CODE > PHP

Organize Your Next PHP Project the Right Way

by Derek Reynolds 20 Jul 2009 Difficulty: Intermediate Languages: English

PHP Web Development

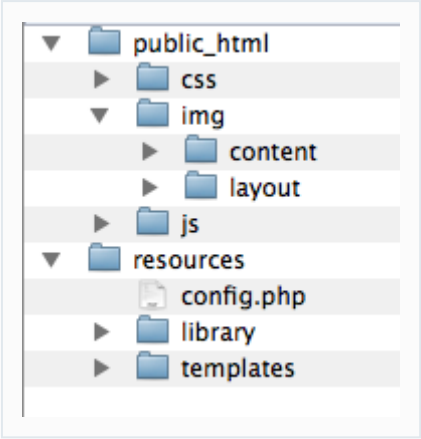
When starting out with PHP, it can be daunting figuring out how best to organize a project. If you've ever been confused with where to put your images, external libraries, or keeping your logic separate from your layout, then check out these tips; they'll get you heading in the right direction.

Tutorial Details

- **Program**: PHP/Projects
- **Version**: 1
- **Difficulty:** Easy
- **Estimated Completion Time:** 20 minutes

Directory Structure

I'd say the number one thing in getting your project up and running quickly is having a solid directory structure you can reuse for multiple projects. If you are using a framework, usually it will provide a structure to use, but in this scenario we're working on a simple site or app.



Breakdown

- You are probably very familiar with the `public_html` structure. This is the Document Root in which all your public files are accessed (`/public_html/page.php` is accessed at `example.com/page.php`).

    - img — All your image files. I decided to split content images from layout images.
    - css — All your css files.
    - js — All your javascript files.
- The `resources` directory should hold all 3rd party libraries, custom libraries, configs and any other code that acts as a resource in your project.

    - config.php — Main configuration file. Should store site wide settings.
    - library — Central location for all custom and third party libraries.
    - templates — Reusable components that make up your layout.

The Config File

As designers and developers our main goal is to do as little work as possible. One way to reach this goal is with config files. To get a better idea of what the configuration file should have check out this example.

```
<?php

/*
    The important thing to realize is that the config file should be
    page of your project, or at least any page you want access to the
    This allows you to confidently use these settings throughout a pr
    if something changes such as your database credentials, or a path
    you'll only need to update it here.
*/

$config = array(
    "db" => array(
        "db1" => array(
```

```
14              "dbname" => "database1",
15              "username" => "dbUser",
16              "password" => "pa$$",
17              "host" => "localhost"
18          ),
19          "db2" => array(
20              "dbname" => "database2",
21              "username" => "dbUser",
22              "password" => "pa$$",
23              "host" => "localhost"
24          )
25      ),
26      "urls" => array(
27          "baseUrl" => "http://example.com"
28      ),
29      "paths" => array(
30          "resources" => "/path/to/resources",
31          "images" => array(
32              "content" => $_SERVER["DOCUMENT_ROOT"] . "/images/content
33              "layout" => $_SERVER["DOCUMENT_ROOT"] . "/images/layout"
34          )
35      )
36  );
37
38  /*
39      I will usually place the following in a bootstrap file or some ty
40      setup file (code that is run at the start of every page request),
41      just as well in your config file if it's in php (some alternative
42  */
43
44  /*
45      Creating constants for heavily used paths makes things a lot easi
46      ex. require_once(LIBRARY_PATH . "Paginator.php")
47  */
48  defined("LIBRARY_PATH")
49      or define("LIBRARY_PATH", realpath(dirname(__FILE__) . '/library'
50
51  defined("TEMPLATES_PATH")
52      or define("TEMPLATES_PATH", realpath(dirname(__FILE__) . '/templa
```

```
53
54  /*
55      Error reporting.
56  */
57  ini_set("error_reporting", "true");
58  error_reporting(E_ALL|E_STRCT);
59
60  ?>
```

This is a basic drop-in config file. A multi-dimensional array serves as a flexible structure for accessing various config items such as database credentials.

- db — Store database credentials or other data pertaining to your databases.
- paths — Commonly used paths to various resources for your site.
  - log files
  - upload directories
  - resources
- urls — Storing urls can be really handy when referencing remote resources throughout your site.
- emails — Store debugging or admin emails to use when handling errors or in contact forms.

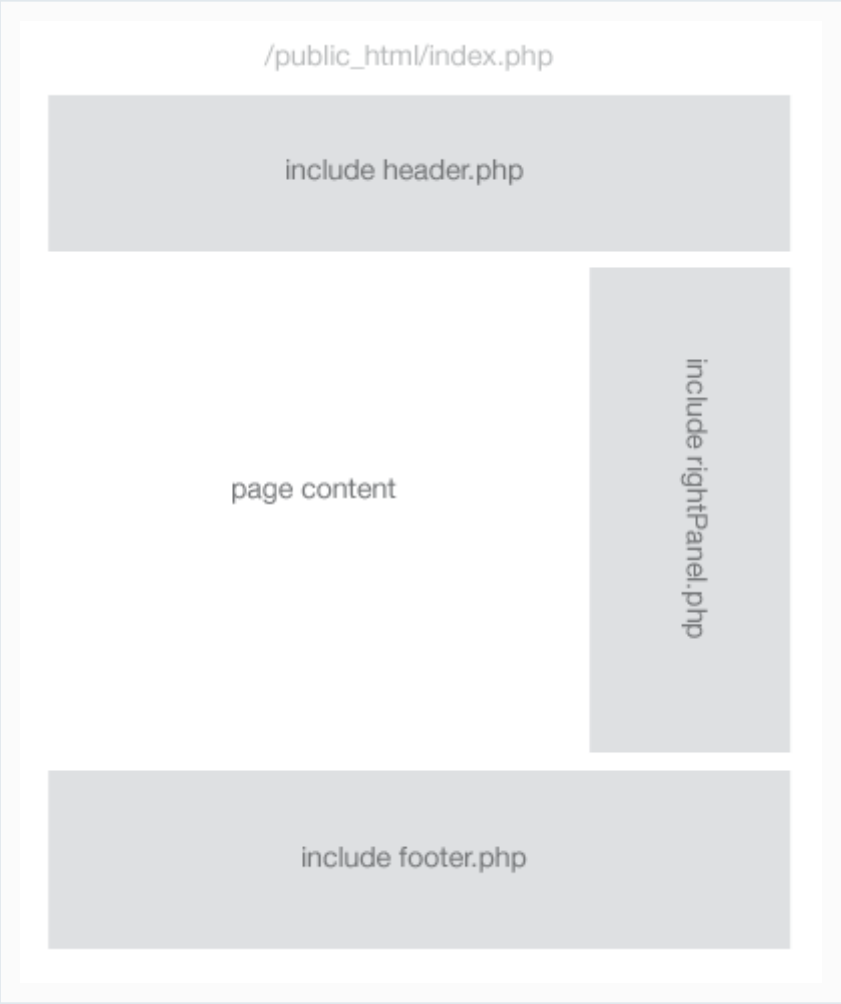
Using constants for commonly used paths makes include statements (`require` or `include`) a breeze, and if the path ever changes you'll only need to update it in one place.

Using Different Config Files For Multiple Environments

By using different config files for multiple environments you can have relevant settings depending on the current environment. Meaning, if you use different database credentials or different paths for each environment, by setting up the respective config files you ensure that your code will work without hassle when updating your live site. This also allows you to have different error reporting settings based on the current environment. Never ever display errors on your live site! Displaying errors on the live site could expose sensitive data to users (such as passwords).

The Layout

Reusable templates are another big time saver. There are some great libraries for templating (such as Smarty), and I always encourage using such a library rather than reinventing the wheel. These libraries offer a lot of functionality (like helper methods for formatting currency and obfuscating email addresses). Since this is a simple site however we don't want to take the time to setup the library and will be using the most basic of basic templates. We achieve this by including common sections or modules in to our site pages; this way if we want to change something in the header, like adding a link to the global navigation, it is propagated throughout the site.



header.php

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
   "http://www.w3.org/TR/html4/strict.dtd">

<html lang="en">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8
    <title>Simple Site</title>
</head>
```

```
09
10  <body>
11  <div id="header">
12      <h1>Simple Site</h1>
13      <ul class="nav global">
14          <li><a href="#">Home</a></li>
15          <li><a href="#">Articles</a></li>
16          <li><a href="#">Portfolio</a></li>
17      </ul>
18
19  </div>
```

rightPanel.php

```
01  <div id="siteControls">
02      <ul class="categories">
03          <li>PHP</li>
04          <li>HTML</li>
05          <li>CSS</li>
06      </ul>
07      <div class="ads">
08          <!-- ads code -->
09      </div>
10
11  </div>
```

footer.php

```
1  <div id="footer">
2      Footer content...
3  </div>
4  </body>
5  </html>
```

index.php

Let's say that we put all of our layout components (header, footer, rightPanel) in our resources directory under templates.

```
<?php
    // load up your config file
    require_once("/path/to/resources/config.php");

    require_once(TEMPLATES_PATH . "/header.php");
?>
<div id="container">
    <div id="content">
        <!-- content -->
    </div>
    <?php
        require_once(TEMPLATES_PATH . "/rightPanel.php");
    ?>
</div>
<?php
    require_once(TEMPLATES_PATH . "/footer.php");
?>
```

Taking It Further

While this basic template system gets you off to a great start, you can take it a lot further. For instance, you can create a class or functions that include all the template files and accept a content file as an argument to render within the layout. This way you don't need to keep including the template files in every page of your site, but rather abstract that logic out meaning even less work down the road. I'll show you a quick example.

/resources/library/templateFunctions.php

```php
<?php
    require_once(realpath(dirname(__FILE__) . "/../config.php"));

    function renderLayoutWithContentFile($contentFile, $variables = a
    {
        $contentFileFullPath = TEMPLATES_PATH . "/" . $contentFile;

        // making sure passed in variables are in scope of the templa
        // each key in the $variables array will become a variable
        if (count($variables) > 0) {
            foreach ($variables as $key => $value) {
                if (strlen($key) > 0) {
                    ${$key} = $value;
                }
            }
        }

        require_once(TEMPLATES_PATH . "/header.php");

        echo "<div id=\"container\">\n"
           . "\t<div id=\"content\">\n";

        if (file_exists($contentFileFullPath)) {
            require_once($contentFileFullPath);
        } else {
            /*
                If the file isn't found the error can be handled in l
                In this case we will just include an error template.
            */
            require_once(TEMPLATES_PATH . "/error.php");
        }

        // close content div
        echo "\t</div>\n";

        require_once(TEMPLATES_PATH . "/rightPanel.php");

        // close container div
```

```
39          echo "</div>\n";
40
41          require_once(TEMPLATES_PATH . "/footer.php");
42      }
43  ?>
```

index.php

This is assuming you have a file called home.php in your templates directory that acts as a content template.

```
01  <?php
02
03      require_once(realpath(dirname(__FILE__) . "/../resources/config.p
04
05      require_once(LIBRARY_PATH . "/templateFunctions.php");
06
07      /*
08          Now you can handle all your php logic outside of the template
09          file which makes for very clean code!
10      */
11
12      $setInIndexDotPhp = "Hey! I was set in the index.php file.";
13
14      // Must pass in variables (as an array) to use in template
15      $variables = array(
16          'setInIndexDotPhp' => $setInIndexDotPhp
17      );
18
19      renderLayoutWithContentFile("home.php", $variables);
20
21  ?>
```

home.php

```
<!-- Homepage content -->
<h2>Home Page</h2>

<?php

    /*
        Any variables passed in through the variables parameter in ou
        are available in here.
    */

    echo $setInIndexDotPhp;

?>
```

Benefits of This Method Include:

- Greater separation of logic and view (php and html). Separating concerns like this makes for cleaner code, and the job of the designer or developer becomes easier as they are mostly working with their respective code.

- Encapsulating the template logic into a function allows you to make changes to how the template renders without updating it on each page of your site.

Symlinks

On Unix based systems (os x, linux) there is a neat little feature called symlinks (Symbolic Links). Symlinks are references to actual directories or files on the filesystem. This is really great for when you have a shared resource, such as a library used between multiple projects. Here are a few concrete things you can do with symlinks:

- Have two versions of your resource directory. When updating your live server you can upload your latest files into an arbitrary directory. Simply point the symlink to this new directory instantly updating your code base. If something goes wrong you can instantly rollback to the previous (working) directory.

- Shared resources are easily managed with symlinks. Say you have a custom library you've been working on, any updates to the library you make in one project will be immediately available in another.

Using Symlinks

Symlinks vs Hardlinks

Symlinks, or softlinks, act as references to full paths on the filesystem. You can use symlinks in multiple locations and the filesystem treats them as if they were the actual file or directory they reference. Hardlinks on the other hand are pointers to a file on the disk (think shortcuts in windows); they take you to the actual location of the file.

There are a few things you should consider when using symlinks. Your server configuration must be set up to follow symlinks. For Apache this is done in the httpd.conf file. Find the Directory block and make sure that Options FollowSymLinks is there. If not add it and then restart Apache.

```
<Directory />
    Options FollowSymLinks
    AllowOverride None
</Directory>
```

Creating Symlinks in OS X

There are 2 ways to create symlinks in OS X:

- Via the command line, navigate (cd, change directory) to the directory in which you want the symlink to be created, then use the following command:

  ```
  $: ln -s /path/to/actual/dir targetDir
  ```

  So if our custom library lives in `~/Sites/libraries/myCustomLibrary` we'd cd to where we want to use that library `cd ~/Sites/mySite/resources/library` and enter:

  ```
  $: ln -s ~/Sites/libraries/myCustomLibrary myCustomLibrary
  ```

  Note that this method should work in all Unix based operating systems.

- The alternative is through the finder. By holding alt + cmd while clicking and dragging a file, a Symlink (or alias in os x) that points to the file is created.

Creating Symlinks in Windows

To accomplish this in windows you'll need to use the mklink command in the command prompt:

```
1 C:\mklink /D C:\libraries\myCustomLibrary C:\Users\derek\Sites\mySite\
```

Summary

These tips are meant for beginners or those creating simple sites or applications. Ideally for larger applications or sites, you'll want to consider something more advanced like the MVC architecture and Object Oriented programming. I encourage you to look into these once you've gotten your feet wet and feel that you've outgrown most of the steps above. I decided not to cover source control as it's a pretty large subject on its own, but these tips should help you in organizing your files for easier source control if desired (hint: store stuff like layout images in your resource directory and symlink it into your `/public_html/img` dir). Definitely look in to using source control, like subversion or git for all of your projects.

Hope you find these tips helpful when starting your next PHP Project. Thanks!

Resources

- Smarty Templating Engine
- Multitier Architecture
- MVC
- Object Oriented Programming
- Subversion For Designers (version control)
- Symlinks
- Hardlinks

Derek Reynolds

Boston based web developer whom primarily lives in PHP, but breaks things in Objective-C from time to time. Loves working with front-end technologies such as HTML, CSS and Javascript. Feel free to follow me on twitter or check out my blog.