"ZPR PWr - Integrated Development ProgramWrocław University of Technology"

Wrocław University of Science and Technology

Advanced Web Technologies

Lab

Subject: Plugin implementation for WordPress

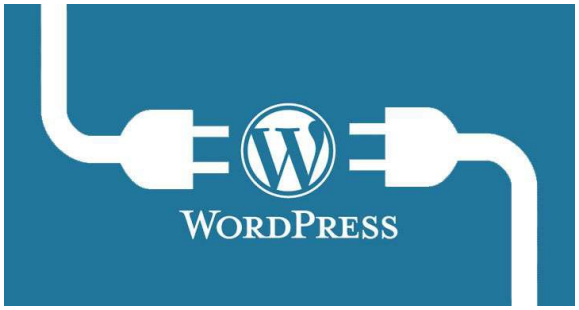
Prepared by: mgr Piotr Jozwiak

Date: July 2020

Number of hours: 2 hours

Table of Contents

Entry



In the next lab, we'll explore how to extend the standard WordPress functionality with your own plugin. The most important reason to create your own plugin is that it allows you to separate your own code from the core WordPress code. If something goes wrong with our plugin, the rest of the site will generally still be there
function properly. Changing WordPress source files is probably the worst thing you can do. In addition to the possibility of damaging our system, we cut ourselves off from the possibility of updating the system to a new version in this way, because the update will automatically erase our change in the code.

Plugins are plain PHP scripts. As we will see, implementing new functionality to WP is not such a difficult process. Of course, at least basic knowledge of the PHP language and the functioning of the WordPress system is required. In this lab, I assume that the Student already has basic knowledge of PHP. The rest of the necessary messages will be described in the material below.

Introduction to WordPress Plugins

WordPress pluginis a standalone set of code that enhances and extends the functionality of WordPress. Using any combination of PHP, HTML, CSS, JavaScript/jQuery, the plugin can add new features to any part of your website, including the admin control panel. We can modify the default WordPress behavior or remove unwanted behavior altogether. Plugins allow you to easily customize and personalize WordPress to your needs.

Since WordPress plugins are self-contained, they do not physically alter any of the core WordPress source files. They can be copied and installed in any WordPress installation. An alternative (and strongly discouraged) way to make changes to WordPress is to save new functions in the WordPress functions.php file, which is stored in the folder
`/wp-include/`or the functions.php file that is part of the theme. This has many potential problems.

WordPress and its themes are updated regularly. If we are not using a child WordPress theme, when functions.php is overwritten by an update, the new code will be deleted and we will have to rewrite it. If we're writing multiple functions and one of them has an undebugable error, we may need to replace the entire file with the original one, removing all the changes we've made. If our functions are removed from the file, our site will generate PHP errors indicating missing functions.

Plugins are never automatically overwritten or deleted when installing WordPress updates. If there are coding errors in the plugin, it can usually be simply deactivated in the admin control panel during repair. If there is a serious error in a plugin, WordPress will sometimes automatically deactivate it, allowing your site to continue to function.

What are WordPress Hooks?

WordPress plugins interact with the core code using so-called Hooks, i.e. certain anchor points. There are two types of hooks.

1. **`Action hooks`**(usedfor adding/deleting functions)
2. **`filterhooks`**(usedto modify data returned by functions)

Action Hooks

When a user visits any WordPress-based website, in the background the server executes a series of PHP functions (called actions) at various call points that are attached to various hooks called Action Hooks. Using the Action Hooks provided by WordPress, you can add your own functions to the list of actions that are triggered when a certain event is triggered, and you can also remove existing functions from anywhere. Action Hooks determine when actions are called in the process of generating the page's HTML code. For example, before the closing </head> tag, an Action Hook called wp_head() is called, which in turn calls all actions hooked to wp_head().

Action Hooks are contextual - some are called on every page of our site, others are only called when viewing the Admin Control Panel, and so on. A full list of available Action Hooks along with a description of the context in which they are called can be found on the websitePlugin API/Action Reference.

Attaching your own Action function to Action Hook

In order to attach its function to the selected anchor point, the plugin must execute a WordPress engine function called add_action(). This function expects two parameters. Let's analyze the following example to learn how this functionality works:

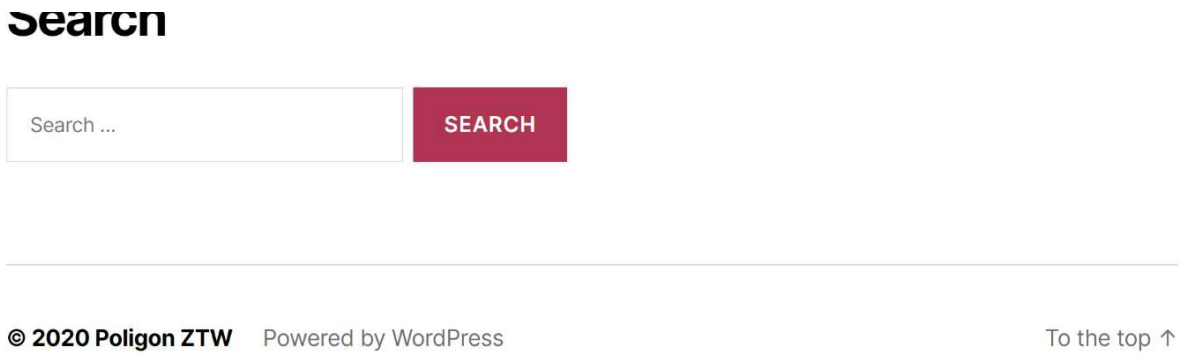
```
// Hook to the 'init' action, which is called after WordPress is finished loading
 the core code
add_action('init','add_Cookie');

// Set a cookie with the current time of
dayfunctionsadd_Cookie() {
  setcookie("last_visit_time",date("r"),time()+60*60*24*thirty,"/");
}
```

- The first required parameter is the name of the Action Hook we want to connect to
- The second required parameter is the name of the function we want to run
- The third parameter (optional) is the priority of the function we want to run. We can connect any number of different functions to the same Action Hook. This parameter allows you to arrange the actions in the desired order. The default priority is 10, placing a custom function after any of the built-in WordPress functions.
- The fourth parameter (optional) is the number of function arguments. The default value is 1.

The next example is to display some text after generating a standard page footer. To do this, we'll pin our action to an Action Hook called wp_footer(). This will call our function every time the WordPress generator is just before the closing tag
</body>. We'll attach a function called mfp_Add_Text(). The only question is where to enter our code? For the moment, for simplicity, we will modify the current template. We will make changes in the already mentioned functions.php file. To do this example, it is not even required to look at the source files stored on the serverWWW.We will use the editor built into the WordPress administration panel. To open it, let's go to the admin panel. Then select Appearance -> Theme Editor from the side menu. After a warning about the consequences of modification, the source code editor of the current theme will appear. On the right, look for a file called functions.php and select it for editing. At the very top of this file, let's put the following code:

```
// Define
'mfp_Add_Text'functionsm
fp_Add_Text()
{
  echo"<p style='color: red;'>This text is displayed when the page footer is
generated</p>";
}

// Hook the 'wp_footer' action hook, add the function named 'mfp_Add_Text' to
```

Let's save the changes and switch to page view. The text was inserted after the footer was generated:



Ten tekst wyświetla się po wygenerowaniu stopki strony

Removal of the Action function from Action Hook

To remove an action from the Action Hoop, we need to write a new function that calls remove_action(). This new feature obviously needs to be registered first. The following example will explain this.

```
// Hook the 'init' action, which is called after WordPress is finished loading
the core code, add the function 'remove_My_Meta_Tags'
add_action('init','remove_My_Meta_Tags');
```

```
// Remove the 'add_My_Meta_Tags' function from the wp_head action
hookfunctionsremove_My_Meta_Tags()
{
  remove_action('wp_head','add_My_Meta_Tags');
}
```

Functionremove_action() expects at least two parameters.

- The first required parameter is the name of the Action Hook to which the function is hooked
- The second required parameter is the name of the function we want to remove
- The third parameter (optional) is the priority of the original function. This parameter must be identical to the priority that was originally defined when adding the action to the action hook. If we have not defined a priority for our function, we should not specify this parameter.

So let's expand on our example with added text in the footer. Let's imagine that for some reason we don't want this text to be displayed on Mondays. Naturally, the simplest way would be to add an appropriate if statement in the body of our Action function. However, we want to see how Action delete works. Therefore, our example will remove the action on Mondays. Examine the code below:

```
// Define
'mfp_Add_Text'functionsm
fp_Add_Text()
{
  echo"<p style='color: red;'>This text is displayed when the page footer is
generated</p>";
}

// Hook the 'wp_footer' action hook, add the function named 'mfp_Add_Text' to
itadd_action("wp_footer","mfp_Add_Text");

// Define the function named 'mfp_Remove_TextOnMondays()' to remove our previous
function from the 'wp_footer' action
functionsmfp_Remove_TextOnMondays()
{
  if(date("l") ==="Monday") {
    // Target the 'wp_footer' action, remove the 'mfp_Add_Text' function from
    itremove_action("wp_footer","mfp_Add_Text");
  }
}

// Hook the 'wp_head' action, run the function named
```

In the example above, we registered an action in wp_head. Inside this action, if today is a Monday, we delete another action that adds a text under the footer. The question is why

we put our deletion action in wp_head instead of wp_footer? The answer is quite obvious. The wp_head action is executed earlier than the wp_footer actions. So we need to ensure that the action is removed before it executes. This is how we did the alignment.

Filters and Filter Hooks

The filter function allows you to modify the result data returned by existing functions and must be hooked to one of the Filter Hooks - just like it was in the case of Action Hooks. Filter Hooks behave similarly to Action Hooks in that they are called at various points in the script and are also context sensitive. The full list of filter points and the context in which they are called is described in**WordPress Plugin API/Filter Reference page**.

Pinning Filters

To add a filter function to any Filter Hook, the plugin needs to call a WordPress function called add_filter(), with at least two parameters. Analyze the following example:

```
// Hook the 'the_content' filter hook (content of any post), run the function
named 'mfp_Fix_Text_Spacing'
add_filter("the_content","mfp_Fix_Text_Spacing");

// Automatically correct double spaces from any
postfunctionsmfp_Fix_Text_Spacing($the_Post)
{
 $the_New_Post=str_replace(" "," ",$the_Post);

 return$the_New_Post;
}
```

- The first required parameter is the name of the Filter Hook
- The second required parameter is the name of the filtering function we want to run
- The third parameter (optional) is the priority of the function we want to run.
- The fourth parameter (optional) is the number of arguments, which means how many parameters our filter function can take. The default value is 1.

Plugin implementation

Since we have already learned the basics of Hooks in WordPress, we can go on to discuss the rules of creating a plugin. We will discuss the method of implementing WP extensions on the basis of an example plugin. We will follow the process of its creation.

The basic operation of our plugin will be to check whether a given post is a new post (published no more than X days ago). And for such new posts, the plugin will generate an appropriate label next to the post title. The plugin will also have a simple administration panel, in which we will specify what it means that a post is new - otherwise, how long the post is treated as new.

Step 1 - plugin name

We start by coming up with a name for our plugin. The name of the plugin is of little importance because the only recipients will be ourselves. In such a situation, make sure that the plugin does not collide with already installed extensions. However, if we plan to publish our plugin in the public WordPress repository, the name should be unique. It's best to check the uniqueness of the name in the search engine here:https://wordpress.org/plugins/.

In our case, we will use the name Newly Added Post Highliter.

Step 2 - preparation of the plugin's folder structure

A plugin, as we have already mentioned, is a set of PHP, CSS, JavaScript and other resources files. In the simplest approach, a plugin can be a single PHP file. The name of this file should correspond to the name of the plugin. It is best to close the whole thing in one parent folder with the name of our plugin.

All plugins are placed in the wp-content/plugins/ folder. So let's prepare our plugin folder structure as follows:

Of course, this is only a suggestion. We do not have to strictly adhere to such a hierarchy, if another one suits our needs more, it can easily be changed. In the example above, I also placed two files at once. The first newly-added-post-highliter.php will be the main file of our plugin. However, the css/style.css file, as the name suggests, will define the graphic design.

Step 3 - meta information about the plugin

We put the description of our plugin in the main PHP file as a comment. It requires the use of a specific format. Based on this comment, the WordPress engine loads the necessary information about the plugin. ABOUTrequirements, therefore, the header can be read more here:https://developer.wordpress.org/plugins/plugin-basics/header-requirements/. In our case, in the newly-added-post-highliter.php file, we will insert the following description:

```
<?php
/**
 * Plugin Name:       Newly Added Post Highliter
 * URI plugin:        https://example.com/plugins/Newly Added Post Highliter/
 * Description:       Highlight newly posts with tag.
 * version:           1.0
 * Requires at least: 5.0
 * PHP requirements:  7.2
 * Author:Peter        Jozwiak
 * AuthorURI: https://darksource.pl/
```

```
 * License:           GPL v2 or later
 * License URI:       https://www.gnu.org/licenses/gpl-2.0.html
 */
```

The minimum necessary is to enter the Plugin Name field. However, there are other fields worth considering here as well.

Step 4 - launching the plugin

Before we write any code, it is worth running the plugin on the website. Since it doesn't actually have a single line of code yet, it shouldn't cause errors. So we are sure that nothing bad will happen to our website after launch. We will develop the plugin incrementally to see which code causes script errors.

So let's upload the NewlyAddedPostHighliter folder with its contents to the wp-content/plugins/ folder on the serverWWW. Next, let's log in to the WordPress dashboard and go to the Plugins section. In the list of plugins we should see our new plugin:

Let's activate it by clicking the Activate link.

Step 5 – implementation of plugin functionality

We will start the implementation of our plugin by placing the styles. Since it is not the purpose of this lab to discuss how CSS works, I present the contents of the css/style.css file only for completeness. Remember to write your own plugin to define your own styles corresponding to the graphic assumptions of the website. For the purposes of this example, this file has the following content:

```
.naph_marker{
    display:none;
}


.naph_marker{
    color:white;margin
    -left:10px;
    background-
    color:red;padding:1px
    10px;border-
    radius:5px;font-
    size:0.5em;
```

```
}

article
    .naph_marker{disp
    lay:inline;
```

Let's move on to writing the plugin itself. All the code will be included in the main PHP file. We will discuss the code in the order in which it is placed.

Admin panel implementation

The first functionality we will deal with will be writing the admin panel in which we will set the parameters of our plugin. The configuration will be very simple and will come down to setting one parameter specifying the number of days from publication for which we will mark a given post with our tag.

Let's start by registering the link in the Settings menu. To do this, add the following code to the main file:

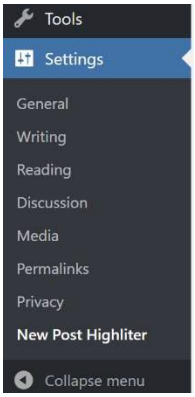
```
functionsnaph_admin_actions_register_menu(){
    add_options_page("Newly Added Post Highlighter","New Post
Highlighter",'manage_ options',"naph","naph_admin_page");
}

add_action('admin_menu','naph_admin_actions_register_menu');
```

This simple example does two things. In the first step, it registers an Action Hook for the admin_menu event. This is the event that is executed when the admin menu is generated. We assign a function called naph_admin_actions_register_menu to this event. This function calls the add_options_page function. This function is responsible for adding menu items in the Settings sectionYou can read the details of this function here:https://developer.wordpress.org/reference/functions/add_options_page/. It accepts up to 6 parameters:

- The first parameter is Page Title.
- The second parameter is responsible for the name/text displayed in the administrator menu
- The third parameter is the required permission for this element, the so-called capability. You can read more about roles and permissions here: https://wordpress.org/support/article/roles-and-capabilities/. In our case, we insert the required permission to change the settings stored in the capability named: manage_options.
- The fourth parameter is a string of characters that will build the URL to the administration page. It must be unique throughout the site.
- The fifth parameter is the name of the function to be run to generate the admin panel page.

Since we have already registered a reference to the function generating the admin panel, let's save its simplified version for now to be able to test the operation of our code. In our case, it will look like this (we will still modify it):

```
functionsnaph_admin_page(){
?>
    <h1>Newly Added Post Highliter</h1>
<?php
}
```



Let's save the changes to the file and refresh the page with the admin panel. Let's click on the Settings section. Now a link to our admin panel should be generated.

After clicking on it, a blank page will appear with the header stored in the function that generates the page content.

Let's move on to writing a function that supports the admin panel. In our case, it looks like this:

```
functionsnaph_admin_page(){
  // get _POST variable from
  globalsglobal$_POST;

  // process changes from
  formif(isset($_POST['naph_do_change'])){
    if($_POST['naph_do_change'] =='Y'){
      $opDays=$_POST['naph_days'];
      echo'<div class="notice notice-success is-
dismissible"><p>Settings saved.</p></div>';
      update_option('naph_days',$opDays);
    }
  }

  //read current option value
  $opDays=get_option('naph_days');

  //display admin page
?>
  <divclass="wrap">
    <h1>Newly Added Post Highliter</h1>
     <formsname="naph_form"methods="fast">
       <inputtype="hidden"name="naph_do_change"value="Y">
```

```
        <p>Highlight post title for
        <inputtype="number"name="naph_days"min="0"max="thirty"value="<?phpecho$opD
ays?>">days
        </p>
        <pclass="submit"><inputtype="submit"value="Submit"></p>
      </forms>
   </div>
<?php
}
```

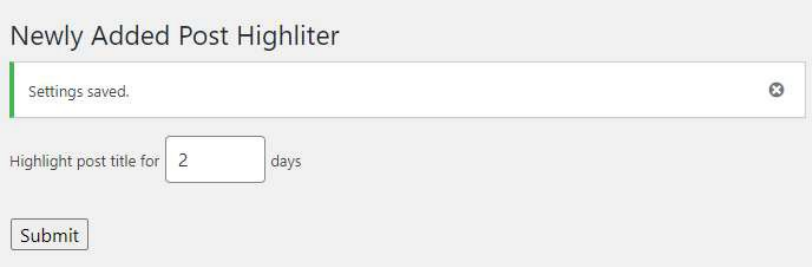
As you can see from the code comments, this function takes care of several things one by one. The first part is responsible for processing the form and saving the new data to the WordPress database. We use the built-in functionality of the CMS engine in the form of two functions:

- **update_option**(string $option, mixed $value, string|bool $autoload = null)
- **get_option**(string $option, mixed $default = false)

The first one stores the value under naph_days in the database. We do not have to worry about how to store this data. WordPress takes care of this for us. The second function reads the current value of this option. As you can see, it is a very easy and pleasant solution to use.

The next part of this function generates a form with the current settings.

This is all the functionality responsible for administration of the plugin. Let's save the changes and go to the browser. Let's try to set the value for our option and test if everything saves. The result should resemble the one in the image below:



Implementation of frontend functionality

Now it's time to write the part of the plugin responsible for generating a marker for the titles of the latest posts. To do this, we need to enter the code that supports this requirement into the plugin's main file. You can add this to the end of this file. In our example, the code looks like this:

```
functionsnaph_mark_new_post_title($content,$id){
    //read post publish date
    $date=get_the_date('ymd',$id);
    //get current date
    $now=date('ymd');
    //get setting for how long post is a new post
    $opDays=get_option('naph_days');
```

```
    //generate proper post
    titleif($now-$date<=$opDays)
        return$content."<supclass=\"naph_marker\">new</sup>";re
    turn$content;
}

add_filter("the_title","naph_mark_new_post_title",10,2);
```

As before, we register the naph_mark_new_post_title function to the filter named the_title. This filter is responsible for displaying the title of the post. We will modify it depending on the settings of our plugin.

Marking logic has been implemented in the naph_mark_new_post_title function; This function receives two parameters:

- The first is the current post title value
- The second is the post ID

At the beginning, we load information about the date of publication of the post using the get_the_date function, passing it the format and the ID of our post. Then we load the current date. Next we load our plugin settings.

At the end, we generate a post title with a marker if the post is newer than the number of days loaded from the plugin settings or the title is unchanged.

Let's save the changes and check if the plugin works. Of course, we haven't connected the styles yet. Therefore, the effect is not very impressive. We'll deal with that in the next step.

Style file registration

The last thing we have to dois to register predefined styles. We also do this in our main PHP file. At the very end, add the following code:

```
functionsnaph_register_styles(){
    // register styles
    wp_register_style('naph_styles',plugins_url('/css/style.css',FILES));
    //enable style (load in meta of
    html)wp_enqueue_style('naph_styles');
}

add_action('init','naph_register_styles');
```

For this purpose, we will again use a pinaction to the Action Hook named init. The init action is fired after the WordPress engine has loaded, but before any HTTP header is sent. In the pinned function, we use two functions to:

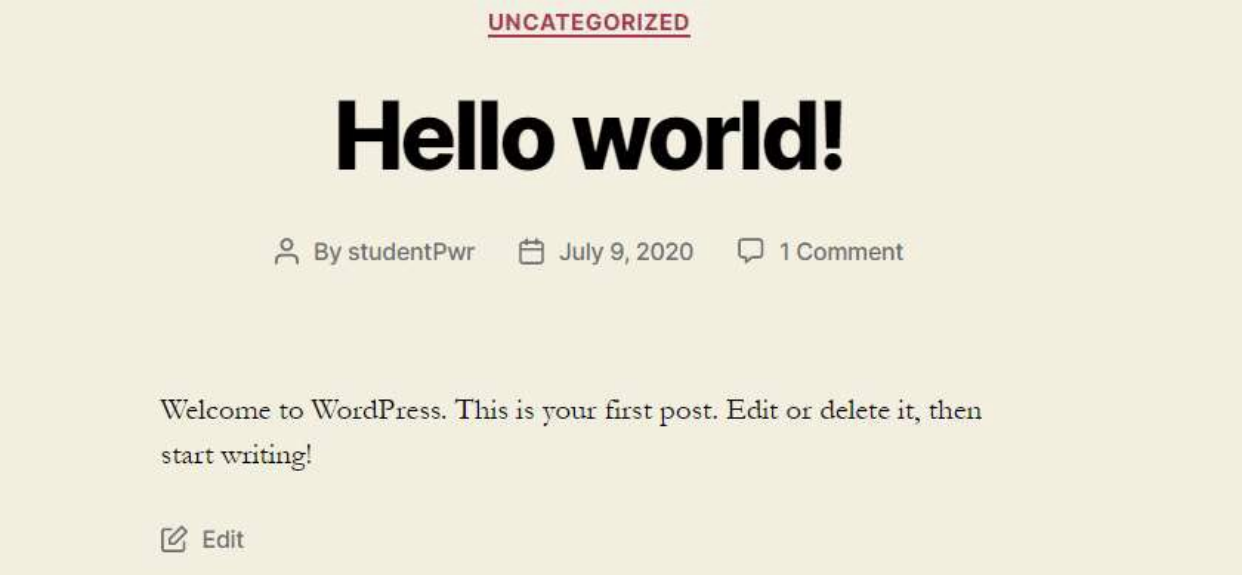
- **wp_register_style()**–registers our style file with the name given in the first argument

- **`wp_enqueue_style()`**– indicates to attach the file registered under the name from the argument to the html of the page.

The above mechanism gives you full control over styles management. By separating this functionality into registration and queue, we have the ability to include style files (as well as script files) only in the required places. We do not have to load these files every time - which saves transfer and speeds up the operation of the websiteWWW.

Let's save the changes to the file. It's time to check how the plugin works. Let's go to the browser. As a result, we should get a result similar to the following:

However, for the older post, the title remained unchanged:



WordPress debugging

Just one more digression to end this lab. When writing any code, it is natural for errors to occur. WordPress will not display an error message by default to protect this sensitive information from attackers. However, during implementation, they are of invaluable help to the developer.

To enable this information, just add the following line of code to the wp-config.php file:

```
define('WP_DEBUG',true);
```

From now on, error messages should be much more helpful in troubleshooting.We will find out more about WordPress debugging

here:https://wordpress.org/support/article/debugging-in-wordpress/