

Générer une requête SQL sur WordPress

PHP, SQL, Wordpress

Web Compétences : 5

Wordpress n'est pas magique. Il s'appuie sur une base de données pour stocker les données des publications. Et qui dit base de données donne la possibilité de lancer des requêtes SQL pour les trier selon certains critères. Et Wordpress a tout prévu, jusqu'au actions permettant d'interférer dans une requête.

Publié dimanche 15 décembre 2019, 13h42
Modifié lundi 26 août 2024, 10h06



By Olivier Paudex

Introduction

Une page d'archives sans filtre de recherche n'est tout simplement pas utile. Le précédent chapitre " **Formulaire et filtre de recherche**" donnait la démarche pour construire un formulaire à l'aide d'Elementor sur la page d'archives et de récupérer les données dans la barre de requêtes du navigateur. L'explication détaillée qui va suivre permettra à cette nouvelle URL de questionner la base de données de WordPress en utilisant le langage SQL.

Affichage de la requête SQL

La page d'archives de la section "**voyages**" commence à ressembler à quelque chose. En plus d'afficher les publications du même type, elle affiche un formulaire réalisé avec Elementor. Celui-ci nous permet d'entrer des critères de recherche. La partie suivante va nous permettre de créer la requête SQL.

Avant toute chose, et pour bien séparer le code PHP, nous allons créer un nouveau fichier portant le nom de "**Travels_Filter.php**".

- Ajoutez sa dénomination au fichier "**functions.php**", puis créez-le dans le dossier "**php**".

```
// Travelsinclude_once ($dir . '/php/Travels_Filter.php');
```

- Dans le fichier "**Travels_Filter.php**", créez la fonction "**dump_request_travels**" comme ci-dessous.

> Cette fonction n'est utile que pendant la réalisation de notre page d'archives. Elle utilise le crochet **"posts_request"**, qui permet d'afficher en clair la requête SQL, dans le but de comprendre celle-ci et de la corriger si nécessaire. En production, il faudra supprimer le code où le mettre en commentaire.

```
<?php
  /**
  * Plugin Name: Travels_Filter
  * Description: Travels archives filter
  * Author: Olivier Paudex
  * Author Web Site: https://www.fuyens.ch
  */
  function dump_request_travels ($input,$query) {
    // Run only on travels archive
    if (!is_admin() && $query->is_post_type_archive('travels')) {
      var_dump ($input);
    }
    return $input;
  }  add_filter ('posts_request', 'dump_request_travels',10,2);
```

Rechargez votre page d'archives et la requête SQL va apparaître au sommet de celle-ci.

Les variables de requêtes

Dans la requête SQL, les paramètres sur lesquels va se dérouler l'action sont du type "**?s=bern&travel_country=suisse**". Si le paramètre "**s**" est bel et bien connu de WordPress comme étant celui de la recherche, tous les autres sont inconnus. WordPress ne sait quoi faire de telles informations, alors il pense que c'est le nom d'une page et essaie de l'afficher. Mais comme celle-ci n'existe pas, il va afficher **une page d'erreur 404, page non trouvée**

Le code ci-dessous va remédier à ceci. La fonction "**add_query_travels**" va exécuter un filtre "**query_vars**"

pour que WordPress puisse prendre connaissance de ces nouvelles variables de requêtes. On appelle ces dernières les "**query vars**", dans le jargon WordPress.

Toutes les variables sont à inscrire dans un tableau (ici **$vars**). Et pour enjoliver le tout, il est possible de remplacer la fameuse variable "**s**" par une autre bien plus parlante "**search**".

> Attention, si vous me suivez dans l'exemple, n'oubliez pas de remplacer le paramètre "**s**" par "**search**", dans la fonction "**travels_filter**", vu au chapitre précédent.

```
function add_query_travels_vars_filter ($vars) {

    // Global
    $vars[] .= 'search';
    // Type travels (ACF)
    $vars[] .= 'travel_description';
    $vars[] .= 'travel_country';
    $vars[] .= 'travel_month';
    $vars[] .= 'travel_year';
    $vars[] .= 'travel_taxonomy';
    $vars[] .= 'travel_order';

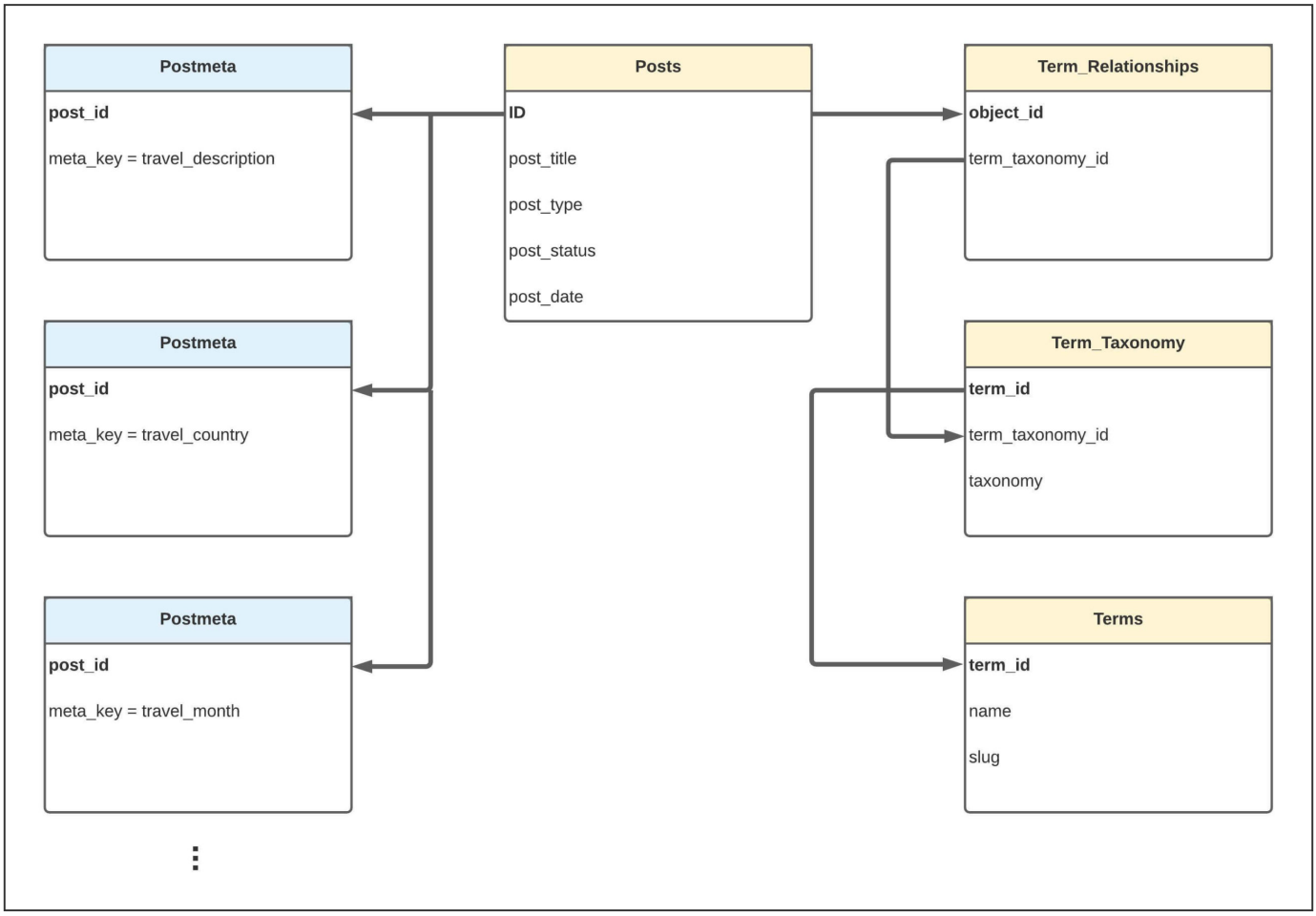
    return $vars;
  }  add_filter('query_vars', 'add_query_travels_vars_filter');
```

Schéma de la base de données de WordPress

Pour autant, le faite de créer les "**query vars**", ne va pas tout résoudre. Faut-il encore écrire la requête SQL correctement. Pour réaliser ceci, un petit coup d'œil au schéma de principe de WordPress s'impose.

WordPress est composé de **douze tables** dans sa version de base. Ci-dessous, je vous montre le schéma de **cinq tables**, sans les relations de cardinalités, constituant WordPress et sur lequelles ce chapitre va s'appuyer. Les tables à entête jaune sont des tables existant physiquement dans la base de données. Les tables à entête bleue (**Postmeta**) ne représentent en réalité qu'une seule table. Dans l'exemple qui va suivre, elles représentent chacune des **"query vars"**, à l'exception de **"travel_taxonomy"** qui va rechercher les données dans les trois tables jaunes de droite. Enfin, **"travel_order"** qui n'est pas une donnée mais une requête de tri, n'utilise pas de table à proprement parlé.

La règle est simple. Tous champs personnalisés créés avec **le plugin ACF** ajoute une pseudo table "**postmeta**" à entête bleue.



Un extrait du schéma de la base de données de WordPress

Les jointures

La première étape de notre code va s'occuper de joindre les tables entre-elles. En gros, cela représente les flèches. Pour créer les jointures en SQL, WordPress à prévu un filtre qui se nomme '**posts_join**".

- Commencez par créer une fonction "**travels_posts_join**" se basant sur ce filtre.

La fonction possède deux arguments (**$join et $query**). Le premier n'est rien d'autre que le résultat de la jointure que l'on retourne d'ailleurs en fin de fonction. Le deuxième est la requête proprement dite.

Enfin, on initialise une variable de type global, "**$wpdb**", qui est une instance de la classe wpdb. Elle permet de se connecter et de dialoguer avec la base de données.

```
function travels_posts_join ($join, $query) {

  global $wpdb;
  return $join;
}add_filter ('posts_join', 'travels_posts_join',10,2);
```

Limiter l'exécution de la fonction

La fonction, comme décrite ci-dessus va s'exécuter sur toutes les pages du site, si l'on ne limite pas son exécution. Ici, on veut qu'elle se lance uniquement sur la page d'archives des**"voyages"**. Ceci est possible grâce à la fonction WordPress "**is_post_type_archive()**".

- Ajoutez cette ligne au code ci-dessus.

```
// Searching and not in admin
if (!is_admin() && $query->is_post_type_archive('travels')) {
}
```

Remise à zéro de la requête

Avant toute chose, il faut remettre à zéro la requête. Ceci se fait tout simplement en effaçant le contenu du paramètre de sortie "**$join**".

- Ajoutez cette ligne au code ci-dessus.

```
// Reset initial join (!! Very important !!)$join = '';
```

Préfix et collecte des noms des tables

Chaque table WordPress porte un numéro de préfix et commence par l'acronyme "**wp**". Dans l'exemple, la table " **Posts**" porte en réalité le nom de "**wp_123456_posts**". Le nombre à 6 chiffres est unique pour une base de données. Pour faciliter la lecture, il est possible de stocker ces noms de tables (ou de champs) dans des variables. Le prefix se récupère avec la syntaxe "**{$wpdb->prefix}**".

- Ajoutez cette ligne au code ci-dessus.

```
// Field names all$post_ID = "{$wpdb->prefix}posts.ID";
```

L'exemple ci-dessus stocke le nom du champ "**ID**" de la table "**Posts**" dans la variable "**$post_ID**".

La suite du code est du même style. Pour chaque champ personnalisé :

- Une variable est créée pour stocker le nom de la table "**postmeta**".
- Une variable est créée pour stocker le nom du champ "**postmeta.post_id**".
- Une variable est créée pour stocker le nom du champ "**postmeta.meta_key**".

Puis, pour la taxonomie :

- Une variable est créée pour stocker le nom de la table "**term_relationships**".
- Une variable est créée pour stocker le nom du champ "**term_relationships.object_id**".
- Une variable est créée pour stocker le nom du champ "**term_relationships.term_taxonomy_id**".
- Une variable est créée pour stocker le nom de la table "**term_taxonomy**".
- Une variable est créée pour stocker le nom du champ "**term_taxonomy.term_taxonomy_id**".
- Une variable est créée pour stocker le nom du champ "**term_taxonomy.term_id**".
- Une variable est créée pour stocker le nom de la table "**terms**".
- Une variable est créée pour stocker le nom du champ "**terms.term_id**".

```
// Field names travels
$meta_travel_description = "{$wpdb->prefix}postmeta travel_description";
$meta_travel_description_ID = "travel_description.post_id";
$meta_travel_description_key = "travel_description.meta_key";
$meta_travel_country = "{$wpdb->prefix}postmeta travel_country";
$meta_travel_country_ID = "travel_country.post_id";
$meta_travel_country_key = "travel_country.meta_key";
$meta_travel_month = "{$wpdb->prefix}postmeta travel_month";
$meta_travel_month_ID = "travel_month.post_id";
$meta_travel_month_key = "travel_month.meta_key";
$meta_travel_year = "{$wpdb->prefix}postmeta travel_year";
$meta_travel_year_ID = "travel_year.post_id";
$meta_travel_year_key = "travel_year.meta_key";
// Field names taxonomy
$tr_travels_terms = "{$wpdb->prefix}term_relationships tr_travel_terms";
$tr_travels_terms_ID = "tr_travel_terms.object_id";
$tr_travels_tt_terms_ID = "tr_travel_terms.term_taxonomy_id";
$tt_travels_terms = "{$wpdb->prefix}term_taxonomy tt_travel_terms";
$tt_travels_terms_ID = "tt_travel_terms.term_taxonomy_id";
$tt_travels_t_terms_ID = "tt_travel_terms.term_id";
$t_travels_terms = "{$wpdb->prefix}terms t_travel_terms";
$t_travels_terms_ID = "t_travel_terms.term_id";
```

Enfin, les jointures sont créées à l'aide du code ci-dessous. Le code est beaucoup plus lisible que si, pour chaque table et chaque champ, le préfix aurait été ajouté.

```
// Join clauses travels
$join .= " LEFT JOIN $meta_travel_description ON ($post_ID = $meta_travel_description_ID)";
$join .= " AND $meta_travel_description_key = 'travel_description'";
$join .= " LEFT JOIN $meta_travel_country ON ($post_ID = $meta_travel_country_ID)";
$join .= " AND $meta_travel_country_key = 'travel_country'";
$join .= " LEFT JOIN $meta_travel_month ON ($post_ID = $meta_travel_month_ID)";
$join .= " AND $meta_travel_month_key = 'travel_month'";
$join .= " LEFT JOIN $meta_travel_year ON ($post_ID = $meta_travel_year_ID)";
$join .= " AND $meta_travel_year_key = 'travel_year'";
$join .= " LEFT JOIN $tr_travels_terms ON ($post_ID = $tr_travels_terms_ID)";
$join .= " LEFT JOIN $tt_travels_terms ON ($tr_travels_tt_terms_ID = $tt_travels_terms_ID)";
$join .= " LEFT JOIN $t_travels_terms ON ($tt_travels_t_terms_ID = $t_travels_terms_ID)";
```

Classement ordonné

La suite du code va permettre de gérer l'ordre des publications, soit de la plus ancienne à la plus récente (ordre croissant) ou le contraire (ordre décroissant).

Je passe les premières lignes de code qui ne font que répéter le principe mis en place pour les jointures.

La ligne qui débute par "**$query->query_vars**", récupère la donnée du champ "**travel_order**", si celle-ci existe, et la stocke dans une variable "**$travel_order**". Sinon, la variable prend comme valeur par défaut le mot " **desc**", pour signifier que le classement se fera dans l'ordre décroissant.

- Ajoutez ces lignes dans une nouvelle fonction "**travels_posts_orderby**", qui va utiliser le filtre " **posts_orderby**".

```
function travels_posts_orderby ($orderby, $query) {
  global $wpdb;
  // Searching and not in admin
  if (!is_admin() && $query->is_post_type_archive('travels')) {
    // Reset initial orderby (!! Very important !!)
    $orderby = '';
    // Tables names
    $post_date = "{$wpdb->prefix}posts.post_date";
    // Get the GET parameters
    $query->query_vars['travel_order'] ? $travel_order = trim(rawurldecode($query->query_vars[
    // Order by clause
    $orderby .= " $post_date " . $travel_order;
  }
  return $orderby;
}add_filter ('posts_orderby', 'travels_posts_orderby',10,2);
```

Les critères de la requête

Pour terminer le code, il va falloir créer une dernière fonction qui porte le nom de "**travels_posts_where**" et qui utilise le filtre "**posts_where**". Les premières lignes sont identiques aux fonctions de jointure et de classement.

Vient ensuite l'écriture de la clause "**where**". Celles-ci commencent toujours par l'appel de la méthode "**$wpdb->prepare()**". Elle permet de faire d'une pierre deux coups. Le premier permet de concaténer les clauses "**where**". La deuxième permet de gérer la sécurité de ce que l'on appelle dans le jargon, l'injection SQL. Elle permettrait à des visiteurs mal intentionnés, d'injecter du code SQL à partir d'un navigateur, pour, par exemple, lancer des ordres de suppressions.

> Gérer la sécurité des URL dans un site web n'est certainement pas une tâche facile, mais le faite de préparer les requêtes SQL à l'avance, en ne permettant qu'une certaine syntaxe, interdit l'injection SQL automatiquement.

- Ajoutez cette nouvelle fonction à la suite des autres.

```
function travels_posts_where ($where, $query) {

  global $wpdb;
  // Searching and not in admin
  if (!is_admin() && $query->is_post_type_archive('travels')) {

    // Reset initial where ((!! Very important !!))
    $where = '';
    // Field names all
    $post_ID = "{$wpdb->prefix}posts.ID";
    $post_title = "{$wpdb->prefix}posts.post_title";
    $post_content = "{$wpdb->prefix}posts.post_content";
    $post_type = "{$wpdb->prefix}posts.post_type";
    $post_status = "{$wpdb->prefix}posts.post_status";
    $post_author = "{$wpdb->prefix}posts.post_author";
    // Field names travels
    $meta_travel_description = "{$wpdb->prefix}postmeta travel_description";
    $meta_travel_description_key = "travel_description.meta_key";
    $meta_travel_description_value = "travel_description.meta_value";
    $meta_travel_country = "{$wpdb->prefix}postmeta travel_country";
    $meta_travel_country_key = "travel_country.meta_key";
    $meta_travel_country_value = "travel_country.meta_value";
    $meta_travel_month = "{$wpdb->prefix}postmeta travel_month";
    $meta_travel_month_key = "travel_month.meta_key";
    $meta_travel_month_value = "travel_month.meta_value";
    $meta_travel_year = "{$wpdb->prefix}postmeta travel_year";
    $meta_travel_year_key = "travel_year.meta_key";
    $meta_travel_year_value = "travel_year.meta_value";
    $t_travels_terms = "{$wpdb->prefix}terms t_travel_terms";
    $t_travels_terms_slug = "t_travel_terms.slug";
    $tt_travels_terms = "{$wpdb->prefix}term_taxonomy tt_travel_terms";
    $tt_travels_terms_taxonomy = "tt_travel_terms.taxonomy";
    // Prepare the placeholder for the post_type
    $custom_post_type_placeholder = '%s';
```

```
    $custom_post_type = "travels";
    // Get the GET parameters
    $query->query_vars['search'] ? $search_text = trim(rawurldecode($query->query_vars['search
    $query->query_vars['travel_country'] ? $travel_country = trim(rawurldecode($query->query_va
    $query->query_vars['travel_month'] ? $travel_month = trim(rawurldecode($query->query_vars[
    $query->query_vars['travel_year'] ? $travel_year = trim(rawurldecode($query->query_vars['tr
    $query->query_vars['travel_taxonomy'] ? $travel_taxonomy = trim(rawurldecode($query->query_
    // Write the where clause
    if (!empty($search_text)) {
      $where .= $wpdb->prepare(" AND (($post_title LIKE '%%%s%%')", $search_text);
      $where .= $wpdb->prepare(" OR ($post_content LIKE '%%%s%%')", $search_text);
      $where .= $wpdb->prepare(" OR ($meta_travel_description_key = 'travel_description' AND $m
    }
    // Where clause travels
    if (!empty($travel_country)) {
      $where .= $wpdb->prepare(" AND ($meta_travel_country_key = 'travel_country' AND $meta_tra
    }
    if (!empty($travel_month) && $travel_month != pll__('all')) {
      $where .= $wpdb->prepare(" AND ($meta_travel_month_key = 'travel_month' AND $meta_travel_
    }
    if ($travel_year > 0) {
      $where .= $wpdb->prepare(" AND ($meta_travel_year_key = 'travel_year' AND $meta_travel_ye
    }
    if (!empty($travel_taxonomy) && $travel_taxonomy != 'tous')) {
      $where .= $wpdb->prepare(" AND $t_travels_terms_slug = %s", $travel_taxonomy);
      $where .= $wpdb->prepare(" AND $tt_travels_terms_taxonomy = %s", 'travel_type');
    }
    // Where clause all
    $where .= $wpdb->prepare(" AND $post_type IN ($custom_post_type_placeholder)",$custom_post_type);
    $where .= " AND ($post_status = 'publish'";
    $where .= " OR $post_author = 1";
    $where .= " AND $post_status = 'private')";
    // Group by
    $where .= " GROUP BY $post_ID";
  }
  return $where;
}add_filter('posts_where', 'travels_posts_where',10,2);
```

Et WP_Query, alors …

Cetains puristes de WordPress me diront qu'il existe des fonctions '**WP_Query**", embarqués dans le noyau de celui-ci. La réponse à cette dernière remarque est que '**WP_Query**" ne permet pas d'être aussi souple, surtout dans les relations algébriques **AND et OR**.

J'utilise également "**WP_Query**", par exemple pour afficher les publications sur la page d'accueil ou encore les minis publications mise en avant. Elle font le sujet d'un chapitre **WP Query avec Elementor**.

Le mot de la fin

Ceci met fin à ce chapitre, certainement celui qui m'a demandé le plus de temps de codage pour arriver à mes fins. Il est rapide, très maniable et s'adapte à n'importe quelle situation. Si vous désirez créer d'autres types personnalisés, aucun problème. Il suffit de copier-coller le code dans un nouveau fichier et d'adapter les noms des variables.

Le prochain chapitre va s'occuper de la réécriture des URL. En effet, écrire un URL de la manière décrite dans cette publication, n'est pas très **SEO** compatible. **SEO** est un terme anglais qui signifie littérallement " **search engine optimization**". Beaucoup de moteur de recherche, dont **Google** s'appuie sur les URL pour indexer les pages d'un site web. Si des derniers contiennent des paramètres du style de " **?s=bern&travel_country=suisse**", vous êtes mal parti pour recevoir une bonne note et allez être positionné au fond du classement.

Voilà, suivez-moi sur ce sujet dans la prochaine publication "**SEO et réécriture des URL**".