WordPress

CS-16: Content Management System using WordPress

B.C.A. Semester - 3

Topic-5

Advanced Development

- Advanced functions
  - add_action()
  - add_filter()
  - add_shortcode()
  - do_shortcode()
  - register_nav_menu()
- Custom Post Types
  - register_post_type()
  - register_taxonomy()
- Widget Area
  - register_sidebar()
  - dynamic_sidebar()

Mr. Gaurav.k.sardhara

9067351366

Advanced Functions

- add_action()- Hooks a function on to a specific action.

Actions are the hooks that the WordPress core launches at specific points during execution, or when specific events occur. Plugins can specify that one or more of its PHP functions are executed at these points, using the Action API.

```
add_action( string $tag, callable $function_to_add, int $priority = 10, int $accepted_args = 1 )
```

Parameters

- $tag (string) (Required) the name of the action to which the $function_to_add is hooked.
- $function_to_add (callable) (Required) the name of the function you wish to be called.
- $priority (int) (Optional) Used to specify the order in which the functions associated with a particular action are executed. Lower numbers correspond with earlier execution, and functions with the same priority are executed in the order in which they were added to the action. Default value: 10
- $accepted_args (int) (Optional) The number of arguments the function accepts. Default value: 1

Return: (true) will always return true

Source File: wp-includes/plugin.php

Usage:

```
add_action( $hook, $function_to_add, $priority, $accepted_args );
```

To find out the number and name of arguments for an action, simply search the code base for the matching do_action() call. For example, if you are hooking into 'save_post', you would find it in post.php:

```
do_action( 'save_post', $post_ID, $post, $update );
```

Your add_action call would look like:

```
add_action( 'save_post', 'wpdocs_my_save_post', 10, 3 );
```

And your function would be:

```
function wpdocs_my_save_post( $post_ID, $post, $update )
{     // do stuff here   }
```

Example (1): Using with a Class

To use add_action() when your plugin or theme is built using classes, you need to use the array callable syntax. You would pass the function to add_action() as an array, with $this as the first element, then the name of the class method, like so:

```
class WP_Docs_Class
{
      public function_construct()
      {     add_action( 'save_post', array( $this, 'wpdocs_save_posts' ) );    }
      public function wpdocs_save_posts()
      {     // do stuff here...      }
}
$wpdocsclass = new WP_Docs_Class();
```

Example (2): Using with static functions in a class

If the class is called statically the approach has to be like below as $this is not available. This also works if class is extended. Use the following:

```
class WP_Docs_Static_Class
{
      public static function init()
      {  add_action('save_post',array(get_called_class(),'wpdocs_save_posts'));   }
      public function wpdocs_save_posts()
      {     // do stuff here...      }
}
WP_Docs_Static_Class::init();
```

Example (3): Simple Hook

To email some friends whenever an entry is posted on your blog:

```
function wpdocs_email_friends( $post_id )
{
	$friends = 'bob@example.org, susie@example.org';
	wp_mail( $friends, "sally's blog updated", 'I just put something on my blog:
http://blog.example.com' );
	return $post_id;
}
add_action( 'publish_post', 'wpdocs_email_friends' );
```

Example (4): Accepted Arguments

A hooked function can optionally accept arguments from the action call, if any are set to be passed. In this simplistic example, the echo_comment_id function takes the $comment_id argument, which is automatically passed to when the do_action() call using the comment_id_not_found filter hook is run.

```
function echo_comment_id( $comment_id )
{	printf( 'Comment ID %s could not be found', esc_html( $comment_id ) );  }
add_action( 'comment_id_not_found', 'echo_comment_id', 10, 1 );
```

- add_filter()

  Hook a function or method to a specific filter action.

  ```
  add_filter( string $tag, callable $function_to_add, int $priority = 10, int
  $accepted_args = 1 )
  ```

  WordPress offers filter hooks to allow plugins to modify various types of internal data at runtime.

  A plugin can modify data by binding a callback to a filter hook. When the filter is later applied, each bound callback is run in order of priority, and given the opportunity to modify a value by returning a new value.

  The following example shows how a callback function is bound to a filter hook.

  Note that $example is passed to the callback, (maybe) modified, and then returned:

  ```
  function example_callback( $example )
  {	// Maybe modify $example in some way.
  	return $example;
  }
  add_filter( 'example_filter', 'example_callback' );
  ```

  Bound callbacks can accept from none to the total number of arguments passed as parameters in the corresponding apply_filters() call.

  In other words, if an apply_filters() call passes four total arguments, callbacks bound to it can accept none (the same as 1) of the arguments or up to four. The important part is that the $accepted_args value must reflect the number of arguments the bound callback actually opted to accept. If no arguments were accepted by the callback that is considered to be the same as accepting 1 argument. For example:

  ```
  // Filter call.
  $value = apply_filters( 'hook', $value, $arg2, $arg3 );
  // Accepting zero/one arguments.
  function example_callback()
  {	...	return 'some value';	}
  add_filter( 'hook', 'example_callback' ); // Where $priority is default 10,
  $accepted_args is default 1.
  // Accepting two arguments (three possible).
  function example_callback( $value, $arg2 )
  ```

```
{     ...    return $maybe_modified_value; }
add_filter( 'hook', 'example_callback', 10, 2 ); // Where $priority is 10,
$accepted_args is 2.
```

Note: The function will return true whether or not the callback is valid. It is up to you to take care. This is done for optimization purposes, so everything is as quick as possible.

Parameters

- $tag (string) (Required) The name of the filter to hook the $function_to_add callback to.
- $function_to_add (callable) (Required) The callback to be run when the filter is applied.
- $priority (int) (Optional) Used to specify the order in which the functions associated with a particular action are executed. Lower numbers correspond with earlier execution, and functions with the same priority are executed in the order in which they were added to the action. Default value: 10
- $accepted_args (int) (Optional) The number of arguments the function accepts. Default value: 1

Return: (true)

Source File: wp-includes/plugin.php

Example:

```
function add_filter( $tag, $function_to_add, $priority = 10, $accepted_args = 1 )
{
      global $wp_filter;
      if ( ! isset( $wp_filter[ $tag ] ) )
      {     $wp_filter[ $tag ] = new WP_Hook(); }
      $wp_filter[ $tag ]->add_filter( $tag, $function_to_add, $priority,
$accepted_args );
      return true;
}
```

Hooked functions can take extra arguments that are set when the matching do_action() or apply_filters() call is run. For example, the comment_id_not_found action will pass the comment ID to each callback.

Although you can pass the number of $accepted_args, you can only manipulate the $value. The other arguments are only to provide context, and their values cannot be changed by the filter function.

You can also pass a class method as a callback.

Static class method:

```
add_filter( 'media_upload_newtab', array( 'My_Class', 'media_upload_callback' ) );
```

Instance method:

```
add_filter( 'media_upload_newtab', array( $this, 'media_upload_callback' ) );
```

You can also pass an anonymous function as a callback. For example:

```
add_filter( 'the_title', function( $title ) { return '<strong>' . $title .
'</strong>'; } );
```

Example: Let's add extra sections to TwentySeventeen Front page.

By default, TwentySeventeen theme has 4 sections for the front page. This example will make them 6

```
add_filter('twentyseventeen_front_page_sections', 'prefix_custom_front_page_sections');
function prefix_custom_front_page_sections( $num_sections )
{     return 6;    }
```

- add_shortcode()

  Adds a hook for a shortcode tag.

Usage: <?php add_shortcode($tag , $func); ?>

Parameters:

- $tag (string) (required) Shortcode tag to be searched in post content Default: None
- $func (callable) (required) Hook to run when shortcode is found Default: None

Return Values: (none)

Examples:

Simplest example of a shortcode tag using the API: [footag foo="bar"]

```
function footag_func( $atts )
{       return "foo = {$atts['foo']}";        }
add_shortcode( 'footag', 'footag_func' );
```

Example with nice attribute defaults: [bartag foo="bar"]

```
function bartag_func( $atts )
{
        $atts = shortcode_atts( array ('foo' => 'no foo', 'baz' => 'default baz'),
$atts, 'bartag' );
        return "foo = {$atts['foo']}";
}
add_shortcode( 'bartag', 'bartag_func' );
```

Example with enclosed content: [baztag]content[/baztag]

```
function baztag_func( $atts, $content = "" )
{       return "content = $content";  }
add_shortcode( 'baztag', 'baztag_func' );
```

If your plugin is designed as a class write as follows:

```
class MyPlugin
{       public static function baztag_func( $atts, $content = "" )
        {       return "content = $content";  }
}
add_shortcode( 'baztag', array( 'MyPlugin', 'baztag_func' ) );
```

Notes

- The shortcode callback will be passed three arguments: the shortcode attributes, the shortcode content (if any), and the name of the shortcode.
- There can only be one hook for each shortcode. Which means that if another plugin has a similar shortcode, it will override yours or yours will override theirs depending on which order the plugins are included and/or ran.
- Shortcode attribute names are always converted to lowercase before they are passed into the handler function. Values are untouched.
- Note that the function called by the shortcode should never produce output of any kind. Shortcode functions should return the text that is to be used to replace the shortcode. Producing the output directly will lead to unexpected results. This is similar to the way filter functions should behave, in that they should not produce expected side effects from the call, since you cannot control when and where they are called from.

- do_shortcode()

  Search content for shortcodes and filter shortcodes through their hooks.

  ```
  do_shortcode( string $content, bool $ignore_html = false )
  ```

  If there are no shortcode tags defined, then the content will be returned without any filtering. This might cause issues when plugins are disabled but the shortcode will still show up in the post or content.

Parameters:

- $content (string) (Required) Content to search for shortcodes.
- $ignore_html (bool) (Optional) When true, shortcodes inside HTML elements will be skipped. Default value: false

Return: (string) Content with shortcodes filtered out.

Source File: wp-includes/shortcodes.php

Example:

```
function do_shortcode( $content, $ignore_html = false )
{	global $shortcode_tags;
	if ( false === strpos( $content, '[' ) )
	{	return $content;	}

	if (empty($shortcode_tags) || !is_array($shortcode_tags))
		return $content;

	// Find all registered tag names in $content.
	preg_match_all( '@\[([^<>&/\[\]\x00-\x20=]++)@', $content, $matches );
	$tagnames = array_intersect( array_keys( $shortcode_tags ), $matches[1] );

	if ( empty( $tagnames ) )
	{	return $content;	}

$content = do_shortcodes_in_html_tags( $content, $ignore_html, $tagnames );
$pattern = get_shortcode_regex( $tagnames );
$content = preg_replace_callback( "/$pattern/", 'do_shortcode_tag', $content );

	// Always restore square braces so we don't break things like <!--[if IE ]>
	$content = unescape_invalid_shortcodes( $content );
	return $content;
}
```

- register_nav_menu()

  Registers a single custom Navigation Menu in the custom menu editor (in WordPress 3.0 and above). This allows administration users to create custom menus for use in a theme.

  See register_nav_menus() for creating multiple menus at once.

Usage: <?php register_nav_menu($location, $description); ?>

Parameters:

- $location (string) (required) Menu location identifier, like a slug. Default: None
- $description (string) (required) Menu description - for identifying the menu in the dashboard. Default: None

Return Values: None.

Examples:

```
<?php
	add_action( 'after_setup_theme', 'register_my_menu' );
	function register_my_menu()
	{	register_nav_menu('primary', __('Primary Menu','theme-slug'));	}
?>
```

Notes

- This function automatically registers custom menu support for the theme therefore you do not need to call add_theme_support('menus');
- This function actually works by simply calling register_nav_menus() in the following way:
  `register_nav_menus( array( $location => $description ) );`

- register_nav_menus()
  Registers multiple custom navigation menus in the new custom menu editor of WordPress 3.0. This allows for the creation of custom menus in the dashboard for use in your theme.
  See register_nav_menus() for creating multiple menus at once.

Usage: <?php register_nav_menus($locations); ?>

Parameters:

- $locations (array) (required) an associative array of menu location slugs (key) and descriptions (according value). Default: None

Return Values: None.

Source File: register_nav_menus() is located in wp-includes/nav-menu.php

Examples:

```
register_nav_menus( array(
      'pluginbuddy_mobile' => 'PluginBuddy Mobile Navigation Menu',
      'footer_menu' => 'My Custom Footer Menu', ) );
```

Notes

- This function automatically registers custom menu support for the theme, therefore you do not need to call add_theme_support('menus');
- Use wp_nav_menu() to display your custom menu.
- On the Menus admin page, there is a Show advanced menu properties to allow "Link Target" "CSS Classes" "Link Relationship (XFN) Description"
- Use get_registered_nav_menus to get back a list of the menus that have been registered in a theme.

Custom Post Types

- register_post_type()
  Create or modify a post type. register_post_type should only be invoked through the 'init' action. It will not work if called before 'init', and aspects of the newly created or modified post type will work incorrectly if called later.

  Note: You can use this function in themes and plugins. However, if you use it in a theme, your post type will disappear from the admin if a user switches away from your theme.

Reserved Post Types

The following post types are reserved and used by WordPress already.

- post
- page
- attachment
- revision
- nav_menu_item
- custom_css
- customize_changeset

In addition, the following post types should not be used as they interfere with other WordPress functions

- action
- author
- order
- them

In general, you should always prefix your post types, or specify a custom `query_var`, to avoid conflicting with existing WordPress query variables.

Usage: <?php register_post_type($post_type, $args); ?>

Parameters:

- $post_type (string) (required) Post type. (max. 20 characters, cannot contain capital letters or spaces). Default: None
- $args (array) (optional) An array of arguments. Default: None

Source File: register_post_type() is located in wp-includes/post.php.
Example: An example of registering a post type called "book".

```
function codex_custom_init()
{
	$args = array('public' => true, 'label' => 'Books');
	register_post_type( 'book', $args );
}
add_action( 'init', 'codex_custom_init' );
```

- register_taxonomy()
  This function adds or overwrites a taxonomy. It takes in a name, an object name that it affects, and an array of parameters. It does not return anything.

  Care should be used in selecting a taxonomy name so that it does not conflict with other taxonomies, post types, and reserved WordPress public and private query variables.

Usage: <?php `register_taxonomy( $taxonomy, $object_type, $args );` ?>
Parameters:

- $taxonomy (string) (required) The name of the taxonomy. Name should only contain lowercase letters and the underscore character, and not be more than 32 characters long (database structure restriction). Default: None
- $object_type (array/string) (required) Name of the object type for the taxonomy object. Object-types can be built-in Post Type or any Custom Post Type that may be registered. Default: None

Source File: register_taxonomy() is located in wp-includes/taxonomy.php.
Example:

```
add_action( 'init', 'create_book_tax' );
function create_book_tax()
{	register_taxonomy('genre', 'book', array(
	'label' => __( 'Genre' ), 'rewrite' => array( 'slug' => 'genre' ), 'hierarchical' =>
true,));
}
```

Widget Area

- register_sidebar()
  Builds the definition for a single sidebar and returns the ID. Call on "widgets_init" action.

Usage: <?php `register_sidebar( $args );` ?>
Default Usage:

```
<?php $args = array(
	'name'          => __( 'Sidebar name', 'theme_text_domain' ),
	'id'            => 'unique-sidebar-id',
	'description'   => '',
	'class'         => '',
	'before_widget' => '<li id="%1$s" class="widget %2$s">',
	'after_widget'  => '</li>',
	'before_title'  => '<h2 class="widgettitle">',
	'after_title'   => '</h2>');
?>
```

Parameters:

- args (string/array) (optional) Builds Sidebar based off of 'name' and 'id' values. Default: None
  - name - Sidebar name (default is localized 'Sidebar' and numeric ID).
  - id - Sidebar id - Must be all in lowercase, with no spaces (default is a numeric auto-incremented ID). If you do not set the id argument value, you will get E_USER_NOTICE messages in debug mode, starting with version 4.2.
  - description - Text description of what/where the sidebar is. Shown on widget management screen. (Since 2.9) (default: empty)
  - class - CSS class to assign to the Sidebar in the Appearance -> Widget admin page. This class will

only appear in the source of the WordPress Widget admin page. It will not be included in the frontend of your website. Note: The value "sidebar" will be prepended to the class value. For example, a class of "tal" will result in a class value of "sidebar-tal". (Default: empty).

- before_widget - HTML to place before every widget(default: <li id="%1$s" class="widget %2$s">) Note: uses sprintf for variable substitution
- after_widget - HTML to place after every widget (default: \n).
- before_title - HTML to place before every title (default: <h2 class="widgettitle">).
- after_title - HTML to place after every title (default: </h2>\n).

The optional args parameter is an associative array that will be passed as a first argument to every active widget callback. (If a string is passed instead of an array, it will be passed through parse_str() to generate an associative array.) The basic use for these arguments is to pass theme-specific HTML tags to wrap the widget and its title.

Notes: Calling register_sidebar() multiple times to register a number of sidebars is preferable to using register_sidebars() to create a bunch in one go, because it allows you to assign a unique name to each sidebar (eg: "Right Sidebar", "Left Sidebar"). Although these names only appear in the admin interface it is a best practice to name each sidebar specifically, giving the administrative user some idea as to the context for which each sidebar will be used.

Source File: register_sidebar() is located in wp-includes/widgets.php.
Example: This will create a sidebar named "Main Sidebar" with <h2> and </h2> before and after the title:

```
add_action( 'widgets_init', 'theme_slug_widgets_init' );
function theme_slug_widgets_init()
{
	register_sidebar( array(
	'name' => __( 'Main Sidebar', 'theme-slug' ),
	'id' => 'sidebar-1',
	'description' => __( 'Widgets in this area will be shown on all posts and pages.',
'theme-slug' ),
	'before_widget' => '<li id="%1$s" class="widget %2$s">',
	'after_widget'  => '</li>',
	'before_title'  => '<h2 class="widgettitle">',
	'after_title'   => '</h2>',	) );
}
```

- dynamic_sidebar()

  This function calls each of the active widget callbacks in order, which prints the markup for the sidebar. If you have more than one sidebar, you should give this function the name or number of the sidebar you want to print. This function returns true on success and false on failure.

  The return value should be used to determine whether to display a static sidebar. This ensures that your theme will look good even when the Widgets plug-in is not active.

  If your sidebars were registered by number, they should be retrieved by number. If they had names when you registered them, use their names to retrieve them

Usage: <?php dynamic_sidebar($index); ?>
Parameters:

- index (integer/string) (optional) Name or ID of dynamic sidebar. Default: 1

Return Value: (boolean) True, if widget sidebar was found and called. False if not found or not called
Source File: dynamic_sidebar() is located in wp-includes/widgets.php.
Examples: Here is the recommended use of this function:

```
<?php if ( is_active_sidebar( 'left-sidebar' ) ) : ?>
	<ul id="sidebar"><?php dynamic_sidebar( 'left-sidebar' ); ?></ul>
<?php endif; ?>
	<ul id="sidebar"><?php dynamic_sidebar( 'right-sidebar' ); ?></ul>
	<ul id="sidebar">
```

```
            <?php if ( ! dynamic_sidebar() ) : ?>
            <li>{static sidebar item 1}</li>
            <li>{static sidebar item 2}</li>
            <?php endif; ?>
      </ul>
```

Multiple Sidebars

You can load a specific sidebar by either their name (if given a string) or ID (if given an integer). For example, dynamic_sidebar('top_menu') will present a sidebar registered with register_sidebar(array('name'=>'top_menu',)).

Using ID's (dynamic_sidebar(1)) is easier in that you don't need to name your sidebar, but they are harder to figure out without looking into your functions.php file or in the widgets administration panel and thus make your code less readable. Note that ID's begin at 1.

If you name your own ID values in the register_sidebar() WordPress function, you might increase readability of the code. The ID should be all lowercase alphanumeric characters and not contain white space. You can also use the - and _ characters. IDs must be unique and cannot match a sidebar name. Using your own IDs can also make the sidebar name translatable.

```
// See the __() WordPress function for valid values for $text_domain.
register_sidebar( array(
      'id'          => 'top-menu',
      'name'        => __( 'Top Menu', $text_domain ),
      'description' => __( 'This sidebar is located above the age logo.', $text_domain ),
) );
```

This allows the use of dynamic_sidebar('top-menu') which uses an ID and is readable.