ENI Service - WordPress - PHP

Paul Schuhmacher

Durée : 28h

Novembre 2025

Partie 3 : Les API de WordPress

Partie 3 : Les API de WordPress

Objectifs : Comprendre et savoir utiliser :

- La **Template hierarchy**;
- La **boucle WordPress** et les boucles *customs*;
- **Template tags** : développer des templates idiomatiques et sécurisés, gestion des *permaliens*;
- Les **API de la base de données** : `WP_Query`, `$wp_db` et *custom query*;
- **Hooks** : **actions** et **filtres**.

> Savoir utiliser ces API permet de développer des thèmes (parent ou enfant) sur mesure et d'implémenter de nombreuses fonctionnalités !

Thème fil rouge

Pour explorer toutes ces API nous allons développer un thème sur mesure.

Nous allons développer un site dédié *à la consultation et à la gestion d'une bibliothèque*.

> Remarque : Par *soucis de simplicité* et au *regard du temps imparti*, certaines fonctionnalités seront intégrées directement dans le thème plutôt que dans des plugins. Les bonnes pratiques d'architecture d'une application WordPress seront abordées dans la suite de la formation.
> N'hésitez pas à faire des suggestions !

Remarque

- Ces spécifications nous serviront de **guide et de cadre**, **le but n'est pas de tout implémenter !**
- Ces spécifications **couvrent de nombreux usages des différentes API de WordPress** (Template hierarchy, hooks, template tags, Cron Job, log, WP_Query, transient API, etc.)
- Nous pourrons les changer au besoin.

Thème fil rouge : Spécifications

Le site doit :

- Exposer un catalogue de livres numériques ;
- Exposer un **formulaire de recherche central** avec des filtres (auteur, titre, catégorie, disponibilité, etc.) ;
- Exposer un blog où des actualités sont publiées régulièrement ;
- Chaque livre, auteur·ice et éditeur dispose de sa propre page ;
- Un livre peut être lié jusqu'à trois autres livres pour proposer des recommandations ou différentes éditions.

Thème fil rouge : Rôle et permissions

- Dispose de six rôles :
  - **Visiteur** : consulter les ressources;
  - **Abonné** (utilisateur avec un compte) : **emprunter** des livres;
  - **Gestionnaire de collections** : peut éditer les livres, auteurs et maisons d'éditions;
  - **Éditeur** : peut publier des articles de blog (*news*);
  - **Libraire** : peut marquer le retour d'un livre;
  - **Administrateur**.

Thème fil rouge : Spécifications du système d'emprunt et simplifications

- Un abonné peut emprunter un livre directement, sans passer par le **libraire**;
- Un **abonné** peut emprunter jusqu'à cinq livres simultanément;
- Un emprunt dure 3 semaines maximum;
- Lorsqu'un **abonné** emprunte le livre, le système considère qu'un exemplaire du livre est *immédiatement* emprunté et décrémente le nombre de copies disponibles de un;
- Un livre est indisponible à l'emprunt si son nombre d'exemplaires disponibles est nul;
- Un livre indisponible à l'emprunt apparaît toujours dans le catalogue;
- Un livre indisponible à l'emprunt affiche l'information "Indisponible pour le moment";
- Le **libraire** informe le système du retour d'un livre. L'exemplaire retourné est alors *immédiatement* disponible à l'emprunt, le nombre de copies disponible est incrémenté de un;

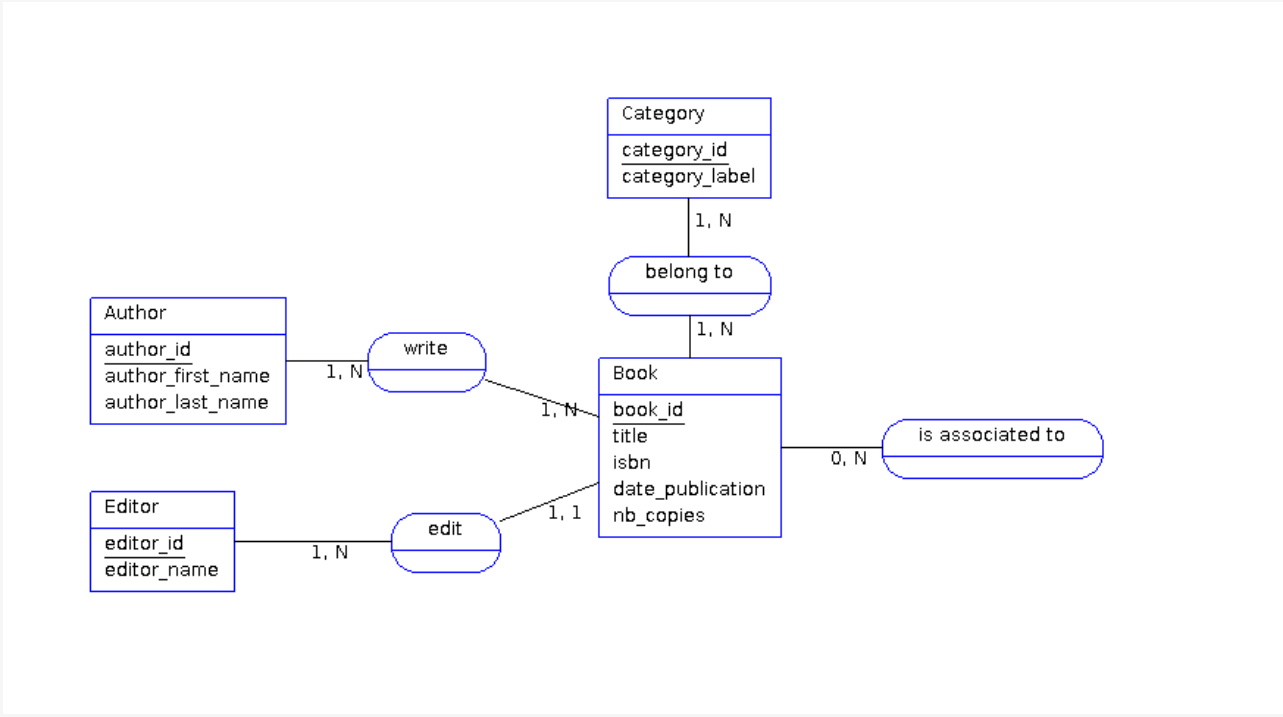
Thème fil rouge : Spécifications du système de log et de notifications

- Le système envoie une notification email à tout abonné lorsque la durée d'emprunt d'un ouvrage est dépassée;
- Le système *log* les évènements suivants :
  - Un emprunt ;
  - Un retour ;
  - Création de compte ;
  - Suppression de compte ou tentative de suppression de compte ;
- Lors de la suppression de compte, le système vérifie que l'abonné n'a pas d'emprunts en cours. Si c'est le cas, la suppression est annulée et une notification email est envoyée à l'abonné lui indiquant la liste des retours à faire. Le libraire est également averti par email (compte concerné, livres à retourner)

Thème fil rouge : Modèles de données

{.marp-bg-img}

- Un *livre* est défini par :
  - Un ISBN,
  - Un titre,
  - Une date de publication,
  - **Un nombre d'exemplaires**.
- Un·e *auteur·ice* est défini par :
  - Un prénom,
  - Un nom,
  - Une date de naissance,
- Une *maison d'édition* est définie par :
  - Un nom.



Règles de gestion:

- Un livre peut être écrit par un·e ou plusieurs auteur·ices (*many-to-many*);
- Un livre peut être lié directement jusqu'à trois autres livres;
- Un livre peut appartenir à une ou plusieurs catégories (*many-to-many*);
- Une maison d'édition peut éditer un ou plusieurs livres;

Créer un nouveau thème 1/2

Dans `wp-content/themes` :

1. **Créer** le dossier `library` ;
2. Y **créer** les fichiers requis pour un thème :
    i. `index.php` ;
    ii. `style.css` .
3. Placer les **métadonnées** nécessaires dans `style.css` :

```
/*
Theme Name: Bibliothèque
Author: John Doe
Description: Un site de bibliothèque municipale
Version: 1.0
License: GPL
Text Domain: BM
Tags:
*/
```

Créer un nouveau thème 2/2

4. Dans `index.php` :

```php
<?php

var_dump("index.php : Bibliothèque municipale");
```

5. **Activer** le thème.

Sitemap (Visiteur)

```
home
    - Liste des ouvrages
        - Détail d'un ouvrage
    - Liste des auteur·ices
        - Détail d'un·e auteur·ice
    - Liste des éditeurs
        - Détail d'un éditeur
    - Blog
        - Détail d'un post
    - Categories
        - Term
            - Détail d'un ouvrage
    - À propos
    - Login
    - Création de compte
```

Sitemap (Abonné)

```
home
    - Liste des ouvrages
        - Détail d'un ouvrage
    - Liste des auteur·ices
        - Détail d'un·e auteur·ice
    - Liste des éditeurs
        - Détail d'un éditeur
    - Blog
        - Détail d'un post
    - Categories
        - Term
            - Détail d'un ouvrage
    - À propos
    - **Mon compte** (consulter les livres empruntés)
    - Login
    - Création de compte
```

Sitemap et templates WordPress

```
home (front-page)
    - Liste des ouvrages (archive)
        - Détail d'un ouvrage (single book)
    - Liste des auteur·ices (archive)
        - Détail d'un·e auteur·ice (single author)
    - Liste des éditeurs  (archive)
        - Détail d'un éditeur (single editor)
    - Categories (taxonomy)
        - Term (archive-term)
            - Détail d'un ouvrage (single book)
    - Blog (archive des posts)
        - Détail d'un éditeur (single post)
    - À propos (single)

    - Login (fourni, à custom)
    - Création de compte (fourni, à custom)
```

Préparation des templates

Comme vu au module 01, WordPress analyse l'URL pour construire un contexte (récupérer les données pertinentes auprès de la base) et **choisir** un **template** pour rendre ces données.

La *Template Hierarchy* est la stratégie employée par WordPress pour sélectionner le template PHP adapté à la ressource demandée.

Bien la comprendre est indispensable pour développer un thème et bénéficier du framework.

Source (à garder auprès de soi quand on développe des templates)


Archive Page
Author Archive
author-$nicename.php
author-$id.php
author.php
Category Archive
category-$slug.php
category-$id.php
category.php
Custom Post Type Archive
archive-$posttype.php
Custom Taxonomy Archive
taxonomy-$taxonomy-$term.php
taxonomy-$taxonomy.php
taxonomy.php
Date Archive
Year Archive
Month Archive
Day Archive
date.php
Tag Archive
tag-$slug.php
tag-$id.php
tag.php
archive.php
index.php
Singular Page
Single Post Page
Attachment Post
$mimetype-$subtype.php
$subtype.php
$mimetype.php
attachment.php
Custom Post
If selected: $custom.php
single-$posttype-$slug.php
single-$posttype.php
Blog Post
If selected: $custom.php
single-post.php
single.php
singular.php
Static Page
Page Template
Custom Template
$custom.php
Default Template
page-$slug.php
page-$id.php
page.php
Site Front Page
front-page.php
Page Shown On Front
Posts Shown On Front
home.php
Blog Posts Index page
Error 404 Page
404.php
Search Result Page
search.php
For oEmbeds:
embed-{post-type}-{post_format}.php
embed-{post-type}.php
embed.php
wp-includes/theme-compat/embed.php
Primary Template
Secondary Template
Variable Template
Page Type

Template hierarchy : du type de ressource (URL) au template, via des fallbacks

> À lire de **gauche** (input *via* URL => type de contenu) à **droite** (=> template choisi par WordPress *core*)

Fonctionnement et intérêt

Fonctionnement

- Par défaut, chaque template dispose d'un un template de *fallback*. Le template de *fallback* est celui utilisé si le template est absent du thème. Le template `index.php` sert de **fallback à tous les autres** (voilà pourquoi il est requis !)
- Par exemple, le template `archive.php` peut être utilisé pour afficher les *Custom Post Types*(*livres*, *auteurs*, *éditeurs*, etc.), les posts d'une *catégorie*, etc. Si on souhaite un template de page différent pour afficher les éditeurs, on pourra créer un template supplémentaire (`archive-editor.php`). On part du *général* vers le *spécifique*.

Intérêt

- Possibilité de développer des templates progressivement, **uniquement lorsque c’est utile**;
- **Moins de code dupliqué**, maintenance simplifiée.

S'essayer à la template hierarchy avec les taxonomies natives (category et tag)

1. **Créer** un template `archive.php` :

```php
<?php
var_dump(__FILE__);
```

2. **Visiter** l'URL `/category/uncategorized/` ;
3. **Créer** un template `category.php` ;
4. **Vérifier** que la *template hierarchy* est bien respectée;

Exercice

1. Via le *dashboard*, **créer** un tag *"Coup de coeur"*, qui aura pour slug `coup-de-coeur` ;
2. Se rendre à l'URL `/tag/coup-de-coeur` ;
3. **Créer** un template dédié aux tags (`tag.php`), **puis uniquement dédié au tag** `coup-de-coeur` .
4. **Quel template sera utilisé** si l'URL `/tag/rentree-litteraire-2025` est demandée ?

Correction

1. **Créer** le template `tag.php` :

```php
<?php
var_dump(__FILE__);
```

2. Accéder à l'URL `/tag/coup-de-coeur` avec votre navigateur favori";

3. **Créer** le template `coup-de-coeur.php` :

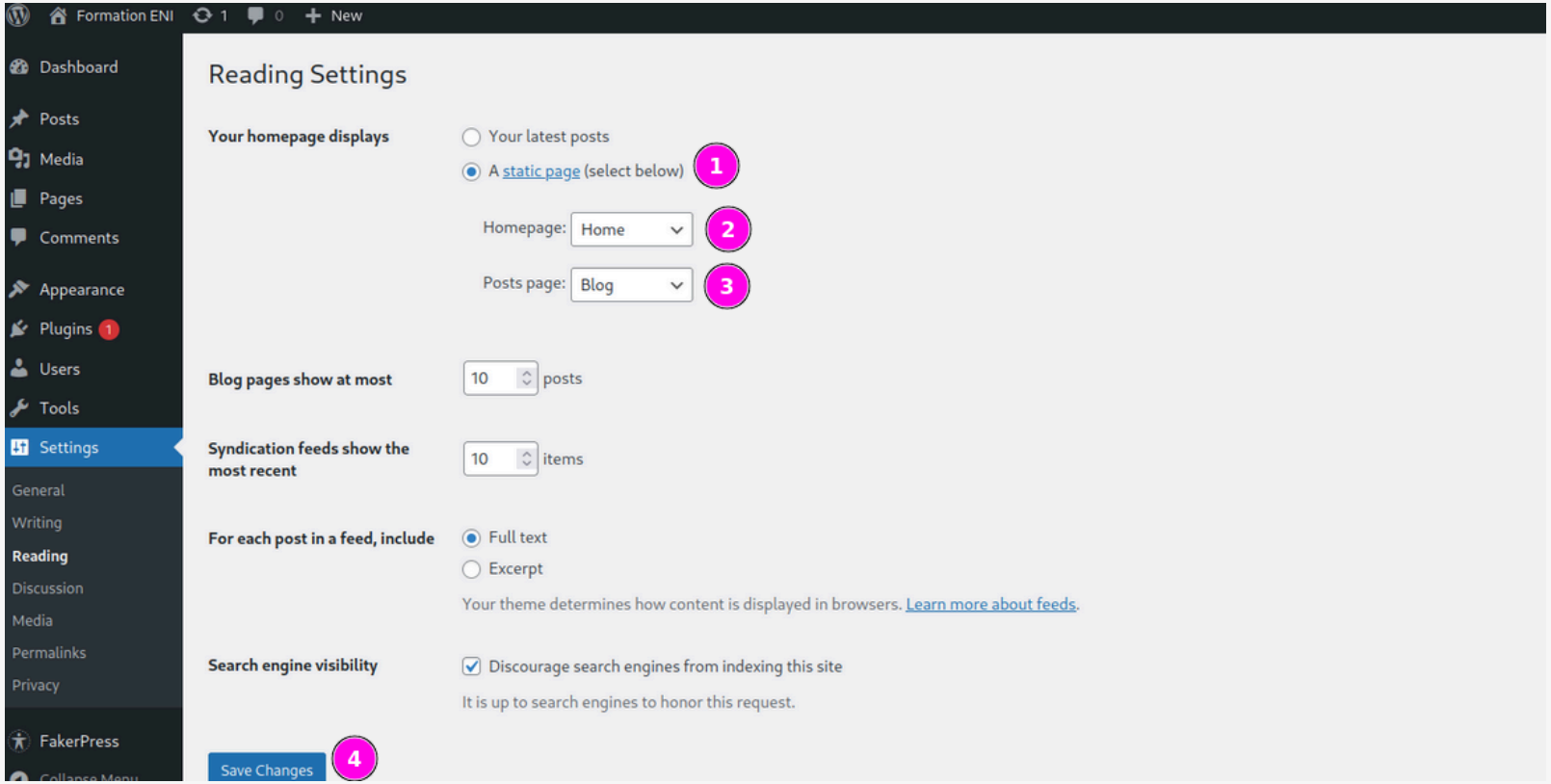
```php
<?php
var_dump(__FILE__);
```

Vérifier l'URL `/tag/coup-de-coeur`. Il doit s'afficher le chemin du template "tag-coup-de-coeur.php";
4. Le template `tag.php`, qui sert de fallback car le template `rentree-litteraire-2025.php` n'existe pas.

Créer les pages Home et *Blog* (articles)

- **Créer** une page "News" (slug `news`), pour afficher la liste des posts;
- **Créer** une page "Home" (slug `home`), pour afficher la page d'accueil du site;

Configurer la page d'accueil et la page des articles



> **Configuration la plus utilisée**, sauf si vous faites un blog ou voulez mettre en avant uniquement des articles. On pourra évidemment ajouter des articles sur la home si on le souhaite.

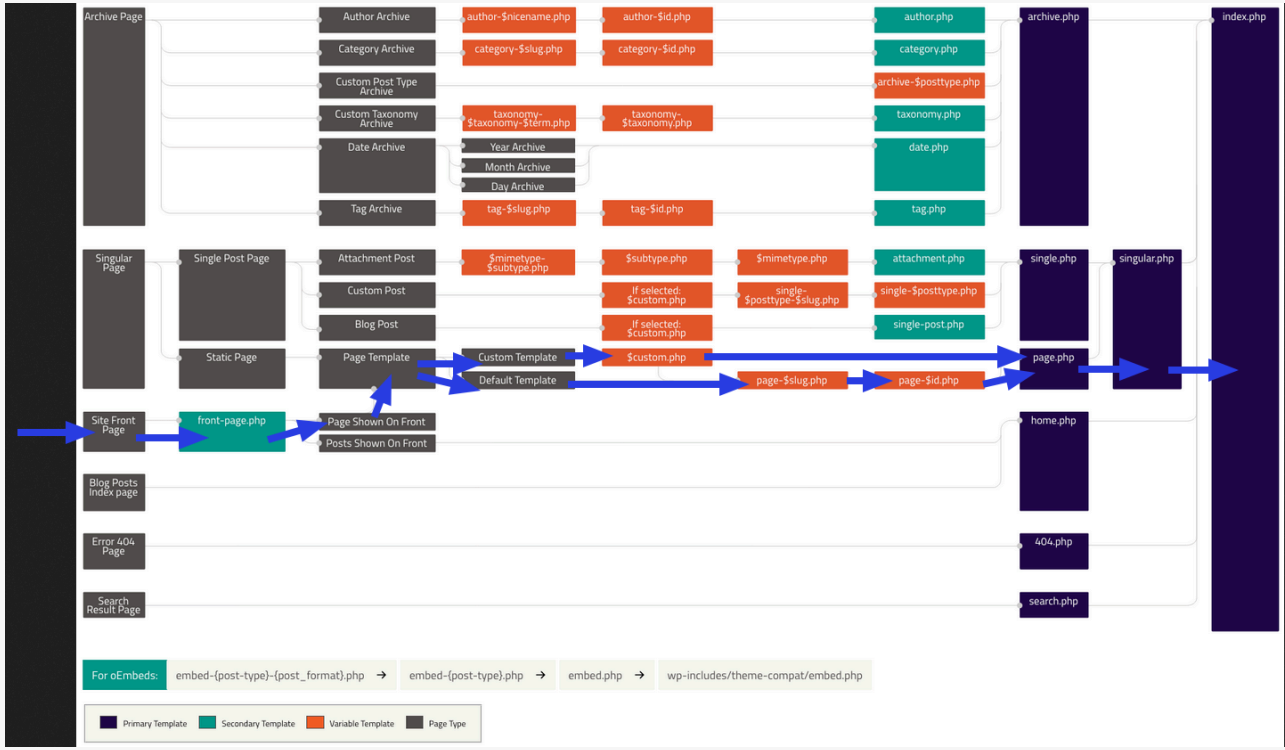
Template de la page d'accueil

{.marp-bg-img}

La page d'accueil est sur l'url racine (`/`).

On peut utiliser les templates suivants :

- **`front-page.php`** (**recommandé**);
- `custom.php`
- `page-$slug.php` ;
- `page-$id.php` ;
- `page.php` ;
- `singular.php` ;
- `index.php` ;



Le cas des templates *custom page*

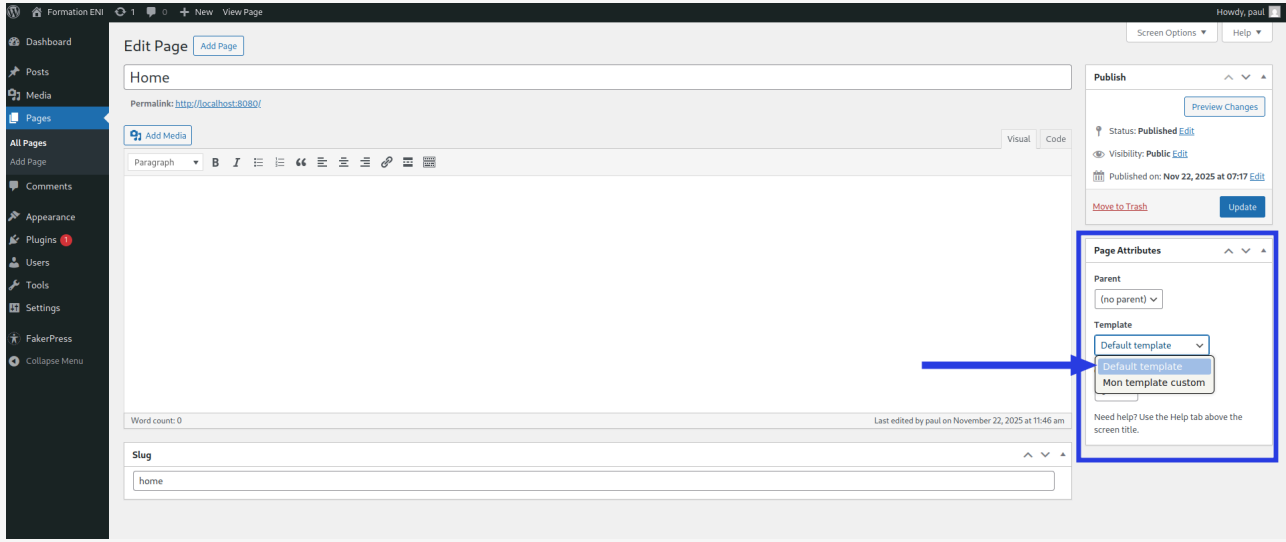
{.marp-bg-img}

Ces templates ne sont pas scannés par défaut par le *core*.

Il faut créer le **template**, par exemple `custom-home.php` et ajouter un header `Template Name: <nom>` dans le fichier :

```php
<?php
// Template Name: Mon template page d'accueil
```

Dans le dashboard d'édition d'une page, la *métabox* `Page Attributes` affiche une option `Template` qui permet de sélectionner un template pour rendre la page.



> Peut être très pratique ! Si besoin d'un template unique pour un contenu, ou permettre aux admins de choisir entre différents templates !

Réécriture des URL des taxonomies par défaut

{.marp-bg-img}

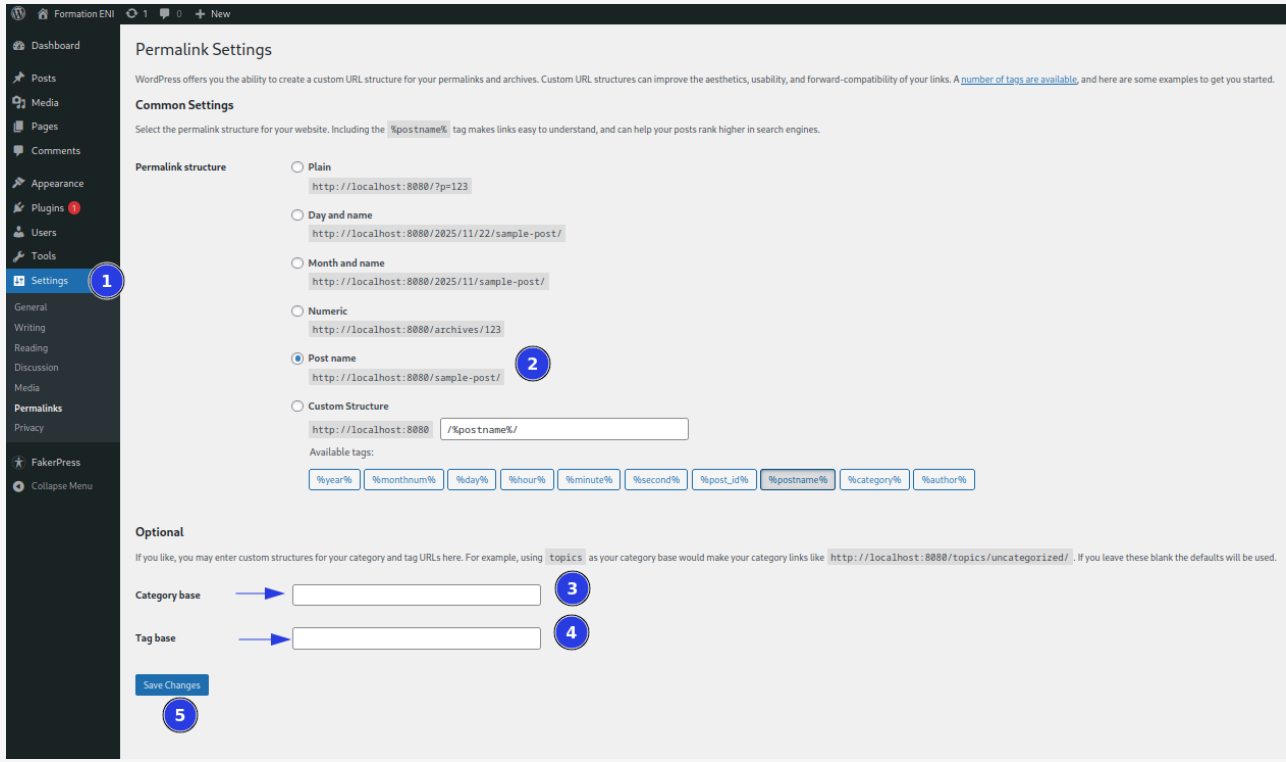
Par défaut, les `path` des taxonomies sont :

- `/category/term` pour la taxonomie hiérarchique `Categories (Catégories)`
- `/tag/term` pour la taxonomie non hiérarchique `Tags (Étiquette)`

On peut les modifier dans l'écran `Permalink Settings` du dashboard (recommandé).

Mettre `categories` et `tags` (au pluriel), pour *uniformiser les noms des urls*.

> Une ressource de type *collection* devrait être au pluriel.



Résumé : URL et templates du thème pour le moment

URL	Ressource	Template
`/`	Page d'accueil	`front-page.php`
`/news`	Actualités (Liste des posts)	`home.php`
`/categories/{term}`	Liste des	`archive.php`
`/tags/{term}`	Liste des	`tag.php`
`/tags/best-seller`	Liste des best-sellers	`tag-best-seller.php`
Autre	Ressource inexistante (erreur client)	`404.php`

Vérifier

Les URL suivantes devraient appeler les bons templates (afficher un message dans chaque template, par exemple `echo __FILE__;`) :

- GET `/`
- GET `/news`
- GET `/categories/uncategorized`
- GET `/tags/best-seller`
- GET `/introuvable`

> Tester `/tags/foo`. Que remarquez-vous ?

=> Snapshot : tag v0.1 <=

Dépôt kit-dev : avec git, se rendre sur le tag `v0.1` pour récupérer le thème à ce stade de développement :

```
git clone https://github.com/paul-schuhm/wordpress-php.git
cd wordpress-php
git checkout v0.1
```

> Des snapshots réguliers du thème fil rouge seront fait pour *voyager* à tous les stades de développement.

Développer ses templates

Une fois que la *template hierarchy* est comprise (garder la *carte* à disposition), il est facile de développer ses templates, **en allant du plus général au plus spécifique**.

Nous continuerons ce travail, mais d'abord il est temps de **développer un template et d'y placer du contenu** !

Générer une page complète (enfin)

Nous allons **développer le template** `home.php` en charge de présenter la liste des articles (*News*).

Header

- Le fichier `header.php`. Inclut tout le code HTML jusqu'au début du contenu.
- **Doit contenir l'appel à la fonction `wp_head()` dans le tag HTML `<head>`.**
- Cet appel déclenche un **hook** (hook de type *action*) permettant l'execution de code pour include des assets (preload fonts, scripts JS, feuilles de style CSS **notamment pour la barre d'admin** et les **plugins**).

```php
<?php
/**
 *
 * Template du header du site.
 *
 * @package BM
 */
?>
<!DOCTYPE html>
<html lang="en">
<head>
 <meta charset="<?php bloginfo( 'charset' ); ?>">
 <meta name="viewport" content="width=device-width, initial-scale=1">
 <?php wp_head(); ?>
</head>
<body <?php body_class(); ?>>
 <header>
  <!-- Navbar (à venir) -->
 </header>
```

Inclure le header

```php
<?php
get_header();
```

Cette fonction inclut automatiquement le fichier `header.php` du thème.

Footer

- Le fichier `footer.php` inclut tout le code HTML de la fin du contenu ainsi que le tag HTML `<footer>`
- Doit appeler la fonction `wp_footer()`.
- Cet appel déclenche le **hook** `wp_footer` (hook de type *action*) permettant l'execution de code pour include des assets (scripts JS **notamment pour la barre d'admin** et les plugins).

```php
<?php
/**
 * Template du footer du site
 *
 * @package BM
 */

?>

<footer>
  &copy; Copyright <?php esc_html_e( date( 'Y' ) ); ?>
 </small>
</footer>
<?php wp_footer(); ?>
</body>
</html>
```

Inclure le footer

```php
<?php
get_footer();
```

Cette fonction inclut automatiquement le fichier `footer.php` du thème. Le footer contient généralement :

- Les informations légales;
- Les informations de contact;
- Le sitemap;
- Etc.

Chaque template de page

Comme le footer et le header sont quasi toujours les mêmes sur toutes les pages, on appelera ces deux fonctions dans chaque template de page. Par exemple notre template `home.php` :

```php
<?php
/**
 *
 * Template pour lister les actualités (posts)
 *
 * @package BM
 */

?>
<?php
get_header();
// Contenu à insérer ici (contenu de <body>)
get_footer();
```

> Vous devez *enfin* voir réapparaître la barre d'admin grâce à l'inclusion du CSS et du JS injecté sur les hooks `wp_head` et `wp_footer`. Activer le plugin Query Monitor pour la suite si ce n'est pas encore le cas.

Changer de header ou de footer

Parfois, on a besoin d'afficher temporairement une information supplémentaire, notamment dans le header (période de l'année, promo, offres spéciales, évènement, etc.)

Avec les fonctions `get_header()` et `get_footer()` changer de template de header ou de footer est très simple.

Par exemple, créer un fichier `header-sale.php`, le customiser et appeler `get_header('sale')`. WordPress va chercher le fichier `header-sale.php` et le charger :

```php
//home.php
<?php
//Défini par un intervalle de temps entre deux dates par ex., ou un paramètre dans le dashboard
$is_sale_period = true;
get_header( $is_sale_period ? 'sale' : '' );
```

La Loop WordPress

La *loop* est au coeur d'un thème WordPress. Elle permet de contrôler comment le contenu est affiché. C'est le composant qui fait le lien entre le *contexte* (requêtes SQL réalisées par le *contrôleur* `WP::main()` en se basant sur l'URL) et les templates (HTML).

On utilisera la *loop* dans *tous* les templates :

- D'une page;
- D'un post;
- D'une archive;
- Etc.

Un *contexte* par défaut est préparé par le contrôleur. Par exemple, sur l'URL `/news` de notre site, le *contexte* préparé contient la liste des posts.

Il est possible d'utiliser plusieurs loops par template, n'importe où pour *étendre* le contexte en fonction des besoins.

La Loop WordPress en pratique

```php
<?php
if ( have_posts() ) :
 while ( have_posts() ) :
  the_post();
   the_title( '<h2>', '</h2>' ); //ex. de template tag
endwhile;
endif;
```

Explications

- `if ( have_posts() )` : *s'il y a des posts* dans le contexte;
- `while(have_posts()) the_post()` : *tant qu'il y a des posts*, les parcourir. `the_post()` **fait avancer un pointeur vers le post suivant** et **charge les données du post courant + ses métadonnées** dans **une variable globale** `$post` pour les rendre accessible aux *template tags*, des fonctions permettant de manipuler le contenu;
- `the_title( '<h2>', '</h2>' )` est un *template tag*, une fonction pour afficher le titre du *post*. On indique ici de le placer entre balises `h2` ;
- `the_post()` applique également un ensemble de filtres au contenu brut stocké en base (ex: shortcodes dans une page, filtre appliqué par un plugin, etc.).

Pratique : Loop WordPress 1/2

1. **Placer** la *loop* dans le template `home.php` ;
2. Utiliser le *template tag* `the_permalink()` pour placer le `h2` du post dans une balise `<a>` afin de créer un lien pour afficher le contenu de l'article;
3. **Publier** plusieurs articles (avec le plugin FakerPress);
4. **Afficher** la page qui liste les posts (`/news`);
5. **Cliquer** sur le lien d'un article. Que s'affiche-t-il ? Pourquoi ?
6. En vous servant de *la template hierarchy*, **créer le template** approprié (le plus générique) pour prendre en charge le rendu d'un post;

Pratique: Loop WordPress 2/2

7. **Ajouter la loop WordPress** dans le **template** en charge de présenter le contenu d'un post. Utiliser le *template tag* `the_content()` pour afficher son contenu principal :
    i. Le titre du post doit apparaître cette fois dans un header de niveau 1 (`h1`),
    ii. Le titre et le contenu sont placés dans une balise HTML `<main>` ;
8. **Tester** en affichant le contenu d'un post;
9. **Afficher** le code source de la page /news et **vérifier** la **validité** du HTML généré auprès du Markup Validation Service du W3C.

> C'est un bon réflexe de vérifier le HTML produit par ses templates pour s'assurer de ne pas avoir oublié de fermer une balise par exemple.

Les templates tags

Les *template tags* sont des **fonctions du core WordPress** pour afficher du contenu, notamment dans la loop WordPress. Ces fonctions offrent une interface claire pour afficher le contenu du site et customiser son affichage. C'est **l'API pour extraire et afficher le contenu des posts** dans les templates.

Par exemple, `the_title()` affiche (`echo`) le titre d'un post ou d'une page au sein de la loop. La fonction `get_the_title()` retourne le titre du post ou d'une page.

Il existe de nombreux template tags.

Template tags principaux

Noter **la convention de nommage** :

- Les template tags en `get_*` **retournent** le contenu (`string`), par ex. `get_the_title()` ;
- Les template tags en `the_*` **affichent** (`echo` , *effet de bord*) le contenu, par ex. `the_title()` .

Les template tags sont des **fonctions**, et la plupart d'entre **acceptent des arguments** pour personnaliser leur fonctionnement.

Voici la liste des principaux *template tags*.

> On apprendre à en utiliser un certain nombre. Toujours se demander quand on a besoin d'accéder à une donnée "existe-t-il un template tag pour ça ?"

Content tags

Pour manipuler/afficher le contenu des *posts* (post, CPT, page) :

```
the_permalink()     // Affiche l'URL du post
the_title()         // Affiche le titre du post
the_ID()            // Affiche l'ID du post
the_content()       // Affiche le contenu du post
the_excerpt()       // Affiche l'excerpt du post
the_date()          // Affiche le datetime de la publication du post
the_time()          // Affiche l'heure de la publication du post
the_author()        // Affiche l'auteur du post
the_tags()          // Affiche le tags du post
the_category()      // Affiche les catégories du post
edit_post_link()    // Affiche un lien d'édition du post uniquement si vous êtes connecté et autorisé à éditer le post
comment_form()      // Affiche un formulaire pour laisser un commentaire
wp_get_attachment_image()
wp_get_attachment_image_src()
post_class() // Affiche un ensemble de classes CSS générées par WordPress (très utile pour l'intégration)
```

Pagination

WordPress fournit des *template tags* nécessaires **pour gérer entièrement la pagination** (génération du markup, création des liens, traitement des liens paginés) :

```
**paginate_links**
the_posts_pagination
the_posts_navigation
previous_posts_link
next_posts_link
```

`paginate_links` :

- Génère l'ensemble des **liens** de pagination **à partir d'une WP_Query** (ou de la **query globale**);
- Retourne le **markup HTML** contenant les URL des pages précédentes/suivantes, prête à afficher;
- **Utilise** automatiquement la structure de **permaliens** du site et **gère les query vars**.

Exemple

```php
<?php
//Template d'archive (template home.php)

$posts_per_page = 5;

global $wp_query;

$args_pagination = [
    'type'      => 'plain',
    'next_text' => '=>',
    'prev_text' => '<=',
    'total'     => $wp_query->max_num_pages,
    'current'   => max(1, get_query_var('paged')),
];

echo paginate_links($args_pagination);
```

> Copier/coller ce code dans `home.php` du thème `library` pour tester, avant `get_footer()`.

Include tags

Pour inclure des templates :

```
get_header()
get_footer()
get_sidebar()
get_template_part() // On y reviendra dans la séquence 5 sur le développement de thème
get_search_form()   // On y reviendra dans la séquence 5 sur le développement de thème
comments_template()
```

> On retrouve nos fonctions `get_header()` et `get_footer()` !

Template parts

Penser les templates comme une **composition d'éléments réutilisables** de template en template : **composants**. Utiliser les composants via le *template tag* get_template_part() :

```
//Fichier parts/list-item-book.php
//Template item de liste (Livre)
//Données : objet de type $BN_Book
<li>
    <h2><?php the_title(); ?></h2>
    //Etc
</li>
```

Depuis `archive-book.php` :

```
<?php
//Utiliser le template du composant (éviter appel à require + définir dépendance du template via $args !)
get_template_part(
 slug: 'list-item-book',
 args: array(
  'book' => $book,
 )
);
```

Taxonomies et termes

```
single_cat_title()
the_category()
single_tag_title()
tag_description()
the_tags()
get_the_term_list()
the_terms()
the_taxonomies()
```

Site, assets tags

Pour obtenir les informations du site et sur les assets pour construire des URL :

```
blog_info();
blog_info('name');
blog_info('admin_email');
blog_info('charset');
blog_info('language');
blog_info('rss_url');

get_stylesheet_directory_uri(); // URL de style.css du thème (racine du thème). Assets du thème
get_template_directory_uri();  // URL du template du thème. Assets du thème
```

Link tags

```
admin_url(); // URL du site (admin)
home_url(); // Site Address (URL de la home), voir Settings/General
get_site_url(); // WordPress Address (URL de l'application WordPress), voir Settings/General


the_permalink() // a connaître
get_permalink() // a connaître
get_post_permalink()
get_page_link()
get_attachment_link()

edit_post_link()
get_edit_post_link()
get_delete_post_link()
edit_comment_link()
edit_tag_link()

get_search_url($s)
get_search_query()
the_feed_link()
```

Login tags

- Fournissent des URLs et *templates* liés à l’authentification WordPress ;
- Certains génèrent des liens prêts à afficher, d'autres renvoient simplement des URLs.

```
wp_loginout()           // Génère automatiquement un lien Connexion ou Déconnexion selon l’état de l'utilisateur
wp_login_form()         // Affiche un formulaire de connexion complet et personnalisable
wp_register()           // Génère un lien Inscription ou Profil selon que l’inscription est activée et que l’utilisateur est connecté

wp_login_url()          // Retourne l’URL de la page de connexion
wp_logout_url()         // Retourne l’URL de déconnexion (avec nonce !)
wp_lostpassword_url()   // Retourne l’URL de récupération du mot de passe

is_user_logged_in()     // Renvoie true/false selon si un utilisateur est authentifié
```

En conclusion sur les template tags

- *Listing* principalement à titre informatif ;
- Certains doivent être **maîtrisés** et **utilisés régulièrement** (titres, **contenu**, métadonnées, **liens**, pagination, **taxonomies**, **utilisateurs**, etc.);
- L'important est de **connaître leur existence** pour éviter de **réinventer (en moins bien) ce que WordPress fournit déjà**.
- **Ne pas manipuler `$post` directement** : utiliser systématiquement les API (`the_post()`, `get_post()`, *template tags*). Cela garantit **cohérence**, **sécurité**, **compatibilité avec les extensions** et le *core*.

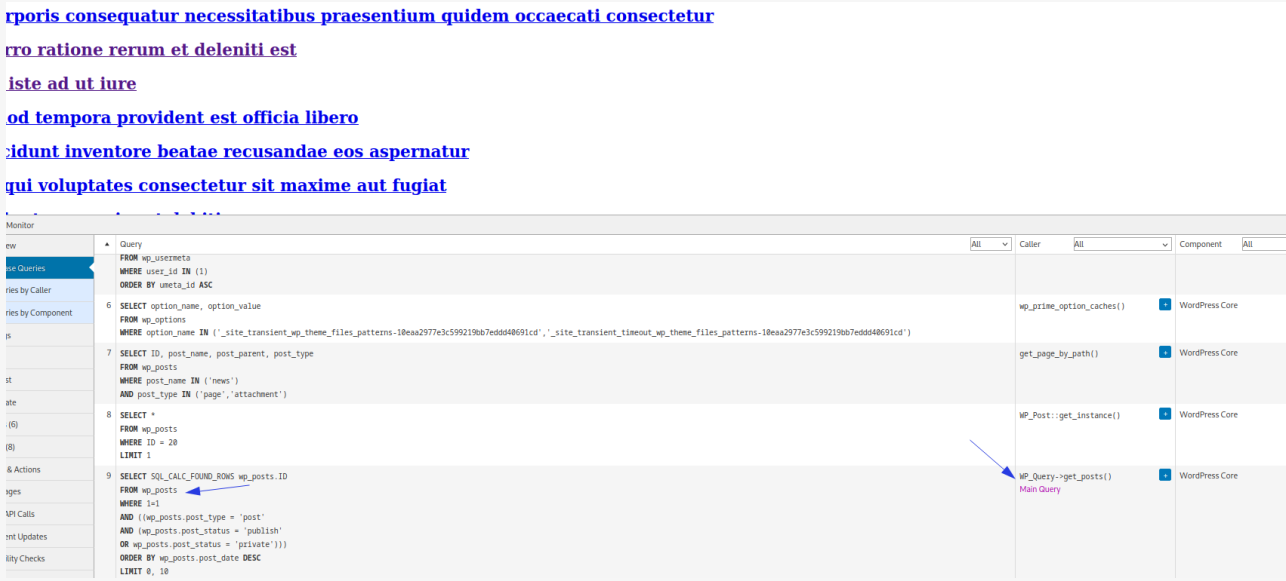
Requête principale (Main Query)

{.marp-bg-img}

Sous la *loop*, on trouve une instance de la classe `WP_Query`, en charge de la requête principal (*Main Query*).

La **requête principale** (faite à partir de l'analyse de l'URL, *contexte*) est réalisée via `WP_Query->get_posts()` et utilise l'objet global `$wp_query`.

> Faire un `var_dump($wp_query)` sur l'URL d'un post pour voir concrètement cet objet (le fameux *contexte*)



Créer ses propres requêtes

On peut créer ses propres requêtes pour récupérer du contenu soit :

- En créant une instance de `WP_Query` : la méthode **recommandée** ; **flexible**, **complète**, adaptée à la plupart des cas;
- La fonction `get_posts()` : utile pour des **requêtes simples** ; renvoie un tableau d'objets `WP_Post`. Ne modifie pas la requête principale.
- La fonction `query_posts()` : à éviter ; elle **modifie la requête principale** et peut casser la pagination.
- Sur le hook `pre_get_posts` : permet de modifier la requête principale **avant** son exécution (vu dans la suite sur les *hooks*).

Avec `WP_Query`

```php
$custom_query = new WP_Query([
    'posts_per_page' => 5
]);
```

- On crée une *nouvelle* instance, **sans modifier** `$wp_query` (l'instance principale utilisée par la requête principale)
- On peut créer des **requêtes aussi complexes que nécessaires** via son API ;

Custom Loop avec WP_Query

```php
// Création d'une instance WP_Query
$custom_query = new WP_Query([
    'posts_per_page' => 5,          // nombre de posts à récupérer
    'orderby'        => 'date',     // tri par date
    'order'          => 'DESC'      // ordre décroissant
]);

// Utiliser les mêmes méthodes que pour la main query !
if ($custom_query->have_posts()) :
    while ($custom_query->have_posts()) : $custom_query->the_post();
        ?>
        <h2><a href="<?php the_permalink(); ?>"><?php the_title(); ?></a></h2>
        <?php
    endwhile;
    // ! Important ! Repasse la main à la requête principale
    wp_reset_postdata();
endif;
```

Toujours appeler `wp_reset_postdata()` après une loop custom :

- **Restaure le contexte global** `$post` au post courant de la *Main Query*;
- Permet de **continuer à utiliser la boucle principale** ou d'autres template tags **sans interférences** (surtout si la loop custom a lieu au sein de la loop principale !)

Que se passe-t-il quand on crée une requête custom ?

- Lorsque l'on instancie l'objet (`new WP_Query($args)`), la requête SQL est **préparée** et **exécutée** (via la méthode `WP_Query::query()`) et le contenu est récupéré et stocké dans l'objet ;
- **La requête initialise également les conditional tags**, comme `is_home()` , `is_single()` , `is_admin()` , etc. Les conditional tags sont propres à chaque contexte (chaque objet `WP_Query`);
- Après cela, `have_posts()` et `the_post()` parcourent simplement le tableau des posts et mettent à jour le `$post` global (sans déclencher de nouvelle requête SQL)
- Le résultat d'une requête **est mis en cache** (dans un objet PHP), une référence future à la même requête ne génère pas de nouvelles requêtes SQL.

Avec `get_posts()`

```php
$posts = get_posts([
    'posts_per_page' => 5
]);
```

- Utile pour des **requêtes simples**.
- Retourne un tableau d'objets `WP_Post`.
- Ne modifie pas la requête principale (`$wp_query`).

Custom loop avec `get_posts()`

```php
$posts = get_posts([
    'posts_per_page' => 5
]);

foreach ($books as $post) {
    setup_postdata($post); //fait pointer $post global sur $post
    the_title(); // template tag travaille sur le post de get_posts
}
wp_reset_postdata(); // Restaure $post global pour Main Query
```

Pourquoi ne pas utiliser `query_posts()` ?

```php
//This is bad...
query_posts([
    'posts_per_page' => 5
]);
```

- Cette fonction **remplace** l'objet global `$wp_query` par une nouvelle requête ;
- **La boucle principale** (`have_posts()` , `the_post()`) **travaille donc maintenant sur ce nouveau contexte**.

Conséquences

- WordPress **a déjà construit** `$wp_query` , **cette première requête est donc inutilement exécutée** !
- Les filtres et hooks attachés à `pre_get_posts` ou `the_posts` **s'appliquent déjà à la requête principale**. En la remplaçant, certains comportements peuvent **être indéfinis**.

Bonnes pratiques

- **Requête personnalisée simple ou complexe** : créer une instance de `WP_Query` ;
- **Requête rapide et simple** : utiliser `get_posts()` ;
- **Modifier la requête principale** : passer par le hook `pre_get_posts` et ne **jamais écraser** `$wp_query` .

API de WP_Query

Explorons un peu l'API `WP_Query`. Pour écrire une requête il faut fournir au constructeur un tableau associatif PHP contenant l'ensemble des nos paramètres.

API de `WP_Query`, quelques exemples

```php
//Quelques exemples
$query = new WP_Query( array( 'category_name' => 'staff' ) );
$query = new WP_Query( array( 'category__not_in' => array( 2, 6 ) ) );

//Avec les métadonnées (Custom Fields !)
$args = array(
'meta_key' => 'color',
'meta_value' => 'blue',
'meta_compare' => '!='
);
$query = new WP_Query( $args );
```

> Bonne pratique : déclarer un tableau `$args` en amont et le passer ensuite au constructeur de `WP_Query`.

API de `WP_Query`, exemple plus complexe

```php
// Display posts that have meta key ‘color’ NOT LIKE value ‘blue’ OR
// meta key ‘price’ with values BETWEEN 20 and 100:

$args  = array(
 'post_type'  => 'product',
 'meta_query' => array(
  'relation' => 'OR',
  array(
   'key'     => 'color',
   'value'   => 'blue',
   'compare' => 'NOT LIKE',
  ),
  array(
   'key'     => 'price',
   'value'   => array( 20, 100 ),
   'type'    => 'numeric',
   'compare' => 'BETWEEN',
  ),
 ),
);
$query = new WP_Query( $args );
```

Pratique : WP_Query

Sur la page `/news`, **afficher** :

1. Uniquement les posts appartenant à la catégorie *Important* (à créer dans le dashboard);
2. Les posts n'appartenant **pas** à la catégorie *Important*;
3. Uniquement les posts d'un·e éditeur·ice précis (admin pour l'instant). Limiter à trois posts.
4. Les posts par ordre alphabétique sur *le titre*;
5. Les posts publiés *après* le 22/12/25 (exclus);
6. Une **liste paginée** de tous les posts. Il doit y avoir **9 posts par page**. **Utiliser** le template tag `paginate_links()` pour générer les URLs des pages paginées.

> Générer des posts avec le plugin *FakerPress* au besoin

Écrire directement ses requêtes SQL avec $wpdb

Utiliser l'objet global `$wpdb`, c'est **l'API la plus bas niveau offerte par WordPress à la base de données**.

```php
$results = $wpdb->get_results(
 $wpdb->prepare(
  "SELECT ID FROM {$wpdb->posts}
         WHERE post_type = %s AND post_status = 'publish'
         ORDER BY post_date DESC
         LIMIT %d",
  'post',
  5
 )
);

// Parcours et préparation des posts pour template tags
foreach ( $results as $row ) {
 $post = get_post( $row->ID );      // récupère l'objet WP_Post complet (avec ses métadonnées)
 setup_postdata( $post );           // initialise le $post global et les template tags
 the_title();
}

// Restaure le contexte global de la requête principale
wp_reset_postdata();
```

Éviter d'écrire directement des requêtes SQL brutes dans les templates

- Cette approche **ignore la boucle WordPress et les hook**s (pre_get_posts, the_posts, etc.);
- Récupération des données brutes de la table `wp_posts`, etc., **sans filtrage automatique**;
- Possibilité de recharger l'objet complet avec `get_post()`, mais cela **génère souvent des requêtes supplémentaires** et **demande plus de travail**;
- **Plus fragile** (dépendance explicite avec la structure actuelle de la base et des types de posts);
- **Moins idiomatique** que d’utiliser `WP_Query` ou `get_posts()`.

Privilégier l'usage des requêtes SQL customs **dans les plugins** !

Variables globales

```
global $post;
global $authordata;
global $current_user;
```

> Utile parfois de travailler sur les variables globales pour accéderm au contenu brut issu de la base avant transformation via les hooks (notamment par les plugins). En cas de besoin, **y accéder seulement en lecture**, pas en écriture !

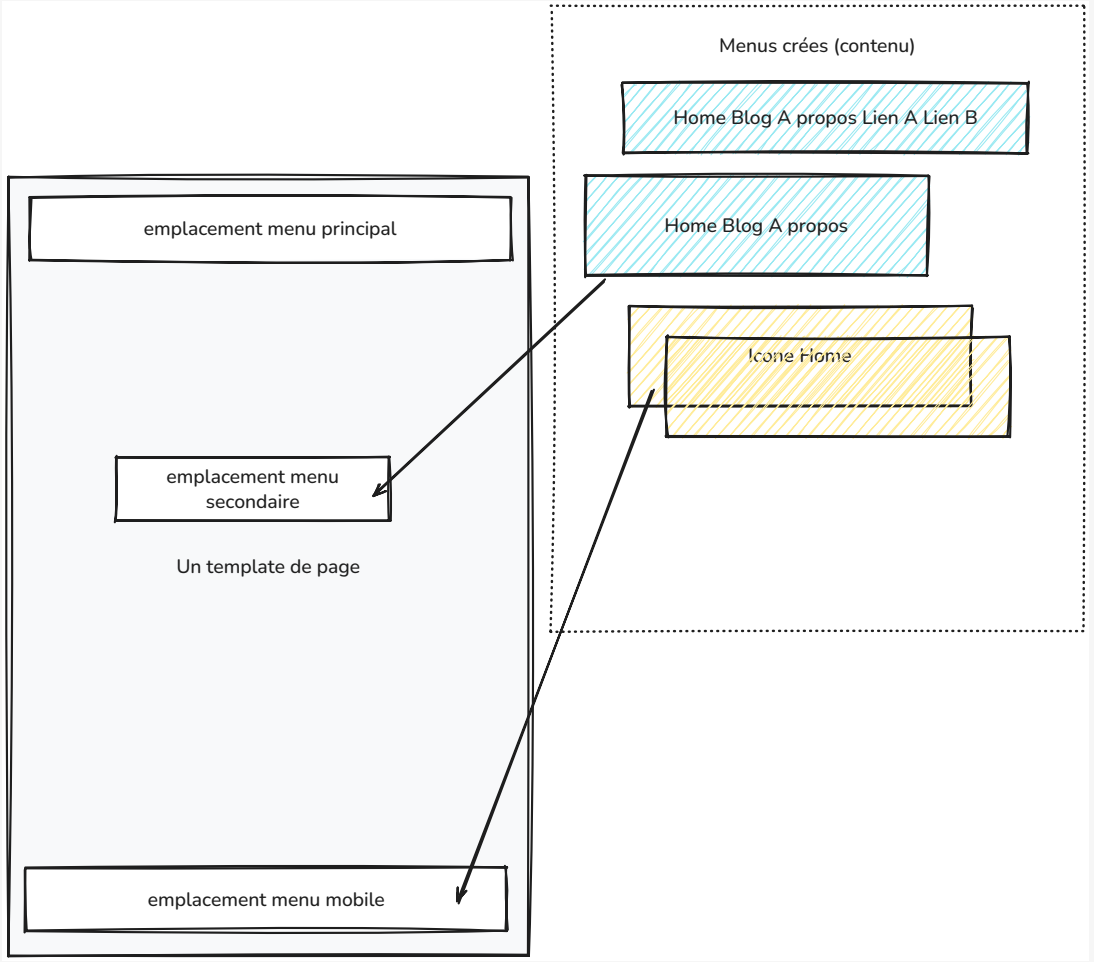
Gestion des menus (navigation)

{.marp-bg-img}

Système de gestion de menus :

1. Déclarer des **emplacements** (*locations*) de menus auprès du *core*;
2. **Générer le markup HTML de la location** dans les **templates** PHP (génération du markup) ;
3. **Créer** un menu (contenu) via l'admin ou du code PHP;
4. **Associer** le menu (contenu) à un emplacement.
5. Le contenu du menu sera récupéré en base et rendu à l'endroit indiqué dans le template.

> La distinction entre emplacement (location) et contenu est puissante et utile pour créer autant de menus que nécessaires et adaptés à chaque template ou plateforme ! Orienté *final user* (admin a le contrôle)



Créer des emplacements de menu et les charger auprès du core

Créer des emplacements de menu (*location*) avec les fonctions `register_nav_menus()` et `register_nav_menu()`.

```php
<?php
/**
 * Initialise le thème
 *
 * @return void
 */
function bp_register_menus() {
/**
 * Créer plusieurs emplacements
 */
register_nav_menus(
 array(
  'primary'   => __(
   'Main navigation menu',
   'BP'
  ),
  'secondary' => __(
   'Secondary menu in left sidebar',
   'BP'
  ),
 )
);
}
//Hook pour executer ce code au bon moment (vu dans la suite)
add_action( 'after_setup_theme', 'bp_register_menus' );
```

Parenthèse sur functions.php (

Ce fichier est automatiquement chargé par le core. S'il est mal aimé c'est qu'il est souvent (mal) utilisé pour implémenter toutes les fonctionnalités du site.

Quoi mettre dans ce fichier ?

- Toutes les fonctions propres **au bon fonctionnement du thème** : initialisation, support de fonctionnalités, déclarations de menus, de sidebars, **chargement des assets**, hooks propres au rendu, etc.
- Ne pas hésiter à répartir ces fichiers dans des modules (scripts) importés par `functions.php` au besoin. Par exemple dans `mon-theme/inc/`
- Tout ce qui ressemble à **une logique métier, une fonctionnalité du site ou un plugin** n'y a pas sa place.
- *Guide (idéal à viser, non à atteindre...)* : On devrait pouvoir **changer de thème sans casser les fonctionnalités du site web**.

La création des menus a donc sa place dans le `functions.php` du thème car c'est propre à ce thème !

Exemple : Structure du thème

```
mon-theme
    assets/
        main.min.css
        main.min.js
        img/
        fonts/
        ...
    inc/
        menus.php
        enqueue.php
        ...
    functions.php
    front-page.php
    header.php
    ...
    parts/
```

Exemple : functions.php typique

```php
<?php
require_once get_template_directory() . '/inc/enqueue.php';
require_once get_template_directory() . '/inc/menus.php';

function mon_theme_setup() {
    //Explication sur cette fonction slide suivante
    add_theme_support(
        'html5',
        array(
            'comment-list',
            'comment-form',
            'search-form',
            'gallery',
            'caption',
            'style',
            'script'
        )
    );
    //Ajoute des options d'admin pour appareance du site (Apparence/*)
    add_theme_support('custom-background');
    add_theme_support('custom-header');
    add_theme_support('custom-logo');
}
add_action('after_setup_theme', 'mon_theme_setup');
```

Fonctionalités de l'admin du thème avec add_theme_support

La fonction `add_theme_support` permet de **configurer des fonctionnalités supportées par le thème et son administration dans le dashboard**: images mises en avant, menus, HTML5, custom-logo, etc. Voir la liste des options dans la doc.

) Fin de la parenthèse sur functions.php, retour aux menus

Définir l'emplacement dans les templates

Dans un template, pour générer le markup du menu, utiliser le *template tag* wp_nav_menu() :

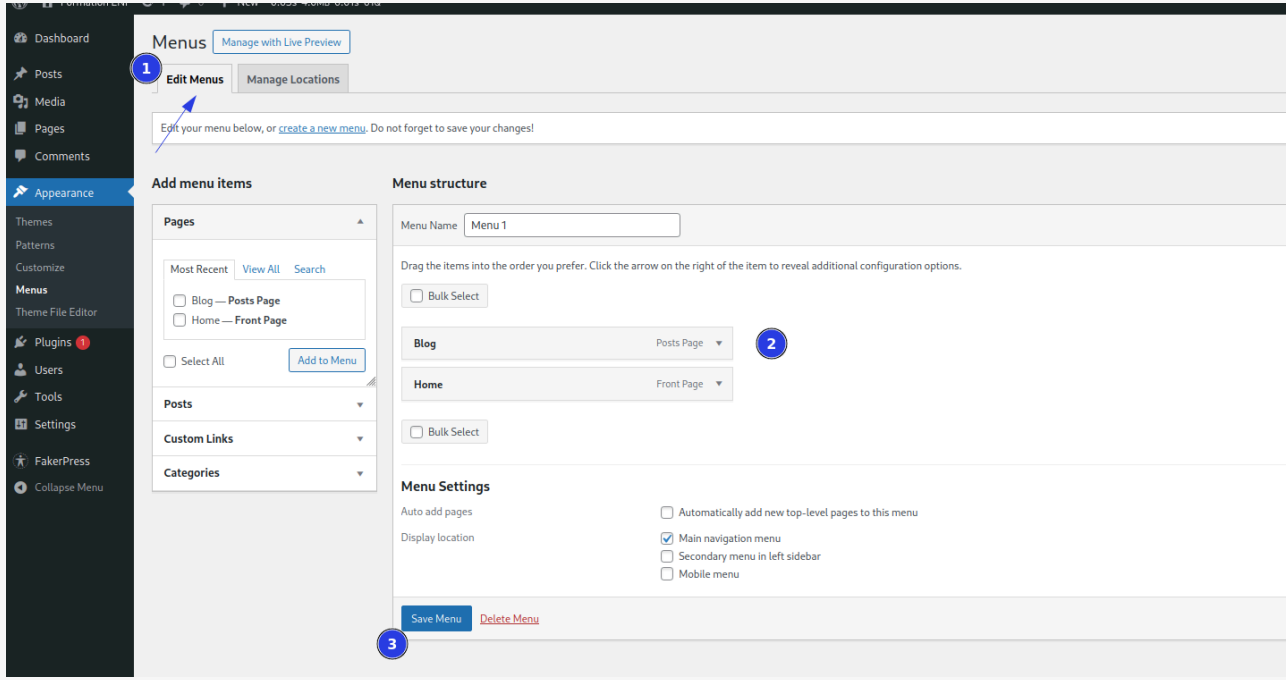
```
//Dans header.php
 <header>
  <?php
  $args = array(
   'theme_location' => 'primary',
   'menu_class'     => 'nav-menu',
  );
  wp_nav_menu( $args ); // va chercher le menu (contenu) associé et produire tout le markup HTML
  ?>
 </header>
```

> Comme la plupart des template tags, `wp_nav_menu()` accepte un **grand nombre d'arguments pour customiser le HTML généré**.

> Il est également possible d'utiliser des hooks *filter* pour personnaliser le contenu des menus (souvent suffisant), ou même d'écrire sa propre classe (`Walker_Nav_Menu`) pour contrôler *entièrement* le HTML généré.

Créer le contenu du menu (Gestion de contenu, éditorial)

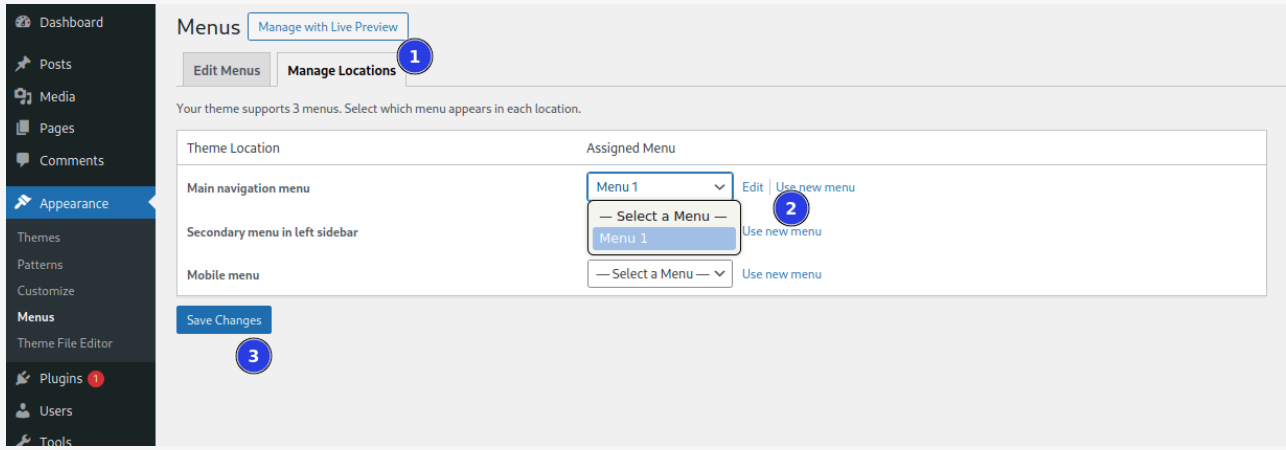
{.marp-bg-img}



Placer le contenu dans l'emplacement prédéfini (Gestion de contenu, éditorial)

{.marp-bg-img}

> Association Menu(contenu) et Location



Exemples de personnalisation des menus

```php
//Afficher un menu différent pour un user connecté ou non en changeant de location dynamiquement
wp_nav_menu( array(
 'theme_location' => is_user_logged_in() ? 'logged-in-menu' : 'public-menu'
) );

//Préparer des emplacement de menu pour différentes tailles d'écran
wp_nav_menu( array(
 'header' => 'Main menu',   //Masqué sur mobile (via media query CSS)
 'footer' => 'Mobile menu', //Masqué sur desktop (via media query CSS)
) );

//Ajouter une classe à chaque élément du menu (via le hook 'nav_menu_css_class')
add_filter('nav_menu_css_class', function($classes, $item, $args, $depth) {
    $classes[] = 'ma-classe';
    return $classes;
}, 10, 4);
```

Pratique : Créer un menu pour le site

1. **Créer** un emplacement de menu principal dans le template `header.php` du thème;
2. **Créer** un menu;
3. **Associer** le menu à l'emplacement;
4. **Tester**. Inspecter le HTML généré.
5. En profiter pour compléter le template de la *front-page*. Elle doit afficher le contenu de la page *Home*.
6. En utilisant la documentation, **trouver** un moyen de placer le menu dans un élément HTML `<nav>` ;

Shortcodes

L’API des shortcodes sert à **insérer**, **dans le contenu d’un article ou d’une page**, des **fonctionnalités dynamiques générées par du code PHP** via une syntaxe compacte de type `[tag attribut=valeur][/tag]`.

Très pratique pour le *final user*. Les shortcodes sont adaptés pour :

- **Injecter** du contenu dynamique **sans exposer/écrire de HTML/PHP** ;
- Fournir au *final user* une **API simple et contrôlée** pour des **composants**, à la syntaxe simple et facilement **documentable** (ex. Formulaire)
- **Encapsuler la logique métier** (règles de gestion pour la production du contenu du composant) tout en laissant l’auteur la manipuler seulement en insérant une chaîne de caractère courte.

API Shortcodes

Créer un shortcode avec la fonction `add_short_code()` :

```php
//tag 'spanc', fonction exécutée lors du traitement du shortcode, doit retourner le contenu
add_shortcode('spanc', function($atts = [], $content = null) {

    //Ne pas autoriser l'interprétation de shortcode pour certains roles
     if (!current_user_can('edit_posts')) {
        return '';
    }

    //Fusionner attributs fournis avec attribut par défaut.
    //Logique de validation des attributs à implémenter vous même !
    $atts = shortcode_atts(['color' => 'red'], $atts);

    return '<span style="color:' . esc_attr( $atts['color'] ) . '">' . esc_html( $content ) . '</span>';
});
```

Utiliser un shortcode

Utiliser le shortcode n'importe où (page, post) :

```
[spanc color="blue"]Texte[/spanc]
```

Il faut que la partie du template appelle `do_shortcode($shortcode_string)`. Ce qui est fait automatiquement dans les template tags `the_content()` et `the_excerpt()`.

> Un shortcode est une **fonctionnalité indépendante du thème**. Elle devrait donc être définie dans un plugin.

Hooks

L'API **la plus importante** de WordPress ? Certainement!

Hooks : actions et filters

C'est l'API la plus utilisée pour étendre le core de WordPress et créer votre site sur mesure.

Les *Hooks* (crochets), comme leur nom l'indique, sont des **moyens d'executer du code à des moments spécifiques du cycle de vie de l'application WordPress**.

> C'est un mécanisme classique utilisé par les frameworks pour qu'ils puissent charger et utiliser *notre code*.

Les hooks permettent donc de modifier des comportements par défaut de WordPress.

C'est notamment l'API principale utilisée par les *plugins* pour interagir avec le core (*plugin architecture*)

Les hooks concrètement

Un hook est un appel de fonction PHP avec plusieurs arguments :

```
<?php add_action($tag, $my_function_to_execute, $priority, $args)>
<?php add_filter($tag, $my_function_to_execute, $priority, $args)>
```

Hooks : actions et filtres

Il existe **deux** types de *hooks* :

- Les **actions**, pour exécuter du code ;
- Les **filtres**, pour modifier des données ;

Actions

- Les actions **ne retournent rien**, elles **exécutent** du code (*procédures*);
- Les *hooks actions* sont **déclenchés lors d'évènements émis au cours du cycle de vie de l'application WordPress**. Par exemple, un évènement `publish_post` est émis par le *core* lorsqu'un $post$ est publié ;
- On peut également émettre n'importe quand ces évènements en appelant la fonction `do_action('event')` ;

Exemple

Enregistrer une procédure (fonction PHP) lors de la publication d'un post :

```php
add_action('publish_post', function($post_id) {
    // code exécuté quand un post est publié
    error_log("Post $post_id publié !");
});
```

> On enregistre ici une fonction anonyme

Exemple : Log lors du changement de status d'un post

Enregistrer une procédure (fonction PHP) lors de la publication d'un post

```php
/**
 * Log chaque publication d'article
 *
 * @param string  $new_status L'id du post qui a changé de status.
 * @param string  $old_status L'ancien status.
 * @param WP_Post $post Le post concerné.
 * @return void
 */
function eni_log_on_post_change_status( string $new_status, string $old_status, WP_Post $post ) {
 $author_id = $post->post_author;
 if ( WP_DEBUG ) {
  error_log( "Post {$post->ID} Nouveau status : $new_status - Ancien status : $old_status. Auteur : $author_id. Titre : {$post->post_title}" );
 }
}

add_action( 'transition_post_status', 'eni_log_on_post_change_status', 10, 3 );
```

Ici `enis_log_on_post_change_status` est une *string* qui référence la fonction (type callable). Va être appelée par WordPress avec la fonction PHP `call_user_func()`.

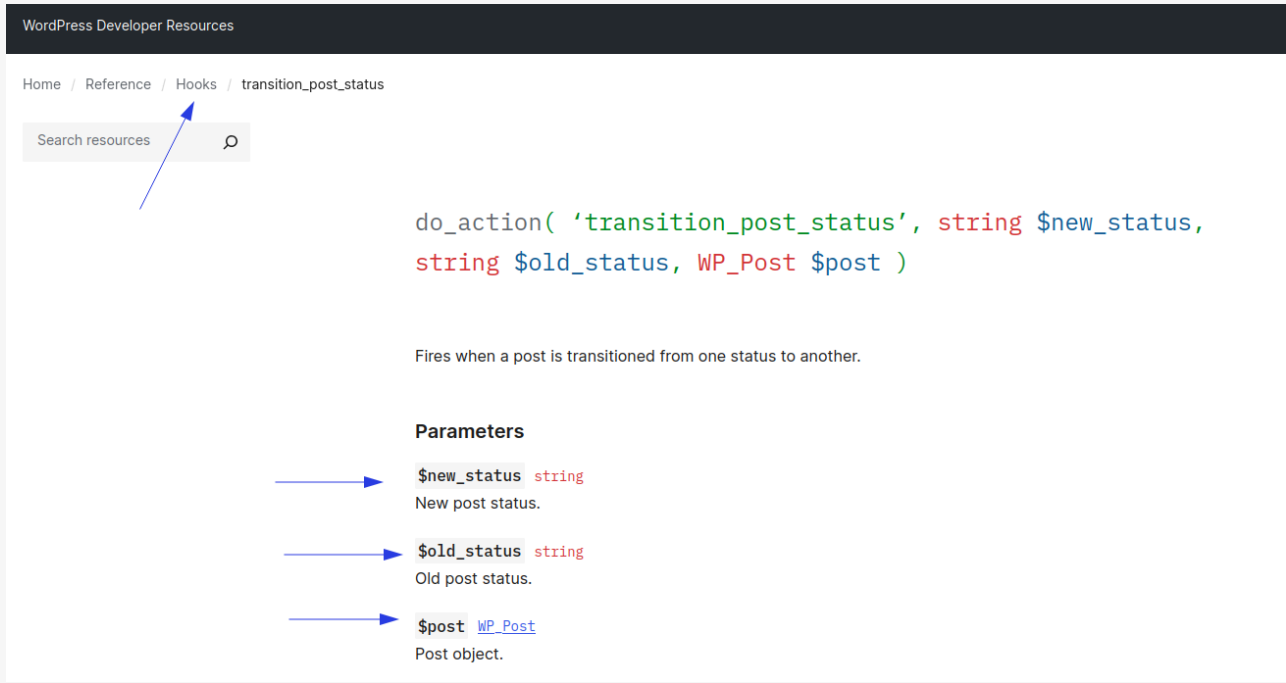
> On a déjà appris deux hooks !

Lire la documentation d'un Hook

{.marp-bg-img}

Par exemple, l'évènement `transition_post_status` :

- **Description** : *Fires when a post is transitioned from one status to another* ;
- **Nom du hook** (évènement) : `transition_post_status`
- **Arguments et nombre** (`3`) reçus par la *callback* (fonction enregistrée) :
  - `string $new_status` ;
  - `string $old_status` ;
  - `WP_Object $post` ;
- `do_action()` : c'est un hook de type **action**.



Utiliser la documentation d'un hook pour écrire sa callback

- La fonction va recevoir **3 arguments dans l'ordre indiqué** ;
- Indiquer les arguments dans la signature :

```
function enis_log_on_post_change_status( string $new_status, string $old_status, WP_Post $post ) {
    ...
}
```

Enregistrer sa *callback*

- Pour une action, utiliser `add_action()` ;
- *Callable* (nom sous forme de chaîne de caractère ici) : `eni_log_on_publish_post` ;
- Priorité : Quand exécuter cette callback dans la **pile d'appel** (**10 par défaut**). Discuté slide suivante;
- Nombre d'arguments : `3` .

```php
add_action( 'transition_post_status', 'eni_log_on_publish_post', 10, 3 );
```

Pile d'appels et priorité

Sur un même *hook*, **plusieurs fonctions peuvent être enregistrées** (celles du *core*, les vôtres, celles de plugins). Ces fonctions forment **une pile**. Quand un *hook* est exécuté (`do_action()`), toute les fonctions présentes dans la pile sont **exécutée dans un ordre précis**.

L'ordre de la pile d'appel est déterminée par :

- La **priorité** (paramètre optionnel de type `int`, **défaut = 10**), **plus petit = plus tôt**;
- L'**ordre d’enregistrement** pour les callbacks avec **même priorité**;

Exemple de pile d'appels

On enregistre 3 *callbacks* sur le hook `init` :

```php
add_action('init', 'callback2', 10);
add_action('init', 'callback3', 10);
add_action('init', 'callback1', 5);
```

Ordre d'exécution :

- `callback1` ;
- `callback2` ;
- `callback3` ;

Autres méthodes pour enregistrer des callbacks

En PHP, les callables peuvent être créés de différentes façons :

- String contenant le nom de la fonction (ce que l'on vient de voir);
- Un **tableau** contenant le nom de la classe ou un **objet** à l'index 0 et le nom de la méthode à l'index 1;
- Une **Closure**;

Enregistrer la méthode d'un objet comme callback

```php
class ENIS_Logger {
 public function log_on_post_change_status( string $new_status, string $old_status, WP_Post $post ) {
       //...
 }
}
$logger = new ENIS_Logger();
// Appel avec une instance d'ENIS_Logger
add_action( 'transition_post_status', array( $logger, 'log_on_post_change_status' ), 10, 3 );
```

> Utile si on veut utiliser le paradigme orienté objet, par exemple dans ses plugins.

Enregistrer la méthode statique d'une classe comme callback

```php
<?php
class ENIS_Logger {

    public static function log_on_post_change_status(string $new_status, string $old_status, WP_Post $post) {
       //...
    }
}

// Appel avec nom de la classe
add_action('transition_post_status', ['ENIS_Logger', 'log_on_publish_post'], 10, 3);
```

> Si l'on utilise le nom de la classe, il faut que **la méthode soit statique** !

Enregistrer une callback sous forme de closure

```php
$some_data = 'Cette donnée est capturée par la closure avec la clause use';
// Closure (capture son environnement)
$log = function ( string $new_status, string $old_status, WP_Post $post ) use ( $some_data ) {
 $author_id = $post->post_author;
 if ( WP_DEBUG ) {
  error_log( "$some_data" );
 }
};
add_action( 'transition_post_status', $log, 10, 3 );
```

Executer une fonction *namespacée* sur un Hook

Si la fonction est namespacée (car on développe un plugin par exempl !), il faut passer le **nom complètement qualifié de la fonction** (ou de la classe).

```php
namespace MonPlugin;

function my_callback() {
    error_log('Hook déclenché !');
}

// Nom complètement qualifié. Utiliser la constante _NAMESPACE_ !
add_action('init', __NAMESPACE__ . '\\my_callback');
```

> Attention à bien échapper le caractère \ !

Executer une méthode d'une classe *namespacé* sur un Hook

```php
<?php
namespace MonPlugin;

class Logger {
    public static function log() {
        error_log('Hook déclenché via méthode statique !');
    }
}

// Nom complètement qualifié. Utiliser la constante _NAMESPACE_ !
add_action('init', [__NAMESPACE__ . '\\Logger', 'log']);
```

Les Hooks filtre (filters)

- Les filtres (`apply_filters`) permettent de **modifier ou enrichir une valeur à différents moments du cycle WordPress**;
- Les filtres **retournent toujours une valeur** !
- Ils sont appelés avec une valeur initiale, et **chaque callback peut la transformer avant de la retourner**
- **Les filtres sont composés** : ils sont exécutés à la chaîne, chaque sortie d'un filtre arrive en entrée du suivant (philosophie des *pipe* UNIX !)
- Comme pour les actions, l'**ordre d'application** des filtres est **défini par la pile d'appel** (priorité, ordre d'enregistrement)

Les filtres sont typiquement utilisés pour modifier du contenu avant de l'enregistrer en base de données, d'en générer le code HTML côté public (page web) ou côté admin (modifier les pages d'administration !), etc.

Forme (même que action)

Pour enregistrer un filtre, utiliser la fonction add_filter() :

```
add_filter( string $hook_name, callable $callback, int $priority = 10, int $accepted_args = 1 );
```

Pour appliquer un filtre (et déclencher l'appel de toutes les fonctions présentes dans sa pile !), utiliser la fonction apply_filters() :

```
apply_filters( string $hook_name, mixed $value, mixed ...$args ): mixed;
```

Enregistrer et utiliser des filtres

```php
add_filter('my_filter', fn($v) => $v . ' B', 10);
add_filter('my_filter', fn($v) => $v . ' C', 10);
add_filter('my_filter', fn($v) => $v . ' A', 5);

// Résultat : "Ordre ? A B C"
echo apply_filters('my_filter', 'Ordre ?');
```

Exemples classiques

```php
// Avant enregistrement en base : pre_post_title
add_filter('pre_post_title', function($title) {
    return strtoupper($title); // le titre sera enregistré en majuscules
});
// Avant affichage HTML côté public du contenu d'un post
add_filter('the_content', function($content) {
    return $content . '<p>Texte ajouté automatiquement</p>';
});

// Côté admin : Ajouter une colonne custom à un tableau d'administration des posts

// Ajouter la colonne sur 'manage_posts_columns'
add_filter('manage_posts_columns', function($columns) {
    $columns['new_col'] = 'Nouvelle colonne';
    return $columns;
});

// Remplir la colonne (action) pour chaque post sur 'manage_posts_custom_column'
add_action('manage_posts_custom_column', function($column_name, $post_id) {
    if ('new_col' === $column_name ) {
        echo 'Valeur pour le post ' . $post_id;
    }
}, 10, 2);

// Appliquer un filtre (retourne une valeur)
$my_custom_title = apply_filters('the_title', '  My Custom Title (tm)  ');
```

Créer vos propres hooks !

On peut **créer ses propres hooks** ! **Et c'est recommandé** !

Cela permet d'utiliser la même logique que le core et de rendre son code plus modulaire et extensible. Pour cela il suffit de :

1. **Créer** son propre évènement en y enregistrant des fonctions avec `add_action()` ou `add_filter()` ;
2. **Déclencher** les évènements et leurs piles d'appels avec `do_action()` ou `apply_filters()`.

```php
<?php
add_action('monplugin_post_save', function($post_id, $post){
    //Action 1
}, 10, 2);
add_action('monplugin_post_save', function($post_id, $post){
    //Action 2
}, 10, 2);
//Etc...

//Déclencher la pile d'appels quelque part dans votre application WordPress (thème, plugins)
do_action('monplugin_post_save', $post_id, $post);
```

Précautions

- Lire la documentation du core pour chaque hook : WP Hook Reference;
- **Tester** dans le contexte pour s'assurer que la **priorité** et les **arguments** sont corrects;
- **Ne pas utiliser des noms de hook natifs pour vos hooks customs (collision)**. Pour cela, comme pour vos fonctions PHP, **préfixer tous vos hooks par votre nom vendor** !

Connaître les évènements fournis par le core

Concernant les hooks natifs, fournis par le core de WordPress, **il faut les connaître**. Ils correspondent à des **moments clefs** dans le cycle de vie de l'application (traitement d'un requête), du chargement du *core* à la production de la réponse HTTP.

- Utiliser la base de données des Hooks développée et maintenue depuis des années par Adam R. Brown;
- Inspecter l'ordre des hooks et hooks déclenchés avec le plugin Query Monitor.

> La connaissance des hooks les plus utilisés et utiles vient vite.

Hooks principaux au cours du cycle de vie de l'application WordPress 1/3

Les hooks principaux, *au cours du temps*.

1. **Chargement initial**
    - `muplugins_loaded` : Tous les must-use plugins chargés.
    - `plugins_loaded` : Tous les plugins classiques chargés.
    - `after_setup_theme` : Après le chargement du thème et son fichier `functions.php`.
    - **`init`** : **WordPress initialisé** (régénération des rôles, taxonomies, scripts, types de posts). **Très utilisé**
    - `wp_loaded` : Tout WordPress chargé, mais avant l'envoi des headers.
2. **Traitement des requêtes**
    - `parse_request` : La requête HTTP est interprétée.
    - `send_headers` : Les headers HTTP sont sur le point d'être envoyés.
    - **`wp`** : La requête principale (WP_Query) est construite. **Très utilisé**
    - **`template_redirect`** : Avant le chargement du template. **Utile pour les redirections !**

Hooks principaux au cours du cycle de vie de l'application WordPress 2/3

3. **Rendu du thème**
    - `get_header` : Juste avant l’inclusion du fichier header.php.
    - **`wp_head`** : Dans la balise . **Utile pour le SEO (meta)**
    - **`wp_enqueue_scripts`** : Juste avant l'inclusion des feuilles de styles et des scripts JS. **Très utilisé pour charger vos assets !**
    - `loop_start` : Début de la Loop principale.
    - `the_post` : À chaque post de la Loop.
    - `loop_end` : Fin de la Loop.
    - `get_footer` : Avant l’inclusion du fichier footer.php
    - `wp_footer` : Juste avant .

Hooks principaux au cours du cycle de vie de l'application WordPress 3/3

4. **Pages d’admin**

- **`admin_init`** : Chargement de l’admin (options et scripts). **Très utilisé**
- `admin_menu` : Pour ajouter des pages de menu dans le back-office. **Très utilisé**
- `admin_bar_menu` : Modifier la barre d’admin en haut (admin bar).
- `set_current_user` : Initialisation de l’utilisateur courant. **Très utilisé**
- `update_post_meta` : Déclenché après mise à jour d’un meta post. **Très utilisé**
- **`add_meta_boxes_{posttype}`** : Ajouter des meta boxes pour tous types de post. **Très utilisé**
- **`current_screen`** : Déclenché après l’instanciation de l’écran courant **Utilisé**
- **`pre_get_posts`** : Modifier la requête principale des posts avant exécution **Très très utilisé**
- `pre_get_terms` : Modifier la requête de récupération des termes (taxonomy) **Très utilisé**
- `admin_notices` : Afficher des notifications dans l’admin **Très utilisé** (notifications, par plugins)
- `all_admin_notices` : Afficher toutes les notifications
- `admin_enqueue_scripts` : Pour charger CSS/JS custom côté admin
- `shutdown` : dernier hook disponible avant fermeture.

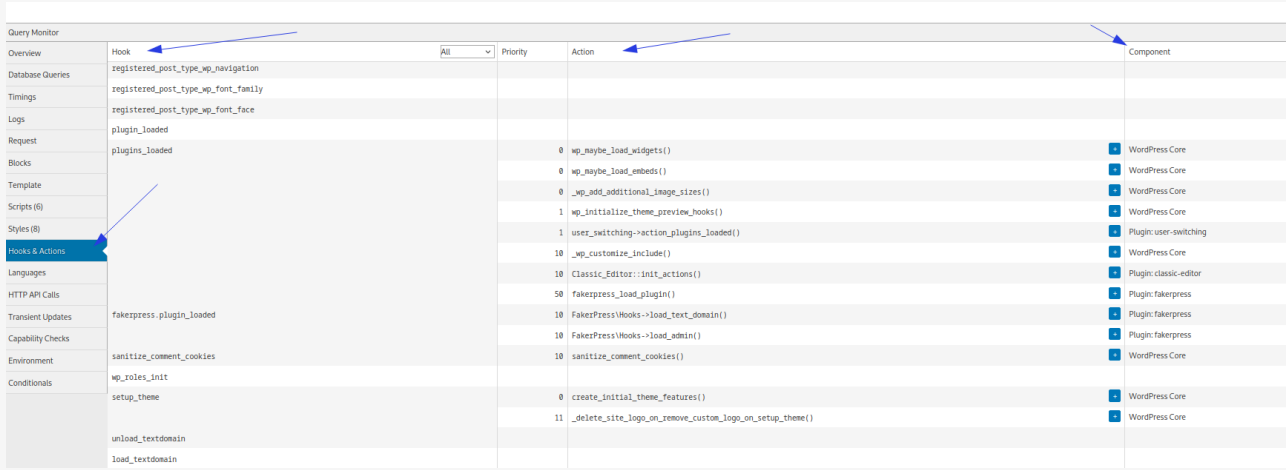
Inspecter les hooks

{.marp-bg-img}

Avec le plugin Query Monitor :

- Utile pour débug;
- Utile pour **découvrir** et **apprendre** d'autres hooks;
- Tester côté public et **côté admin** sur différents écran;
- **Tous ces hooks sont des points de personnalisation de WordPress !**

> Il y en a **beaucoup**. S'apprennent à l'usage. Avec l'expérience, le but est d'essayer de trouver le hook le plus *précis* , pour éviter des comportement inattendus dans le futur.



Hook actions les plus utilisés

```
wp_head
wp_footer
wp_loaded
admin_head
init
admin_init
admin_menu
user_register
set_current_user
publish_post
save_post
pre_get_posts
create_category
add_meta_boxes_{posttype}
comment_post
switch_theme
wp_enqueue_scripts
admin_enqueue_scripts
template_redirect
login_enqueue_scripts
transition_post_status
template_redirect
```

Hook filters les plus utilisés

```
the_content
the_title
the_permalink
post_class
body_class
wp_title
login_redirect
upload_mimes
```

Hooks liés aux users

```
user_register
wp_login
wp_logout
profile_update
```

Développer avec les hooks (*Hook first* !)

- **Utiliser au maximum les Hooks** ! C'est *the WordPress way*;
- Quand vous développez une fonctionnalité, personnalisez de l'existant, **demandez-vous** (ou à une IA générative) : **existe-t-il un hook pour ça ?**
- En créant **vos propres hooks**, vous permettez à d'autres développeurs (y compris vous dans le futur !) ou plugins de **surcharger votre code sans le modifier directement**. Vos fonctionnalités restent **isolées** et **personnalisables** à volonté.

Exemple : Charger vos assets (public) via le hook `wp_enqueue_scripts`

```php
<?php
add_action('wp_enqueue_scripts', function() {
    // CSS
    $css_version = '0.1';
    wp_enqueue_style(
        'mon-plugin-front-css',
        get_stylesheet_directory_uri() . '/styles/main.min.css',
        [],
        $css_version //Toujours mettre une version à incrémenter pour forcer la recharge du cache côté client !
    );

    // JS
    $js_version = '1.2';
    wp_enqueue_script(
        'mon-plugin-front-js',
        get_stylesheet_directory_uri(__FILE__) . '/js/main.min.js',
        ['jquery'], // dépendances
        $js_version, //Toujours mettre une version à incrémenter pour forcer la recharge du cache côté client !
        array(
            'strategy'  => 'defer', // indiquer au navigateur que le script doit être exécuté après l'analyse du document
            'in_footer' => true, // où placer la balise script (ou utiliser async pour une exécution asynchrone.)
        )
    );
});
```

> Conseil : Utiliser le hook `wp_head` pour précharger des assets critiques comme des fonts avec `<link rel="preload">`.
>
> Indique au navigateur de charger la ressource en priorité.

Exemple : Charger des assets CSS et JS côté admin sur une page custom

```php
add_action('admin_enqueue_scripts', function($hook) {
    // Charger seulement sur une page spécifique si besoin
    if ($hook !== 'toplevel_page_mon_plugin') {
        return;
    }

    // CSS
    wp_enqueue_style(
        'mon-plugin-admin-css',
        plugin_dir_url(__FILE__) . 'assets/admin.css',
        [],
        '1.0.0'
    );

    // JS
    wp_enqueue_script(
        'mon-plugin-admin-js',
        plugin_dir_url(__FILE__) . 'assets/admin.js',
        ['jquery'], // dépendances
        '1.0.0',
        true // placer en footer
    );
});
```

Exemple : Modifier l’ordre ou filtrer les posts dans un template

Pour **altérer** la requête principale (*Main Query*) **avant** qu'elle ne soit exécutée, **utiliser le hook** `pre_get_posts` :

```php
function monplugin_modify_home_query($query) {
    if ($query->is_main_query() && !is_admin() && $query->is_home()) {
        $query->set('post_type', array('post', 'movie'));
        $query->set('posts_per_page', 10);
    }
}
add_action('pre_get_posts', 'monplugin_modify_home_query');
```

Pour s'assurer d'exécuter ce hook dans un template :

- *Toujours* vérifier `$query->is_main_query()` pour ne pas affecter les requêtes secondaires (widgets, boucles custom).
- *Toujours* exclure l’admin avec `!is_admin()`.

Conclusion

Nous avons vu dans cette séquence les API clés pour développer sur WordPress :

- **Template hierarchy** pour développer efficacement les templates requis sur mesure;
- La **Loop WordPress** pour lier templates et modèle (base de données)
- Comment créer ses propres **loop customs**, boucles sur mesure supplémentaires selon les besoins.
- Comment utiliser **WP_Query** pour créer des requêtes custom aussi complexes que nécessaires
- Les **shortcodes** pour insérer du contenu dynamique et réutilisable dans les pages ou articles.;
- Les **menus** : locations et contenu des menus administrable;
- Utiliser les **Hooks** : **l'API la plus importante de WordPress** pour étendre et personnaliser le comportement du site
- Savoir comment **utiliser** et **identifier** d'autres hooks disponibles (doc, monitoring)
- Connaître les **hooks principaux fournis par le core** et leur usage.
- **Hooks custom** : **créer ses propres hooks** pour rendre son code extensible et modulable.

Pratique : Thème fil rouge

Utiliser ces APIs pour continuer d'implémenter notre application WordPress pour la Bibliothèque.