
}
```

Avec les attributs de PHP 8, la définition ressemble énormément à la précédente :

```
#[Route("/")]
public function index(): Response
{
    ...
}
```

Lors d'une configuration en YAML, il est nécessaire de donner un nom à la route (avec les annotations, un nom par défaut est attribué) et le nom de l'action du contrôleur, en plus du path :

```
index:
    path: /
    controller: App\Controller\DefaultController::index
```

Ici, la clé de premier niveau **index** représente le nom de la route ; le path et la spécification de l'action doivent être écrits avec un décalage de plusieurs espaces (quatre en général) par rapport au nom de la route.

En plus de ces informations obligatoires, d'autres, optionnelles, peuvent être ajoutées.

2.2 Les options sur la définition des routes

Selon le format de configuration des routes, les options complémentaires de configuration peuvent revêtir la forme de clés supplémentaires dans le fichier YAML ou bien d'attributs nommés dans l'annotation **@Route** ou l'attribut **#[Route]**. Par exemple, il est possible d'indiquer le verbe HTTP (GET, POST…) par lequel l'action sera sollicitée. Voici la définition en annotation pour limiter à une invocation en HTTP GET et POST :

```
/**
 * @Route("/", methods={"GET","POST"})
 */
public function index(): Response
{
    ...
}
```

Avec les attributs PHP :

```
#[Route("/", methods: ["GET","POST"])]
public function index(): Response
{
    ...
}
```

Et la même chose via un fichier YAML :

```
index:
    path: /
    controller: App\Controller\DefaultController::index
    methods: GET|POST
```

Les options de configuration du routage seront présentées dans la suite de ce chapitre.

3. Configurer le path

3.1 Illustration par l'exemple : /hello/world

La règle de routage la plus importante est le **path** ; elle correspond à la variable superglobale **$_SERVER['PATH_INFO']**.

Configurons une route pour un path donné : **/hello/world**.

Annotations ou attributs PHP

Selon les définitions du chapitre Architecture du framework - Le modèle de conception MVC, les actions sont des méthodes de classe appelées « contrôleur » et c'est donc au-dessus de ces actions que nous devons placer notre règle de routage sous la forme d'annotation ou d'attribut :

```
namespace App\Controller;

use
Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class WelcomeController extends AbstractController
{
   /**
    * @Route("/hello/world")
    */
   public function hello()
   {
       return new Response('Hello world!');
   }
}
```

Avec les attributs, nous utiliserions :

```
namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;

class WelcomeController extends AbstractController
```

```
{
    #[Route("/hello/world")]
    public function hello()
    {
        return new Response('Hello world!');
    }
}
```

À noter qu'avec les attributs, il n'y a pas besoin d'utiliser une quelconque directive **use** pour rendre visible une classe !

Ici, l'action **hello** sera exécutée pour toute requête dont le path est **/hello/world**, cette règle de routage étant définie grâce à l'annotation **@Route** ou l'attribut PHP.

Pour que le routage du contrôleur soit effectif, il faut qu'il soit activé dans le fichier **config/routes/annotations.yaml**. Pour nous faciliter la tâche, Symfony autorise la configuration de tous les contrôleurs d'une application en référençant un dossier en tant que ressource ; c'est d'ailleurs la configuration par défaut présente dans ce fichier :

```
controllers:
    resource: ../../src/Controller/
    type: annotation
```

Une fois les règles de routage configurées, le fait de demander l'URL http://monjournal.local/hello/world au travers de votre navigateur affiche une page dont le contenu est « Hello world! ».

Le nom des routes avec les annotations et les attributs

Nous allons voir ci-dessous avec les autres formats qu'un nom doit être assigné aux routes. Avec les annotations ou les attributs, ce n'est pas obligatoire car un nom est généré automatiquement.

Ce nom suit le format : **bundle_controleur_action**. Dans l'exemple précédent, le nom de la route est donc : **app_welcome_hello**.