www.phparch.com

August 2015
VOLUME 14 - ISSUE8

php[architect]

Data Manipulation

How Many CMSs Can You Fit Inside a Website?

Introduction to Development with Aerospike

How GZIP Compression Works

Leveling Up: Databasics with PDO

FREE Article!

ALSO INSIDE

Introduction to Product Management for Developers

Education Station:
PHP Dependency Injection Part Three

Community Corner:
August 2015

finally{}:
Outward Appearances of PHP 7

How Many CMSs Can You Fit Inside a Website?

Julian Egelstaff

Why do we draw neat little boxes around one CMS or another instead of using bits of one system and bits of another? Even though CMSs are supposed to offer modularity and avoid the need for hooking up massive bits of code to each other they only do so within their own walled garden of modules and plugins. What happens if you push this boundary and throw multiple systems at a single HTTP request?

DisplayInfo()

Related URLs:

- Drupal—http://drupal.org
- Formulize—http://www.freeform.ca/en/formulize

Like all good technical ideas, PHP's complexity and usefulness grew...

Once upon a time, PHP was simply a way to extend HTML, instead of writing static HTML like this:

Page 1:

```
<h1>About Us</h1>
```

Page 2:

```
<h1>Products</h1>
```

PHP lets you make a single file and do this:

```
echo "<h1>$pageTitle</h1>";
```

It didn't take long for people to figure out that they were using PHP in the same way repeatedly on many different websites. Like all good technical ideas, PHP's complexity and usefulness grew in response to these discoveries about how it was being used. Before long, PHP started to look like this:

Logic:

```
$page = $database->get($page_id);
$template->assign($page);
```

Template:

```
<h1>{$page->pageTitle}</h1>
```

At this point, our story splits into two paths. Down one path lies the land of frameworks of various shapes and sizes: Zend Framework, Symfony, CakePHP, eZ Components, and CodeIgniter, to name just a handful.

Down the other path lies the land of content management systems (CMSs), which also come in all different shapes and sizes: PHP-Nuke, Joomla, TikiWiki, Drupal, TYPO3, and WordPress, to name just a few.

The paths have crossed from time to time, as a number of CMSs have incorporated some frameworks into their codebases. Notably, Drupal 8 is highly dependent on Symfony 2.

One Language. Multiple Dialects.

As a result of all this work and evolution, we have at our disposal some incredibly powerful tools for creating and managing websites. We have also ended up with a whole series of silos, with distinct ecosystems developing around different shards of the overall PHP landscape.

Code that is written with one framework in mind may or may not be easy to refactor to use with a different framework. On the CMS side, the problem is particularly pronounced, as plugins written for WordPress cannot in any way be directly installed or used in Drupal or any other CMS.

We've ended up with a mountain of open source PHP code, which is trapped inside de facto proprietary-style walled gardens! When the code you're concerned with is on the framework side, you can always opt to spend the time to refactor and connect different pieces of code together. But what about when you're on the CMS side?

The whole point of using a CMS is often speed and efficiency, both of initial deployment and also in making changes. That includes changes to content and to configuration, which should all be point and click, not code based. Therefore, if you want to use features from multiple CMSs, refactoring a great deal of code is likely to be so time consuming that it will fail a cost-benefit analysis. CMS users also often lack ongoing access to developers who can maintain complex code customizations. How can you get the best of two CMS worlds in one website, with minimal effort?

We're not talking about parallel systems where some website content and features are in one system, and some are in another, and they inhabit slightly different parts of the same domain. To regular users, that might look OK, but it's a cumbersome option in terms of themeing (you need to manage the appearance of two systems instead of one) and in terms of interactive features (you quickly run into single sign-on issues.)It would be better if somehow you could simply get the features you want from one CMS inside the other. But how can this be done?

We will take the example of integrating Drupal and Formulize to show how such an arrangement is possible.

It's Just an Extension of HTML

Let's step back and remember that PHP is just an enhancement to answering an HTTP request. Drupal does not answer the HTTP request; the web server does. The web server then searches for the file that was requested, and if the file is PHP, then the instructions in that file are processed, and the web server continues from there until all the included files and code have been processed. The web server then returns a stream of HTML to the client based on all the instructions just read.

There's no reason in principle that you can't have as much PHP code as you would like answer the page request. With little effort, a single request can result in both CMSs starting up and generating content for the HTML stream. All CMSs have a relatively simple process that bootstraps their template system, their connection to their database, their internal API, and so on.

In WordPress, you include `wp-load.php`, as seen in the `wp-blog-header.php` file:

```
if ( !isset($wp_did_header) ) {
    $wp_did_header = true;
    require_once( dirname(__FILE__) . '/wp-load.php' );
    wp();
    require_once( ABSPATH . WPINC . '/template-loader.php' );
}
```

In Drupal, you call `bootstrap.inc`, as seen in the `index.php` file:

```
define('DRUPAL_ROOT', getcwd());
require_once DRUPAL_ROOT . '/includes/bootstrap.inc';
drupal_bootstrap(DRUPAL_BOOTSTRAP_FULL);
menu_execute_active_handler();
```

In Formulize, you call `mainfile.php` from any page (Formulize, like ImpressCMS and XOOPS, which it derives from, uses a page controller architecture, so every page needs to bootstrap the system first before starting its own logic), for example in Listing 1. XOOPS is designed to make component interchange possible, and this makes our task easier.

LISTING 1

```
01. require_once '../../mainfile.php';
02. include XOOPS_ROOT_PATH.'/header.php';
03. global $xoTheme;
04. if($xoTheme) {
05.     $xoTheme->addStylesheet(
06.         '/modules/formulize/templates/css/formulize.css'
07.     );
08.     $xoTheme->addScript(
09.         '/modules/formulize/libraries/formulize.js'
10.     );
11. }
12. include 'initialize.php';
13. include XOOPS_ROOT_PATH.'/footer.php';
```

Bootstrapping Two Systems from One HTTP Request

The next question is how to get both systems to bootstrap as part of the same page request. To solve this, you have to first determine which system is answering the page request. An implicit assumption here is that both systems are installed on the same web server. There would be ways to get this to work across servers if you allowed remote includes and tweaked some other settings.

One system needs to be primary, and the other is secondary. The http request is asking for a file from only one of the systems. At some point during the running of that primary system, the secondary system needs to start up.

This can be accomplished by modifying the core files of the primary CMS, for example, a one line change in Drupal's `index.php` file will trigger a Formulize bootstrap as part of every request:

```
define('DRUPAL_ROOT', getcwd());
// BOOTSTRAP THE PRIMARY SYSTEM
require_once DRUPAL_ROOT . '/includes/bootstrap.inc';
// BOOTSTRAP THE SECONDARY SYSTEM
require_once 'path/to/Formulize/mainfile.php';
drupal_bootstrap(DRUPAL_BOOTSTRAP_FULL);
menu_execute_active_handler();
```

That will work, but modifying core files is rarely recommended for reasons that should be obvious! Sometimes, a good shortcut is your best friend, but not always. This also has the drawback of invoking the secondary system on each request when you likely only want it during specific pages.

LISTING 2

```
01. // LISTEN FOR MODULE INITIALIZATION BY DRUPAL
02. function formulize_init() {
03.     //...
04.     // GET PATH TO FORMULIZE, WHICH IS USER-SPECIFIED IN THE
05.     // ADMIN UI
06.     $formulize_path = variable_get('formulize_full_path', NULL);
07.     $formulize_path = is_dir($formulize_path)
08.         ? rtrim($formulize_path, '/\\')
09.         : dirname($formulize_path);
10.     $integration_api_path = $formulize_path
11.         . DIRECTORY_SEPARATOR . 'integration_api.php';
12.     //...
13.     include_once($integration_api_path);
14.     //...
15. }
```

A more robust solution is to find a way to take advantage of the primary CMS's extension capabilities and extend the extension to bootstrap the secondary CMS only when required. For example, (Listings 2 and 3) we have created a module for Drupal that bootstraps Formulize when it needs to display Formulize content inside Drupal.

Inside `INTEGRATION_API.PHP` **LISTING 3**

```
01. class Formulize {
02.     ...
03.     static function init() {
04.         // INIT IS CALLED INTERNALLY BY ALL API METHODS
05.         ...
06.         // BOOTSTRAP THE SECONDARY SYSTEM
07.         include_once('mainfile.php');
08.         ...
09.     }
10.     ...
11. }
```

The last issue to consider at this point is whether all the objects and variables you will rely on are actually available. Many CMSs assume that their key objects and variables are in the global space, and so sprinkled throughout the rest of their code, you see things like this:

```
global $user;
```

The code can then interact with the global `$user` object. The `$user` object is most likely created during the bootstrap process, which is typically assumed to have taken place in the global namespace. But if you end up bootstrapping the CMS from inside a function, then `global $user` won't work any longer in all your dependent code. If that is the case, then you need to add the `global` keyword before the declaration of all relevant variables. That will ensure the variables are in the global space, even if they are created inside the scope of a function.

This is a bit of a pain, requiring an audit of the bootstrap process to identify where each of the important variables and objects are instantiated, ensuring that they are assigned to the global namespace explicitly, and making sure they won't clobber one another. In more modern architectures, PHP's namespace feature may be in use, and that could simplify this issue quite a bit.

Loose Ends

Now you have both systems bootstrapped and fully available as part of each page request in Drupal. At this point, you probably have some other low-level challenges to overcome.

All CMS's have some kind of security layer that validates certain things about the request and tries to prevent things like CSRF attacks and other things you don't want to happen. In the Formulize codebase, this includes some older code that still checks the referrer information from the browser to verify that the request came from a URL that is part of the same website. While this is not the most useful part of the security mechanism, it can't hurt (except when security software on a PC blocks referrer information)!

The problem is, the referrer check fails when Formulize bootstraps during a request initiated from a Drupal page because it's a different "website." This is easily addressed by neutering the referrer check. Because it is such a low-value part of the security process, it's not a significant risk to turn it off:

```
public function checkReferer($docheck = 1) {
    return true;
    // function continues below
```

Any two systems that you try to tie together this way will lead to some behind-the-scenes issues like this. If they are serious, you will need to spend a bit of time tracking them down.

Another low-level issue you need to consider is the way each system relates to the database(s). It is often possible to have two systems share a single database if one system can use a common prefix on its tables names (to avoid naming collisions). However, it is also possible for each system to connect to its own database on the same database server or even to connect to its own database on a distinct database server. It all depends on how that part of your CMS installation has been configured.

In the case of the Formulize codebase, the legacy `mysql_connect` function can be used to establish the database connection as well as the newer PDO library. Which gets used depends on the settings chosen at the time of installation. If the `mysql_connect` function is used, and the Formulize database is on the same server as the primary system's database, it is possible that the primary system's connection to its database will be replaced by the connection to the Formulize database. Fortunately, this is rare due to the prevalence of PDO now, and it has a simple solution in any case (see Listing 4).

LISTING 4

```
01. if (XOOPS_DB_PCONNECT == 1) {
02.    // FINAL 'TRUE' FORCES A NEW CONNECTION INSTEAD OF
03.    // REUSING EXISTING
04.    $this->conn = mysql_pconnect(
05.       XOOPS_DB_HOST, XOOPS_DB_USER, XOOPS_DB_PASS, true
06.    );
07. } else {
08.    // FINAL 'TRUE' FORCES A NEW CONNECTION INSTEAD OF
09.    // REUSING EXISTING
10.    $this->conn = mysql_connect(
11.       XOOPS_DB_HOST, XOOPS_DB_USER, XOOPS_DB_PASS, true
12.    );
13. }
```

Session Integration

The most critical low-level issue is session integration, or single sign on. To provide your users with any significantly useful behavior in your cojoined set of CMSs, the secondary system needs to be aware of which user is logged in in the primary system.

Most CMSs have a table of users in their database. Each user has a primary key in that table, a user ID, which is referenced elsewhere in other tables. An extremely simple approach to handling single sign on is to

establish a pattern/rule in the real world, manually, so that when new users are created in the primary system, a corresponding user is created in the secondary system. If you manage to do this in the same sequence in both systems, the user IDs will be equivalent. This is not a viable solution for all cases, but we can start with it, if only to illustrate the concept: if the IDs of the two user records are identical, then single sign on can be achieved relatively easily.

For example, at some point in the bootstrapping process for all CMSs, the active user is determined and its session information is loaded from the database or elsewhere. If the primary system bootstraps first, then it will have determined which user is active, and it will have created some global object or other representation of the active user. The secondary system can simply refer to this information to determine the active user, and if the user IDs between the two systems are in sync, the final step would be to force the secondary system to "log in" the appropriate user.

At that point, all you need to do during session initialization in the secondary system is add something like this:

```
// ADDITIONAL CODE TO IDENTIFY THE ACTIVE USER
global $user;
if ($user) {
   $_SESSION['activeUser'] = $user->uid;
}

// CONTINUE WITH NORMAL SESSION INITIALIZATION
if ($_SESSION['activeUser']) {
```

A more robust solution is to create a translation table that records which user ID in the secondary system corresponds to which user ID in the primary system. This is relatively simple in any CMS that has any type of "event" system that triggers events when users are created, updated, or deleted.

The Formulize module in Drupal maintains this translation table by responding to the Drupal "hook" system (which is an API convention for responding to various standard events in Drupal).

In Drupal, when a new user is created in Listing 5.

LISTING 5

```
01. function formulize_user_insert($edit, $account, $category) {
02.    if (!_formulize_integration_init()) {
03.        return;
04.    }
05.
06.    $user_data = array(
07.       'uid' => $account->uid,
08.       'uname' => $account->name,
09.       'login_name' => $account->name,
10.       'name' => $account->name,
11.       'pass' => $account->pass,
12.       'email' => $account->mail,
13.       'timezone_offset' => $account->timezone/60/60,
14.       'language' => _formulize_convert_language(
15.                        $account->language
16.                     ),
17.       'user_avatar' => 'blank.gif',
18.       'theme' => 'impresstheme',
19.       'level' => 1
20.    );
21.
22.    $user = new FormulizeUser($user_data);
23.    Formulize::createUser($user);
24.
25.    // Add user to groups
26.    foreach ($account->roles as $roleid => $rolename) {
27.       Formulize::addUserToGroup($account->uid, $roleid);
28.    }
29. }
```

In the Formulize integration API (Listing 6).

LISTING 6

```
01. static function createUser($user_data) {
02.    self::init();
03.    if($user_data->get('uid') == -1) {
04.       throw new Exception('Formulize::createUser() - '
05.         .'The supplied user doesn\'t have an ID.');
06.    }
07.
08.    //Create a user from the provided data
09.    $member_handler = xoops_gethandler('member');
10.    $newUser = $member_handler->createUser();
11.    $newUser->setVar('uname', $user_data->get('uname'));
12.    $newUser->setVar(
13.       'login_name', $user_data->get('login_name')
14.    );
15.    $newUser->setVar('email', $user_data->get('email'));
16.    $newUser->setVar(
17.       'timezone_offset', $user_data->get('timezone_offset')
18.    );
19.    $newUser->setVar(
20.       'notify_method', $user_data->get('notify_method')
21.    ); //email
22.    $newUser->setVar(
23.       'level', $user_data->get('level')
24.    ); //active, can login
25.
26.    if ($member_handler->insertUser($newUser, true)) {
27.       //Map the created user to the external ID provided
28.       $user_id = $newUser->getVar('uid');
29.       $return = self::createResourceMapping(
30.          self::USER_RESOURCE,
31.          $user_data->get('uid'),
32.          $user_id
33.       );
34.       if ($return > 0) {
35.          /* 2 is registered users group */
36.          $member_handler->addUserToGroup(2, $user_id);
37.       }
38.       return $return;
39.    }
40. }
```

Once that is done, you can refer to the translation table to identify which "secondary" user corresponds to which "primary" user. In Formulize, there is a method in the API called `getXoopsResourceID` that looks up the "external" ID and returns the ID of the corresponding resource in the local system.

During the module initialization in Drupal:

```
global $user, $formulizeHostSystemUserId;
$formulizeHostSystemUserId = $user->uid;
```

During the session initialization in Formulize (Listing 7).

It's the Content, Stupid

With all this administrative work out of the way, you now have two fully integrated CMSs, with single sign on, capable of contributing together to an http request! Just one question: how do you actually integrate the content?

We need a straightforward way to merge content from the secondary system with the page that is being generated in the primary system.

The first thing to keep in mind is the concept that one system is primary, and the other is secondary. All CMSs have some kind of templating system, some way that the various parts of the page that have been generated during the request, are merged into a single stream of HTML that gets sent to the client. The question for us to consider at this point is which part of the page is the part where we want our secondary system's content to appear.

LISTING 7

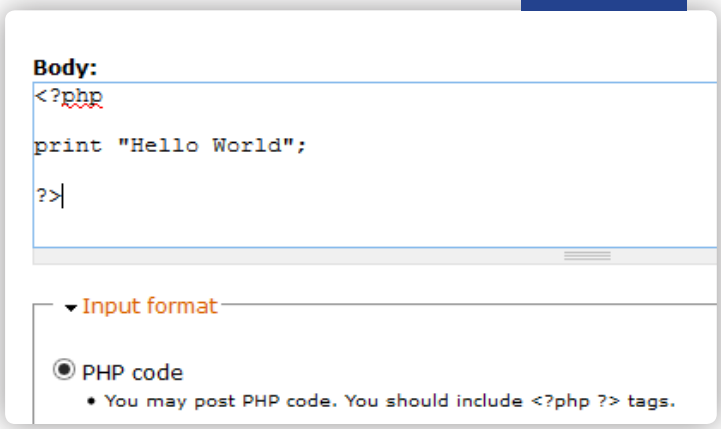
```
01. if (isset($GLOBALS['formulizeHostSystemUserId'])) {
02.
03.    if ($GLOBALS['formulizeHostSystemUserId']) {
04.       $externalUid = $GLOBALS['formulizeHostSystemUserId'];
05.    } else {
06.       $externalUid = 0;
07.       $cookie_time = time() - 10000;
08.       $instance->update_cookie(session_id(), $cookie_time);
09.       $instance->destroy(session_id());
10.       unset($_SESSION['xoopsUserId']);
11.    }
12.
13. }
14.
15. if ($externalUid) {
16.
17.    $xoops_userid = Formulize::getXoopsResourceID(
18.       Formulize::USER_RESOURCE, $externalUid
19.    );
20.    $icms_user = icms::handler('icms_member')
21.                      ->getUser($xoops_userid);
22.    if (is_object($icms_user)) {
23.       // set a few things in $_SESSION, similar to what
24.       // include/checklogin.php does, and make a cookie
25.       // and a database entry
26.       $_SESSION['xoopsUserId'] = $icms_user->getVar('uid');
27.       $_SESSION['xoopsUserGroups']
28.          = $icms_user->getGroups();
29.       $_SESSION['xoopsUserLastLogin']
30.          = $icms_user->getVar('last_login');
31.       $_SESSION['xoopsUserLanguage']
32.          = $icms_user->language();
33.       $_SESSION['icms_fprint']
34.          = $instance->createFingerprint();
35.       $xoops_user_theme = $icms_user->getVar('theme');
36.
37.       if (in_array($xoops_user_theme,
38.                    $icmsConfig['theme_set_allowed'])
39.          ) {
40.          $_SESSION['xoopsUserTheme'] = $xoops_user_theme;
41.       }
42.       $instance->write(session_id(), session_encode());
43.
44.       // need to use the current maxlifetime setting, which
45.       // will be coming from Drupal, so the timing of the
46.       // sessions is synced.
47.       $icms_expiry = ini_get("session.gc_maxlifetime") / 60;
48.       $cookie_time = time() + (60 * $icms_expiry);
49.       $instance->update_cookie(session_id(), $cookie_time);
50.    }
51. }
```

Typically, this would be the body section of the page. Most CMSs provide some way to invoke PHP commands in the body section of the page. Or you can use the extension capabilities of the CMS to create some new type of body content. For example, with the right settings turned on in Drupal, you can type PHP commands directly into the body section of a page (Figure 1).

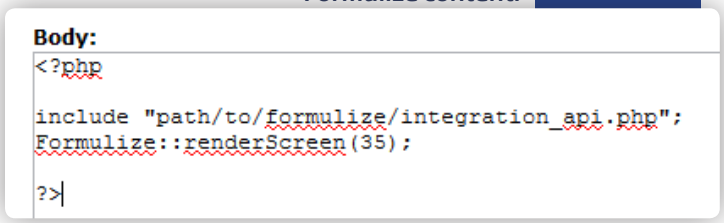
This is an extremely useful trick as long as your secondary CMS has a convenient API for generating/getting the content of its various pages and components. You can simply use PHP commands to essentially embed the content from the secondary CMS inside the primary CMS. In the case of Formulize, each screen that a user can interact with can be easily invoked by a few lines of PHP (Figure 2).

Those lines generate the entire body section of the page (Figure 3).

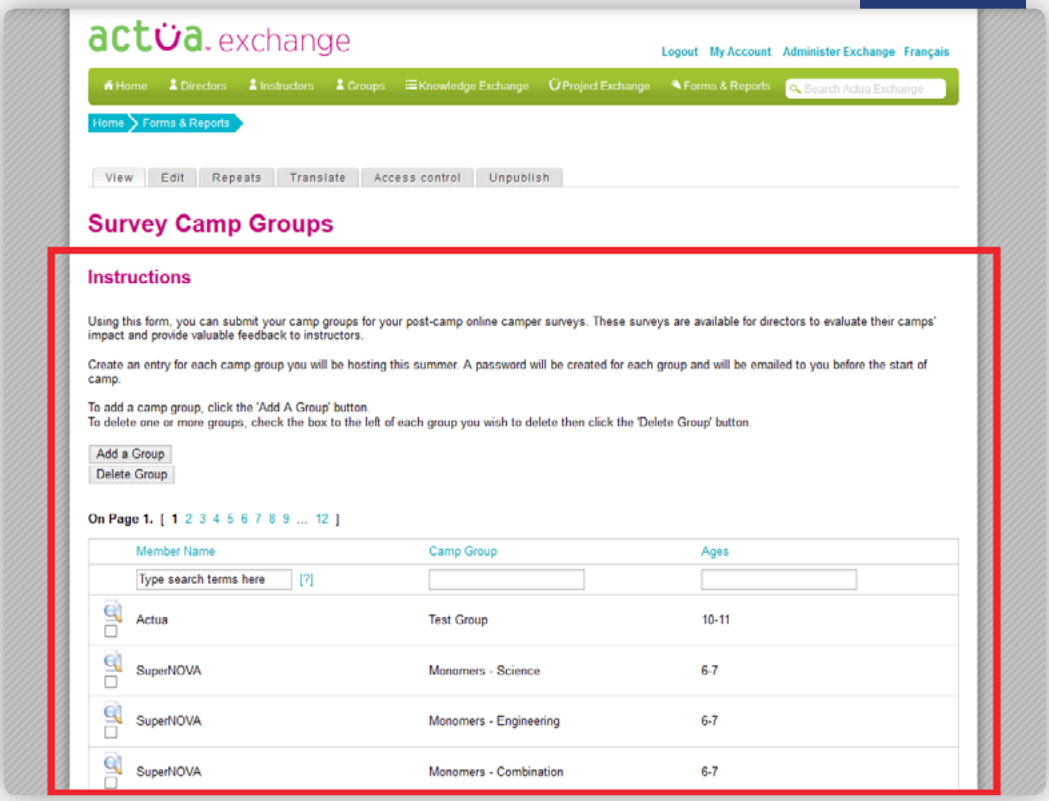
PHP code going into a Drupal page **FIGURE 1**



A Drupal page with PHP code for invoking Formulize content. **FIGURE 2**



The rendered page in Drupal (content from Formulize highlighted in red) **FIGURE 3**



This same behavior is typically easy to achieve using your CMS's extension capabilities. For example, in Drupal, the same module we use to bootstrap Formulize and synchronize the users can also generate the screen contents and make them available to the Drupal templating system:

Integrating the systems through the module means webmasters can use the admin UI to add content from one system to the other rather than having to type in PHP code.

LISTING 8

```
01. function formulize_view($node, $view_mode) {
02.    if (_formulize_integration_init()) {
03.       drupal_add_css(
04.          drupal_get_path('module', 'formulize')
05.          . '/formulize.css'
06.       );
07.       Formulize::init();
08.       drupal_add_js(XOOPS_URL
09.          . '/modules/formulize/libraries/formulize.js');
10.
11.       // start capturing output from Formulize screens
12.       ob_start();
13.       Formulize::renderScreen($node->screen_id);
14.       $output = ob_get_clean();
15.       $node->content['formulize_screen'] = array(
16.           '#markup' => $output,
17.           '#weight' => 1
18.       );
19.    }
20.    return $node;
21. }
```

Links of Doom

There is one big challenge that arises at this point: the generation of links inside the content from the secondary CMS.

CMSs are generally architected based on the assumption that they are the single thing responsible for answering the http request. They often set up some constants or variables during the bootstrap process to represent the canonical root path to the installed CMS software on the server along with the canonical URL at which users can reach the site.

Those constants are reused again and again when the CMS generates the page contents. This is similar to how many frameworks operate as well.

A typical link in a CMS would be generated by code such as this:

```
$link = $base_url . "/profile/?user=" .
            intval($user_id);
```

The problem comes when such a link from the secondary CMS is embedded inside the primary CMS's page, when the user clicks on the link and is then taken to a page governed by the secondary CMS. At that point, all your careful work integrating the two systems is broken, as the new page request is not being answered by the primary system.

This problem does not arise for images and Javascript files and other dependent resources. They can be loaded without issue from their native locations inside the file structure of the secondary CMS. An issue only arises when the user activates a link that is based on the base URL of the secondary CMS and essentially "breaks out" of the primary CMS at that moment.

There are three possible solutions to this problem.

First, this issue highlights a conceptual shortcoming of the standard model of setting up a constant for the root path to the files and a constant for the base URL. Essentially, a third constant might help in some cases, something you could call the "deployment URL" to represent the address on the web where the secondary system's content will be deployed, distinct from the address where the secondary system is installed.

The secondary system's code could then be modified in certain places to generate links based on the "deployment URL" instead of its own base URL. In addition, you would need to take into account the addressing and aliasing scheme in the primary CMS. After all, in the example above, `/profile/?user=27` will not lead to a valid page in the primary CMS. Therefore, the secondary CMS would need to be altered in a way that caused it to generate URLs that pointed to valid pages inside the primary CMS.

Second, you could alter the secondary CMS so that it used the templating system from the primary CMS. That way, when people landed on the `/profile/?user=27` page, the secondary CMS would answer the request, but the contents would be presented inside the standard page template that the primary CMS is using. This is the most foolproof approach, as it would theoretically work for any address inside the secondary CMS.

Some combination of the first and second approaches could likely address most situations. However, there is a third option. This is what we have used when integrating Formulize with other systems, and it allows Formulize to work no matter what other system it is installed with.

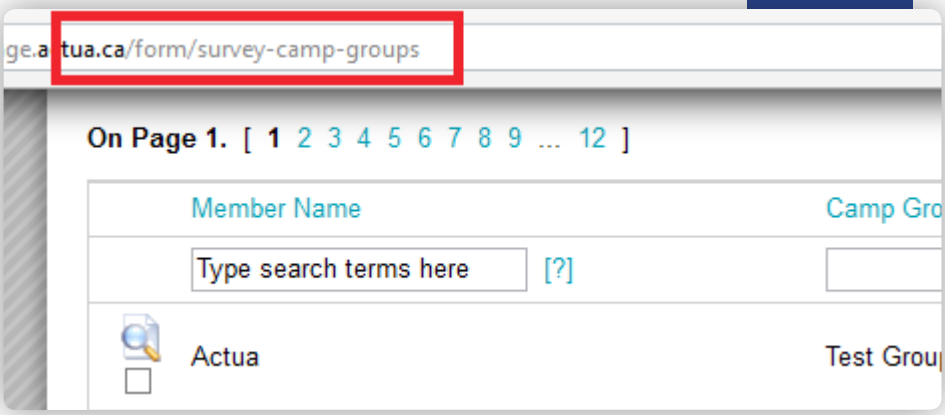
Reuse the Current URL

Formulize doesn't use a base URL concept at all. Instead, it detects the current URL being used for the active http request. It then uses that URL as the destination of all links and the action URL for all forms and so on, regardless of whether that URL is part of the primary or secondary system.

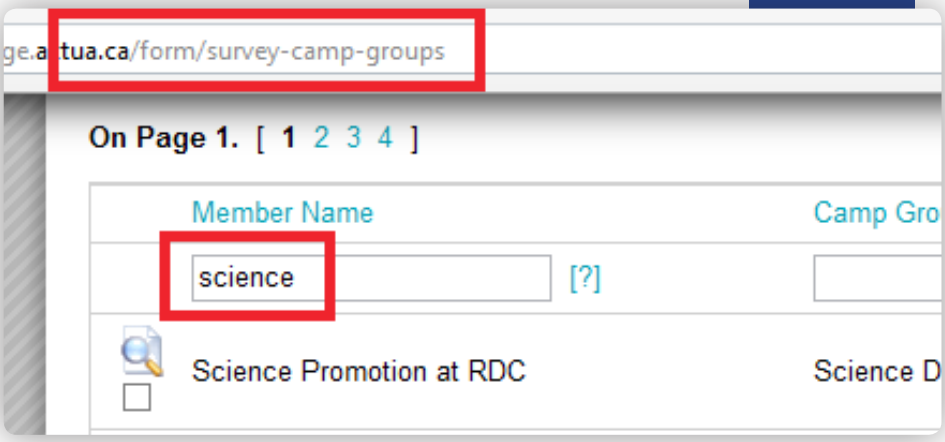
Formulize therefore keeps requesting the same page from the server over and over. All "state" information about what a user has clicked on or what settings users have altered is posted with the request, so we can then see in `$_POST` exactly what the user has done and respond accordingly. For example, Figure 4 shows the URL and page after an initial page load:

Figure 5 shows the page after a reload, after a search term has been submitted.

The initial page after the screen has loaded **FIGURE 4**



After a search term has been submitted, note the URL **FIGURE 5**



Formulize was originally designed this way so that Formulize content could be rendered equally well in the body section of the page or inside a block on another page at a different location in the same CMS. This architectural approach turned out to be perfect for deploying Formulize content inside any other CMS as well.

So What?

With a few conceptually simple steps, it is possible to integrate two (or more) CMSs so they can collaborate in delivering content for a website. The major challenge lies not in tying the two together but occurs when deploying content from one to the other. Architectural assumptions underlying how the systems are built can make this easy or hard. If you can overcome that barrier for your given use case, then you can break out of a CMS's walled garden and reap the benefits of a module or feature from a secondary CMS that is not available in your primary CMS.

This can give your employer or client much greater flexibility in terms of what systems they use and how. It can also avoid the need to migrate 100% of content and features from one system to another when a decision has been made to move from an old CMS to a new one.

JULIAN EGELSTAFF has been working in the software and IT industries for over 18 years, and has been building and using content management systems for most of that time (since PHP 3). In 2003, he co-founded Freeform Solutions, a not-for-profit organization with a mission to help other not-for-profits use technology more effectively. Today, Freeform helps dozens of organizations around the world, specializing in Drupal and CiviCRM.

Julian is also the lead developer of Formulize, an open source project designed to bring the easy deployment and configuration of CMSs, to the world of databases. Formulize lets people quickly create complex data management, reporting and workflow systems in the same way CMSs let you quickly configure complex websites. Freeform uses Formulize to extend the capabilities of Drupal and other systems, when client needs cross over from content to data.

Julian speaks regularly at technology events and conferences, on a variety of topics ranging from technical aspects of website development, to general privacy and security issues online. He holds a Bachelor of Journalism and Philosophy, and is a Zend Certified Engineer.

Twitter: @jegelstaff

Day Camp 4 Developers

AND

php[architect]

PRESENT

Debugging Beyond var_dump()

September 18, 2015
9:00 AM – 3:00 PM CDT
Online

Gone are the days when var_dump() & die() would show you what was actually going on in your app. When your website was all in single pages, var_dump() was your best friend. With today's advanced frameworks however, you have to have more; you have to have a debugging strategy. **Day Camp 4 Developers: Debugging Beyond var_dump()** will help you develop that strategy.

We've brought together five experts in various areas to help you understand the tools that are available to you, and the mindset you need to debug your code. Join us on Sept 18th, 2015 for **Day Camp 4 Developers: Debugging Beyond var_dump()**. Purchase your ticket today.

Register now at daycamp4developers.com

PHP Profiling, an Introduction	Debugging: Past, Present and Future	Characterization Testing for Legacy Applications	Modern Tools for API Debugging and Testing	Don't Reboot, Debug!
				
Fabien Potencier	Derick Rethans	Paul M. Jones	Neil Mansilla	Joshua Thijssen

Day Camp 4 Developers, invest a day in your career.

Want more articles like this one?

Keep your skills current and stay on top of the latest PHP news and best practices by reading each new issue of php[architect], jam-packed with articles.

Learn more every month about frameworks, security, ecommerce, databases, scalability, migration, API integration, devops, cloud services, business development, content management systems, and the PHP community.

php[architect]

magazine

books

conferences

training

phparch.com

Get the complete issue for only $6!

We also offer digital and print+digital subscriptions starting at $49/year.