Développer pour WordPress : wpdb et les requêtes SQL

Voici le premier article d'une série destinée à présenter les bases du développement dédié à WordPress, nous allons entrer directement dans le vif du sujet avec l'utilisation de la base de données à l'aide de la classe wpdb.

Qu'il s'agisse de thèmes ou de modules, les concepteurs de WordPress ont prévu la possibilité d'interfacer nos développements avec la base de données utilisée par le CMS en implémentant une classe, nommée **wpdb**, qui contient de nombreuses méthodes pour manipuler des données en toute sécurité.

La variable globale $wpdb

Pour nous simplifier un peu plus le travail, la **classe wpdb est toujours instanciée** au chargement de l'application et stockée dans une **variable globale** appelée **$wpdb**. Pour l'utiliser, à n'importe quel endroit de votre code, il vous suffit d'utiliser l'instruction suivante

```php
<?php
global $wpdb;
?>
```

Pour des raisons de performance et de sécurité, **vous ne devez pas utiliser un autre moyen de communication avec la base de données que l'objet $wpdb**.

Avant de passer aux requêtes, vous devez savoir que la variable $wpdb possède une propriété très utile pour accéder à vos tables, il s'agit de ***prefix***.

$wpdb->prefix

Si vous avez suivi les conseils proposés dans cet article (au hasard) pour **sécuriser votre installation** de WordPress, le **préfixe de vos tables** ne sera pas *wp_* et sera à coup sur différent du préfixe des tables d'un autre site. Sans précautions votre développement ne sera donc utilisable que sur votre configuration spécifique. C'est là que la propriété *prefix* intervient. Elle vous permet de passer outre cette personnalisation et de **rendre votre code adapté à toutes les situations**. Si vous utilisez la ligne suivante :

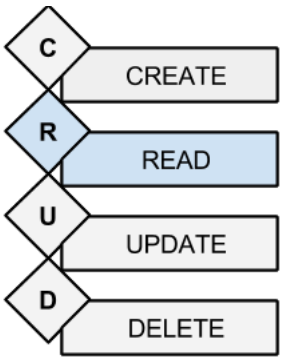
```php
$query = "SELECT * FROM {$wpdb->prefix}comments" ;
```

vous obtiendrez automatiquement le bon nom de table, à savoir wp_comments dans une installation standard.

Sachez qu'il est possible d'obtenir le même résultat en utilisant le nom de la table de votre choix comme une fonction sur $wpdb. La commande suivante aura un résultat identique à la précédente :

```php
$query = "SELECT * FROM $wpdb->comments" ;
```

Le choix entre les deux méthodes est une question de préférence, dans cet article nous utiliserons *prefix* dans toutes les requêtes.



Les requêtes d'interrogation

$wpdb->get_results()

Commençons par la requête la plus large avec ***get_results***. Cette fonction permet d'**obtenir un jeu de plusieurs lignes contenant plusieurs colonnes**, par défaut sous la forme d'un tableau d'objets ou sous la forme d'un tableau de tableaux associatifs ou non, en fonction de l'argument *output_type* que vous aurez éventuellement spécifié.

La signature de la fonction est :

```
mixed get_results(string $requete [, int $output_type]);
```

Voici un exemple d'utilisation de *get_results* : nous demandons à WordPress d'extraire tous les articles de la base de données et pour chaque article de la sélection nous affichons son titre (et un retour à la ligne) :

```php
// Interrogation de la base de données
$resultats = $wpdb->get_results("SELECT * FROM {$wpdb->prefix}posts") ;
// Parcours des resultats obtenus
foreach ($resultats as $post) {
 echo $post->post_title ;
 echo '<br/>' ;
}
```

Avant de passer à la suite, attardons nous un instant sur l'argument *output_type*. Les valeurs possible pour cet argument sont :

- **OBJECT :** *get_results* vous retournera un tableau indexé d'objets;
- **ARRAY_A :** *get_results* vous retournera un tableau indexé de tableaux associatifs;
- **ARRAY_N :** *get_results* vous retournera un tableau indexé de tableaux indexés. Attention à bien définir les colonnes que vous souhaitez utiliser dans votre requête afin d'éviter les surprises qu'un select * pourrait apporter en cas de mise à jour de la table lue;
- **OBJECT_K :** la fonction vous retournera un tableau associatif d'objets en prenant les valeurs de la première colonne comme clé.

Les trois premières possibilités sont assez simples à comprendre, transposées à notre exemple cela donnerait :

```php
foreach ($resultats as $post) {
  echo $post->post_title; // OBJECT
  echo $post['post_title'];// ARRAY_A
  echo $post[5]; // ARRAY_N
}
```

Pour la constante OBJECT_K c'est peut-être un peu plus compliqué. Imaginons que j'ai une table wp_ma_table pour suivre quelles activités ont été faites par des vacanciers dans un club de vacances. Cette table comporte les champs suivants : le nom du vacancier, l'activité réalisée, sa date et une durée. Mes données sont les suivantes :

```
Id |Nom     |Activite  |Date       | Duree
---+------+----------+-----------+-------
0  |Pierre  |Snowboard |24/12/2014 | 4
1  |Pierre  |Snowboard |25/12/2014 | 3
2  |Paul    |Snowboard |24/12/2014 | 4
3  |Paul    |Ski       |25/12/2014 | 5
4  |Jacques |Snowboard |24/12/2014 | 4
5  [Jacques |Ski       |25/12/2014 | 4
```

Si je veux savoir qui a fait quelle activité, je peux utiliser la requête

```
1 | $resultats = $wpdb->get_results("SELECT distinct(nom) FROM {$wpdb->prefix}ma_table WHERE activite = 'snowboard'",
```

et WordPress va me retourner un tableau ayant la forme suivante :

```
Array
(
    [Pierre] => stdClass Object
        (
            [nom] => Pierre
        )

    [Paul] => stdClass Object
        (
            [nom] => Paul
        )

    [Jacques] => stdClass Object
        (
            [nom] => Jacques
        )

)
```

$wpdb->get_row()

Parfois on sait qu'on n'aura qu'**une seule ligne de résultats**. Dans ce cas il est préférable en terme de **performances** de demander à WordPress de nous retourner cette ligne au lieu d'un tableau, en utilisant la fonction *get_row* à la place de *get_results*.

Ici aussi il est possible d'obtenir au choix un objet, un tableau associatif, ou un tableau indexé numériquement grâce à l'argument « **output_type**« . **Attention**, la valeur *OBJECT_K* n'est pas autorisée ici.

Il est aussi possible de définir un *offset* nous permettant de lire une ligne en particulier dans un grand jeu de résultats, par exemple le 5ème article écrit par un auteur précis.

La signature de la fonction est donc :

```
mixed get_row(string $requete [, int $output_type [, int $offset]]);
```

Dans l'exemple ci-dessous nous allons récupérer le titre et la date de création d'un article en particulier et les afficher :

```
1 | $resultat = $wpdb->get_row("SELECT post_title, post_date FROM {$wpdb->prefix}posts WHERE ID = 1337" ) ;
2 | echo $resultat->post_title . ' : ' . $resultat->post_date;
```

$wpdb->get_var()

Les pros du *mysql_result* ne sont pas en reste puisque WordPress nous offre sa fonction jumelle, à savoir *get_var*, qui permet de retourner qu'**une seule cellule** d'une requête.

Il est possible de spécifier un **offset de colonne** et un **offset de ligne** si la requête retourne plus d'un enregistrement unique et qu'on veut extraire par exemple la date de création d'un article précis. Pourquoi faire cela ? Imaginez le code suivant :

```
1 | $requete = "SELECT post_title, post_date FROM {$wpdb->prefix}posts WHERE post_author = 1337 ORDER BY post_title DE
2 | $dateMax = get_var(requete, 1);
3 | if ('2014-12-25' == $dateMax) {
4 |     $tousLesArticles = get_results($requete);
5 |     foreach($tousLesArticles as $article) {
6 |         echo $article->post_title . ' : '. $article->post_date;
7 |     }
8 | }
```

J'ai cherché à savoir si la dernière publication de mon auteur est datée du 25 décembre 2014 et si c'est le cas j'affiche la liste de toutes ses publications. L'intérêt de procéder de cette façon est que WordPress aura **mis en cache la totalité des résultats** de ma requête à l'appel à *get_var*, il n'aura pas besoin de rééxécuter la requête pour faire le *get_results*. Evidement dans cet exemple il faut que la probabilité que mon utilisateur ait publié son dernier article le jour de noël soit grande pour justi er la mise en cache de tout le jeu de données, mais cette façon de faire peut tout à fait se justier.

Je n'oublie pas la signature de la fonction, la voici :

get_var(string $requete [, int $offest_colonne = 0 [, int $offset_ligne = 0]])

Passer des paramètres à une requête en toute sécurité

Vous avez certainement bondi de vos chaises en voyant les requêtes ci-dessus, si ça n'est pas le cas vous auriez dû. Pour **protéger votre code** des actions malicieuses vous devez utiliser des **requêtes préparées**. Voyons ce que donnerait le code suivant :

```
1 | $nom =$_GET['id'];
2 | $resultats = $wpdb->get_results("SELECT * FROM {$wpdb->prefix}ma_table WHERE id = $id" );
```

Est-ce que ce code fonctionne ? Oui. Est-ce un type de code qu'on rencontre souvent ? Oui.

Est-ce une catastrophe ? Oui.

Imaginons que mon *$_GET['id']* soit *4*, tout va bien se passer, c'est ce que le développeur a imaginé. La requête qui sera exécutée sera *SELECT * FROM {$wpdb->prefix}ma_table WHERE id = 4*.

Mais imaginons un instant que la valeur de *$_GET['id']* soit *-1; SELECT * FROM wp_users*. Dans ce cas le code exécuté sera bien différent et dans le meilleur des cas l'internaute mal intentionné aura accès au nom de ma table, dans le pire des cas il aura accès à la liste des utilisateurs de mon sites. Dans tous les cas il aura accès à des informations qu'il ne doit pas connaître et il saura en plus que je code comme une quiche...

Il va donc falloir penser à sécuriser un peu les données qui sont envoyées, à l'aide de la fonction *prepare* qui va se charger d'assurer que **les données envoyées sont bien du type attendu** et d'**échapper les chaînes de caractère** afin d'éviter des injections SQL comme ci-dessus. L'idée est de remplacer dans la requête les valeurs par des **placeholders** qui vont indiquer au choix si la donnée est de type *chaîne de caractères* (%s), *nombre entier* (%d) ou *nombre à virgule flottante* (%f), et d'indiquer ensuite quelles seront les valeurs correspondantes, les unes après les autres. C'est cette requête préparée qui est exécutée.

Attention le caractère «%» devient problématique, il faut donc l'échapper en le remplaçant par «%%».

L'exemple précédent deviendrait alors

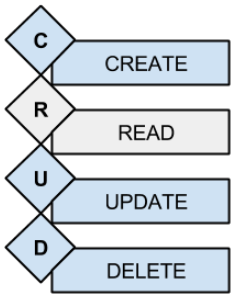
```
1 | $id = $_GET['id'];
2 | $resultats = $wpdb->get_results(
3 |     $wpdb->prepare(
4 |         "SELECT * FROM {$wpdb->prefix}ma_table WHERE id = %d",
5 |         $id
6 |     )
7 | );
```

Vous pouvez tester l'injection proposée avec ce code, votre script ne sera plus aussi bavard;

Depuis WordPress 3.5 il est obligatoire de stipuler au moins deux paramètres : la requête et une valeur. Vous ne pouvez pas utiliser prépare si vous n'avez pas de donnée à manipuler.

La signature de la fonction est :

```
string prepare(string $requete, string|int|float $valeur1 [, string|int|float $valeur2 ...]);
```



Les requêtes de manipulation de données

Fi des interrogations, nous allons maintenant ajouter, supprimer et mettre à jour nos données.

$wpdb->insert()

Pour simplifier l'**enregistrement de données** dans la base, WordPress propose une fonction bien pratique, nommée *insert* (jusque là, facile) qui permet d'**injecter un tableau directement dans une table**. Commençons par l'exemple avant les explications, en reprenant wp_ma_table, je souhaite enregistrer le fait que Laetitia ait fait 4 heures de ski le 25 décembre elle aussi. Je vais utiliser le code suivant :

```
$wpdb->insert(
    $wpdb->prefix.'ma_table',
    array(
        'nom' => 'Laetitia',
        'activite' => 'Ski',
        'date' => '2014-12-25',
        'duree' => 4
    ),
    array(
        '%s',
        '%s',
        '%s',
        '%d'
    )
);
```

J'ai demandé à l'objet $wpdb de d'insérer dans la table wp_ma_table une ligne contenant l'information représentée par le tableau. Vous pouvez remarquer que les valeurs des 3 premiers champs sont des chaînes de caractères alors que la durée est un nombre entier, il n'est pas entouré par des guillemets simples.

Pour lever toute ambiguïté quant au type de données à écrire, insert propose un troisième paramètre optionnel qui lui indiquant ce ou ces types, sous la forme d'une chaîne (%s pour un champ string, %d pour un entier, %f pour un float) ou d'un tableau de chaînes. Dans le premier cas tous les champs seront considérés comme étant du format unique, dans le second cas, chaque champ doit avoir un type déclaré dans le tableau. Si le paramètre est omis, tous les champs seront traités comme des string.

La signature de la fonction est donc

```
mixed insert(string $table [, array $valeurs [, array|string $types = '%s']]);
```

Si l'écriture a réussi ***insert* retourne le nombre de lignes affectées par l'opération** (donc 1), sinon elle renvoie *false*.

$wpdb->update()

Pour **mettre à jour une information**, le mécanisme est un peu le même : on **indique** à WordPress **le nom de la table** sur laquelle on souhaite travailler, on passe **les informations dans un tableau**, ainsi que les facteurs discriminants (**la clause *where***) et les formats possibles pour les valeurs et les conditions. Seuls le nom de la table et le tableau des nouvelles colonnes sont obligatoires.

```
mixed update(string $table, array $valeurs [, array $where [, array|string $format_valeurs [, array|string $format_where]]]);
```

En cas d'erreur la fonction update retourne *false*. Si tout s'est bien passé elle **renvoie le nombre de lignes modifiées**, qui peut être égal à zéro, ce qui est différent de *false*.

Voyons un petit exemple, imaginons que Paul n'ait pas fait de ski, mais du snowboard, pour corriger cette erreur de saisie nous allons utiliser le code suivant :

```
    $wpdb->prefix.'ma_table',
    array('activite' => 'Ski'),
    array('nom' => 'Paul')
);
```

Il n'est pas nécessaire de signaler les types des valeurs puisque nous manipulons des chaînes de caractères.

$wpdb->delete

Finissons ce chapitre par la **suppression** d'enregistrements dans la base de données à l'aide de la fonction ***delete***. Pour procéder WordPress n'a besoin de connaître que 2 informations : **le nom de la table** à manipuler et les **conditions à appliquer** pour supprimer les données. Comme dans les fonctions précédentes il est possible d'indiquer le type des champs de la condition, mais cela reste optionnel.

Comme les fonctions précédentes, **cette fonction renvoie au choix le nombre de lignes affectées en cas de succès ou *false*** en cas d'erreur.

La fonction *delete* se présente comme ceci :

```
mixed delete(string $table, array $where [, array|string $format_where]);
```

Afin de garantir le respect de sa vie privée, Jacques a demandé la suppression des données informatiques le concernant. Voici l'instruction à utiliser pour faire disparaître cette personne de nos registres, même si ça fausse toutes nos statistiques.

```
$wpdb->delete(
    $wpdb->prefix.'ma_table',
    array('nom' => 'Jacques')
);
```

D'autres fonctions à connaître

$wpdb->query

Parfois il peut être plus simple d'**écrire sa requête en pur SQL** et de **l'exécuter telle qu'elle**. Dans le dernier exemple je n'ai pu supprimer les enregistrements de que Jacques, mais si j'avais voulu faire disparaître les données de toutes les personnes dont le nom commence par un «J» il m'aurait fallu une boucle. Pour pallier à ce problème je vais utiliser la fonction ***query*** pour exécuter une requête préparée, j'en profite pour utiliser le caractère «%» comme ça vous aurez vu ce dont j'ai parlé plus haut.

```
$wpdb->query( $wpdb->prepare("DELETE FROM {$wpdb->prefix.'ma_table'} WHERE nom LIKE %s%%", "J"));
```

show_errors() et hide_errors()

Bien pratique quand vous en êtes au stade du développement, appeler *$wpdb->show_errors()* vous permet de demander à WordPress d'**afficher les erreurs SQL**, alors que *$wpdb->hide_errors()* va désactiver leur affichage, pensez-y vous passez en production.

Comme vous pouvez le voir, toutes les opérations du CRUD sont couvertes, simpli ées et sécurisées, il n'est donc pas nécessaire d'utiliser un autre mécanisme pour manipuler vos données. Je le répète : avec WordPress vous devez utiliser l'objet $wpdb pour manipuler vos données. Cela peut vous sembler inutile et certains d'entre vous vont peut-être devoir changer leurs habitudes alors que mysql fonctionne encore très bien (nous n'allons pas lancer un débat à ce sujet aujourd'hui...). Néanmoins mysql ne vous permet pas d'anticiper les évolutions technologies. Si un jour les développeurs du core décident de passer à une autre technologie de stockage, seule l'utilisation de wpdb vous permettra de ne pas avoir à réécrire tout votre code. Et quand on propulse son site avec un noyau utilisé dans 1 site internet sur 4 dans le monde, on s'expose forcément à des attaques à grande échelle, alors autant ne pas tenter le diable avec des fonctions perso qui, elles, n'ont pas été testées à grande échelle.

Selon mon expérience, j'estime à environ 1 site sur 10 dont au moins un composant (thème ou extension) n'utilise jamais la classe wpdb, est-ce que vous avez le même chiffre en tête ? Est-ce que vous-même utilisez systématiquement cette classe et ses méthodes ? Si non, pourquoi ? Comme d'habitude, les commentaires sont là pour vous permettre d'enrichir l'article.