Chapitre 3
Utiliser les fonctions PHP

1. Préambule

L'objectif de ce chapitre est de présenter les fonctions les plus utiles dans le cadre du développement d'un site web.

PHP propose de nombreuses fonctions ; la description de chaque fonction est accessible en ligne sur le site www.php.net.

Version 8

Depuis la **version 8**, il est possible de passer des paramètres à une fonction en utilisant le nom du paramètre au lieu de sa position. Cette fonctionnalité est présentée dans le chapitre Écrire des fonctions et des classes PHP, mais elle peut être utilisée pour les fonctions natives du langage PHP, et donc pour les fonctions présentées dans ce chapitre. Par contre, dans ce chapitre, les noms réels des paramètres des fonctions ne sont pas présentés (ils sont traduits) ; pour les connaître, consultez la documentation en ligne des fonctions.

Depuis la **version 8.1**, passer la valeur `NULL` à un paramètre qui n'est pas explicitement optionnel est déprécié et génère donc une alerte de niveau `E_DEPRECATED`.

Exemple

```
<?php
$x = null;
$n = strlen($x);
?>
```

Résultat

```
Deprecated: strlen(): Passing null to parameter #1 ($string) of type string
is deprecated in /app/scripts/index.php on line 3
```

2. Manipuler les constantes, les variables et les types de données

2.1 Constantes

PHP propose un certain nombre de fonctions utiles sur les constantes :

Nom	Rôle
`defined`	Indique si une constante est définie ou non.
`constant`	Retourne la valeur d'une constante.

defined

La fonction `defined` permet de savoir si une constante est définie ou non.

Syntaxe

```
booléen defined(chaîne nom)
```

`nom` Nom de la constante.

La fonction `defined` retourne `TRUE` si la constante est définie et `FALSE` dans le cas contraire.

Exemple

```
<?php
// Tester si la constante CONSTANTE est définie.
$ok = defined('CONSTANTE');
if ($ok) {
  echo 'CONSTANTE est définie.<br />';
} else {
  echo 'CONSTANTE n\'est pas définie.<br />';
};
// Définir la constante CONSTANTE
define('CONSTANTE','valeur de CONSTANTE');
// Tester si la constante CONSTANTE est définie.
$ok = defined('CONSTANTE');
if ($ok) {
  echo 'CONSTANTE est définie.<br />';
} else {
  echo 'CONSTANTE n\'est pas définie.<br />';
};
?>
```

Résultat

```
CONSTANTE n'est pas définie.
CONSTANTE est définie.
```

constant

La fonction `constant` retourne la valeur d'une constante dont le nom est passé en paramètre.

Syntaxe

```
mixte constant(chaîne nom)
```

Avec :

`nom` Nom de la constante.

Cette fonction est pratique pour récupérer la valeur d'une constante dont le nom n'est pas connu a priori.

Exemple

```
<?php
// définir le nom de la constante dans une variable
$nomConstante = 'AUTRE CONSTANTE';
// définir la valeur de la constante
define($nomConstante,'valeur de AUTRE CONSTANTE');
// afficher la valeur de la constante
echo $nomConstante,' = ',constant($nomConstante);
?>
```

Résultat

```
AUTRE CONSTANTE = valeur de AUTRE CONSTANTE
```

D'autres fonctions permettent de connaître le type d'une constante (cf. section Manipuler les constantes, les variables et les types de données - Types de données).

2.2 Variables

PHP propose un certain nombre de fonctions utiles sur les variables :

Nom	Rôle
`empty`	Indique si une variable est vide ou non.
`isset`	Indique si une ou plusieurs variables sont définies ou non.
`unset`	Supprime une ou plusieurs variables.
`var_dump`	Affiche des informations sur une ou plusieurs variables (type et valeur).

empty

La fonction `empty` permet de tester si une variable est vide ou non.

Syntaxe

```
booléen empty(mixte variable)
```

`variable` Variable à tester.

`empty` retourne `TRUE` si la variable est vide et `FALSE` dans le cas contraire.

Une variable est considérée comme vide si elle n'a pas été affectée ou si elle contient une chaîne vide (""), une chaîne égale à 0 ("0"), un 0, `NULL`, `FALSE` ou un tableau vide.

La fonction `empty` peut aussi être utilisée pour tester si une expression est vide ou non.

Exemple

```
<?php
// Test d'une variable non initialisée.
$est_vide = empty($variable);
echo '$variable non initialisé<br />';
if ($est_vide) {
  echo '=> $variable est vide.<br />';
} else {
  echo '=> $variable n\'est pas vide.<br />';
}
// Test d'une variable contenant une chaîne vide.
$variable = '';
$est_vide = empty($variable);
echo '$variable = \'\'<br />';
if ($est_vide) {
  echo '=> $variable est vide.<br />';
} else {
  echo '=> $variable n\'est pas vide.<br />';
}
// Test d'une variable contenant une chaîne égale à 0.
$variable = '0';
$est_vide = empty($variable);
echo '$variable = \'',$variable,'\'<br />';
if ($est_vide) {
  echo '=> $variable est vide.<br />';
} else {
  echo '=> $variable n\'est pas vide.<br />';
}
// Test d'une variable contenant 0.
$variable = 0;
$est_vide = empty($variable);
echo '$variable = ',$variable,'<br />';
```

```
if ($est_vide) {
  echo '=> $variable est vide.<br />';
} else {
  echo '=> $variable n\'est pas vide.<br />';
}
// Test d'une variable contenant une chaîne non vide.
$variable = 'x';
$est_vide = empty($variable);
echo '$variable = \'',$variable,'\'<br />';
if ($est_vide) {
  echo '=> $variable est vide.<br />';
} else {
  echo '=> $variable n\'est pas vide.<br />';
}
?>
```

Résultat

```
$variable non initialisé
=> $variable est vide.
$variable = ''
=> $variable est vide.
$variable = '0'
=> $variable est vide.
$variable = 0
=> $variable est vide.
$variable = 'x'
=> $variable n'est pas vide.
```

isset

La fonction `isset` permet de tester si une ou plusieurs variables sont définies ou non.

Syntaxe

```
booléen isset(mixte variable[, ...])
```

`variable` Variable à tester (éventuellement plusieurs, séparées par une virgule).

`isset` retourne `TRUE` si la variable est définie et `FALSE` dans le cas contraire.

Si plusieurs paramètres sont fournis, la fonction retourne `TRUE` uniquement si toutes les variables sont définies.

Une variable est considérée comme non définie si elle n'a pas été affectée ou si elle contient `NULL`. À la différence de la fonction `empty`, une variable qui contient une chaîne vide (""), une chaîne égale à 0 ("0"), un 0, un `FALSE` ou un tableau vide, n'est pas considérée comme non définie.

Exemple

```
<?php
// Test d'une variable non initialisée.
$est_définie = isset($variable);
echo '$variable non initialisé<br />';
if ($est_définie) {
  echo '=> $variable est définie.<br />';
} else {
  echo '=> $variable n\'est pas définie.<br />';
}
// Test d'une variable contenant une chaîne vide.
$variable = '';
$est_définie = isset($variable);
echo '$variable = \'\'<br />';
if ($est_définie) {
  echo '=> $variable est définie.<br />';
} else {
  echo '=> $variable n\'est pas définie.<br />';
}
// Test d'une variable contenant une chaîne égale à 0.
$variable = '0';
$est_définie = isset($variable);
echo '$variable = \'',$variable,'\'<br />';
if ($est_définie) {
  echo '=> $variable est définie.<br />';
} else {
  echo '=> $variable n\'est pas définie.<br />';
}
// Test d'une variable contenant 0.
$variable = 0;
$est_définie = isset($variable);
echo '$variable = ',$variable,'<br />';
if ($est_définie) {
  echo '=> $variable est définie.<br />';
} else {
  echo '=> $variable n\'est pas définie.<br />';
}
// Test d'une variable contenant une chaîne non vide.
$variable = 'x';
$est_définie = isset($variable);
echo '$variable = \'',$variable,'\'<br />';
if ($est_définie) {
  echo '=> $variable est définie.<br />';
} else {
  echo '=> $variable n\'est pas définie.<br />';
}
?>
```

Résultat

```
$variable non initialisé
=> $variable n'est pas définie.
$variable = ''
=> $variable est définie.
$variable = '0'
=> $variable est définie.
$variable = 0
=> $variable est définie.
$variable = 'x'
=> $variable est définie.
```

unset

La fonction `unset` permet de supprimer une ou plusieurs variables.

Syntaxe

```
unset(mixte variable[, ...])
```

`variable` Variable à supprimer (éventuellement plusieurs, séparées par une virgule).

Après suppression, la variable se trouve dans le même état que si elle n'avait jamais été affectée. L'utilisation de la fonction `isset` sur une variable supprimée retourne `FALSE` notamment.

Exemple

```
<?php
// Définir une variable.
$variable = 1;
// Afficher la variable et tester si elle est définie.
$est_définie = isset($variable);
echo '$variable = ',$variable,'<br />';
if ($est_définie) {
  echo '=> $variable est définie.<br />';
} else {
  echo '=> $variable n\'est pas définie.<br />';
}
// Supprimer la variable.
unset($variable);
// Afficher la variable et tester si elle est définie.
$est_définie = isset($variable);
echo '$variable = ',$variable??'','<br />';
if ($est_définie) {
  echo '=> $variable est définie.<br />';
} else {
  echo '=> $variable n\'est pas définie.<br />';
}
?>
```

Résultat

```
$variable = 1
=> $variable est définie.
$variable =
=> $variable n'est pas définie.
```

Remarque

Affecter un 0 ou une chaîne vide à une variable ne la supprime pas.

var_dump

La fonction `var_dump` affiche des informations sur une ou plusieurs variables (type et contenu).

Syntaxe

```
var_dump(mixte variable[, ...])
```

`variable` Variable à afficher (éventuellement plusieurs, séparées par une virgule).

La fonction `var_dump` est surtout intéressante lors des phases de mise au point.

Exemple

```
<?php
// afficher les informations sur une variable non initialisée
$variable = NULL;
var_dump($variable);
// initialiser la variable avec un nombre entier
$variable = 10;
// afficher les informations sur la variable
echo '<br />';
var_dump($variable);
// modifier la valeur (et le type) de la variable
$variable = 3.14; // nombre décimal
// afficher les informations sur la variable
echo '<br />';
var_dump($variable);
// modifier la valeur (et le type) de la variable
$variable = 'abc'; // chaîne de caractères
// afficher les informations sur la variable
echo '<br />';
var_dump($variable);
?>
```

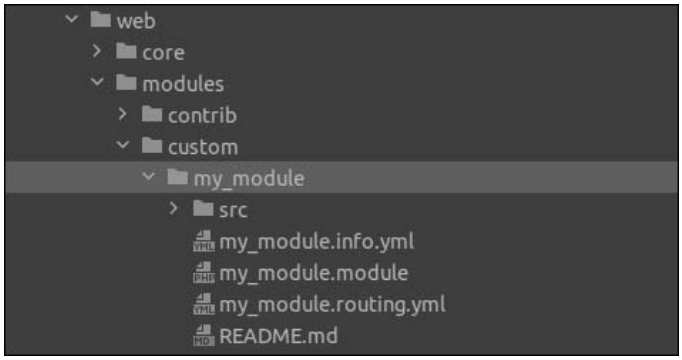
Chapitre 3
Création de modules personnalisés

1. Introduction

Les modules personnalisés jouent un rôle essentiel dans l'extension et la personnalisation des fonctionnalités de Drupal 10. Dans ce chapitre, nous explorerons les composants fondamentaux d'un module Drupal, et nous apprendrons comment créer des modules simples pour étendre les capacités de l'application.

2. Structure d'un module Drupal

Un module Drupal est composé de plusieurs fichiers clés qui définissent son fonctionnement et son comportement. Voici les principaux composants d'un module, illustrés ici à travers l'exemple du module fictif *my_module* (à noter que ce module n'est pas présent par défaut et est utilisé ici uniquement à titre d'exemple) :



Fichier my_module.info.yml

Le fichier *my_module.info.yml* est un fichier de configuration qui définit les métadonnées du module. Ces métadonnées sont essentielles pour l'intégration du module dans l'écosystème de Drupal. Dans cet exemple, « my_module » est le nom machine du module et doit être utilisé comme préfixe pour être détecté par Drupal. Voici quelques métadonnées couramment utilisées :

– `name` : le nom du module ;
– `type` : le type du module, qui est généralement défini comme « module » ;
– `description` : une brève description du module et de ses fonctionnalités ;
– `core_version_requirement` : la version de Drupal requise pour que le module fonctionne ;
– `package` : le regroupement du module, utile pour organiser les modules dans l'interface d'administration ;
– `dependencies` : les dépendances du module, telles que d'autres modules nécessaires pour son fonctionnement.

Fichier my_module.module

Le fichier *my_module.module* contient le code PHP du module. Il n'est pas obligatoire mais il peut contenir des *hooks*, des fonctions et des traitements nécessaires pour le bon fonctionnement du module. Ce fichier facultatif mais très fréquemment utilisé est automatiquement chargé par Drupal lorsque le module est activé. Dans cet exemple, « my_module » doit également être utilisé comme préfixe pour que le fichier soit détecté par Drupal.

Fichier my_module.routing.yml

Le fichier *my_module.routing.yml* définit les routes fournies par le module. Les routes définissent les URL associées à des contrôleurs et des fonctions spécifiques du module. Cela permet de créer des pages personnalisées accessibles via des URL spécifiques. Encore une fois, « my_module » doit être utilisé comme préfixe pour que le fichier soit détecté par Drupal.

Les routes peuvent contenir des paramètres qui permettent de rendre les URL dynamiques et de transmettre des informations entre les différentes parties de votre application. Voici quelques exemples de ce que vous pouvez spécifier dans une route :

– `path` : il s'agit de l'URL relative qui sera associée à la route. Par exemple, `/custom-path` pourra être utilisé pour accéder à la page associée à cette route via l'URL `https://www.my-website.com/custom-path`.
– `defaults` : cette section définit les valeurs par défaut des paramètres qui seront transmis à votre contrôleur. Par exemple, vous pouvez définir un paramètre `node` avec une valeur par défaut pour récupérer automatiquement le nœud associé à cette route.
– `requirements` : les exigences spécifient les conditions nécessaires pour que la route soit accessible. Cela peut inclure des autorisations d'accès, des contraintes de type de données, etc. Par exemple, vous pouvez restreindre l'accès à une route aux utilisateurs authentifiés uniquement. Pour ce faire, il est possible d'y associer une permission Drupal (par exemple *access content*) via l'option `_permission`, ou bien d'y associer une classe où sera effectué un contrôle d'accès spécifique via l'option `_custom_access`.

– `options` : cette section permet de spécifier des options supplémentaires pour la route, telles que la manière dont les paramètres sont convertis en chaînes de requête, la gestion des alias d'URL, etc.
– `methods` : rarement spécifiée, cette section indique les méthodes HTTP que la route acceptera. Par exemple, si une route doit uniquement répondre aux requêtes de type POST, vous indiquerez `methods: [POST]` dans la configuration de la route. Cela garantit que seules les requêtes POST seront traitées pour cette URL, les autres méthodes HTTP, comme GET ou DELETE, seront refusées. Par défaut, toutes les méthodes sont acceptées.
– `_content` : cette option vous permet de spécifier directement du contenu HTML ou du rendu d'élément dans la page associée à la route, sans nécessiter de contrôleur séparé.

Dossier src

Le dossier *src* est un élément clé dans la structure d'un module personnalisé Drupal. Ce dossier contient généralement des classes PHP qui étendent les fonctionnalités de Drupal ou qui implémentent des composants spécifiques du module. Ces classes incluent souvent des contrôleurs, des plugins, des services, et d'autres composants logiques.

Drupal effectue automatiquement la détection et la prise en charge de toutes les classes qu'il contient. Vous n'avez pas besoin de configurer manuellement chaque composant ou de déclarer explicitement sa présence.

Le dossier *src* est également le lieu privilégié pour organiser vos classes en utilisant des espaces de noms spécifiques. Les espaces de noms sont essentiels pour éviter les conflits de noms de classe et pour structurer votre code de manière logique.

En résumé, le dossier *src* est bien plus qu'une simple convention ; il représente une partie fondamentale de la structure de Drupal. En exploitant cette fonctionnalité, vous pouvez développer des modules plus efficacement, tout en profitant de la flexibilité et de l'extensibilité de Drupal pour personnaliser votre application selon vos besoins spécifiques.

3. Création d'un module simple

Créer un module personnalisé dans Drupal peut sembler complexe, mais nous allons suivre un exemple pas à pas pour vous guider à travers le processus.

Imaginons que vous vouliez créer un module qui affiche un message de bienvenue sur une page personnalisée.

3.1 Création de la structure

- Pour commencer, créez un répertoire nommé *my_module* dans le dossier *web/modules/custom* de votre installation Drupal. Ce nom est utilisé ici à titre d'exemple ; vous devrez le remplacer par le nom machine de votre propre module. À l'intérieur de ce répertoire, vous allez créer les fichiers nécessaires.

```
my_module/
|-- my_module.info.yml
|-- my_module.module
|-- my_module.routing.yml
```

3.2 Fichier .info.yml

Le fichier *.info.yml* définit les métadonnées de base de votre module. Dans cet exemple, vous nommerez votre module « My module » et spécifierez qu'il doit être compatible avec Drupal 10.

```
name: 'My module'
type: module
description: 'Define specific features for training.'
package: Custom
core_version_requirement: ^10
```

3.3 Fichier .module

Le fichier *.module* est l'endroit où vous pouvez ajouter du code PHP pour étendre les fonctionnalités de votre module. Il peut contenir des hooks, des fonctions personnalisées et d'autres traitements nécessaires pour le bon fonctionnement de votre module.

L'un des hooks couramment utilisés dans les modules Drupal est `hook_help()`. Ce hook permet de fournir une aide contextuelle pour votre module. Lorsque les utilisateurs accéderont à la page d'aide de Drupal, ils verront votre message d'aide personnalisé.

```
<?php

use Drupal\Core\Routing\RouteMatchInterface;

/**
 * Implements hook_help().
 */
function my_module_help($route_name, RouteMatchInterface
$route_match) {
  switch ($route_name) {
    case 'help.page.my_module':
      $output = '';
      $output .= '<h3>' . t('About') . '</h3>';
      $output .= '<p>' . t('This module is a custom module used
for training.') . '</p>';
      return $output;
      break;
  }
}
```

Remarque

Il est important de garder à l'esprit que des modules plus complexes nécessitent une séparation du code dans des fichiers différents pour une meilleure lisibilité et maintenabilité, ce que nous allons illustrer avec le contrôleur et le fichier .routing.

3.4 Contrôleur

Le contrôleur est l'élément clé qui gère la logique de traitement pour une page spécifique.

Dans cet exemple, vous allez créer un contrôleur qui affiche simplement un message de bienvenue.

- Créez un fichier *WelcomeController.php* dans le répertoire *my_module/src/Controller*.

```
<?php

namespace Drupal\my_module\Controller;

use Drupal\Core\Controller\ControllerBase;

class WelcomeController extends ControllerBase {

  public function build(): array {
    $build['content'] = [
      '#type' => 'html_tag',
      '#tag' => 'p',
      '#value' => $this->t('Welcome to my website!'),
    ];

    return $build;
  }

}
```

3.5 Fichier .routing

Le fichier *.routing.yml* définit les routes de votre module. Vous allez créer une route qui pointe vers votre contrôleur.

```
my_module.welcome:
  path: '/welcome'
  defaults:
    _title: 'Welcome'
    _controller:
'\Drupal\my_module\Controller\WelcomeController::build'
```

```
  requirements:
    _permission: 'access content'
```

3.6 Reconstruire le cache

Pour que votre module soit détecté par Drupal ou que vos dernières modifications soient prises en compte, il vous faut généralement reconstruire le cache de l'application Drupal.

Pour ce faire, deux méthodes sont disponibles.

Drush (recommandé)

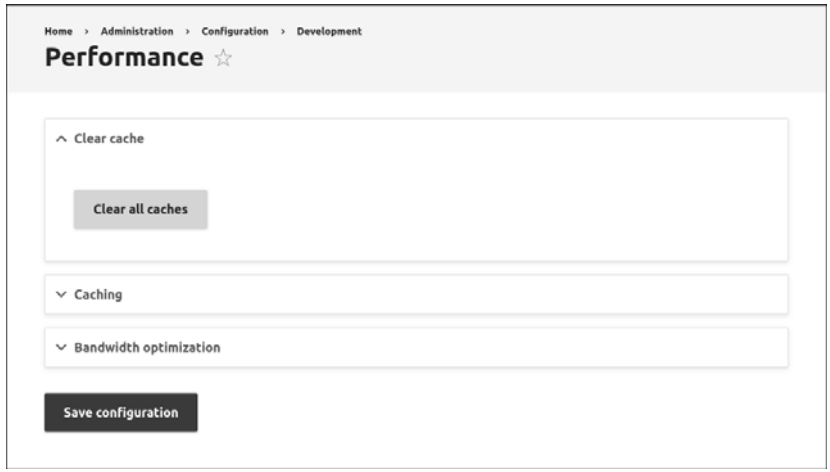
Depuis le terminal, entrez l'une des commandes suivantes :

```
# Verbose Command :
$ drush cache:rebuild
# Alias :
$ drush cr
```

```
nicolas@eni-web:/var/www/html$ drush cr
 [success] Cache rebuild complete.
nicolas@eni-web:/var/www/html$
```

UI Drupal

- Connectez-vous en tant qu'administrateur, puis rendez-vous sur la page d'administration des performances (/admin/config/development/performance).
- Cliquez ensuite sur **Clear all caches**.



3.7 Activation du module

Pour que le module soit pris en compte par Drupal, il est nécessaire de l'activer.

Pour ce faire, deux méthodes sont disponibles.

Drush (recommandé)

- Depuis le terminal, entrez l'une des commandes suivantes :

```
# Verbose Command :
$ drush pm:enable my_module
# Alias :
$ drush en my_module
```

UI Drupal

- Connectez-vous en tant qu'administrateur, puis rendez-vous sur la page listant les extensions (/admin/modules).
- Activez **My module** dans la section **Custom**, puis cliquez sur **Install** en bas de page.