Helpers and Addons

lesson #lesson06

James L. Parry
B.C. Institute of Technology

Agenda

1. Building on CodeIgniter
2. A CodeIgniter Library (Caboose)
3. A CodeIgniter Helper (formfields)
4. A CodeIgniter Package
5. Package Management

BUILDING ON CODEIGNITER

So far, your webapp has been assembled using controllers. models and views. Helpers and libraries are two other kinds of components you can use in a CI webapp.

CodeIgniter also has the notion of packages, which are larger sets of the above kinds of comnponents.

Loading Components

When CodeIgniter loads components, it looks first in the appropriate subfolder inside `application`, and then in the appropriate subfolder inside the framework's `system` folder.

The CodeIgniter loader applies its own naming rules when looking for components, for instance looking only for the "ucfirst" filename corresponding to a library, or appending an appropriate extension ("_helper") when looking for a helper. You already know to watch out for this if developing on Windows and deploying on *nix!

Helper Basics

Helpers are files that contain one or more procedural functions. A number of them are built into CodeIgniter, as shown in the folder list to the right.

Helpers can be explicitly loaded, as in `$this->load->helper('whatever');`, or they can be autoloaded in `application/config/autoload.php`, for instance `$autoload['helper'] = array('url');`. Use the name of the helper without the extension.

Helper functions are not O-O, but they are simpler than having classes with static methods and referring to those.

```
[jim@new-host-16 helpers]$ ls
array_helper.php      form_helper.php       smiley_helper.php
captcha_helper.php    html_helper.php       string_helper.php
cookie_helper.php     index.html            text_helper.php
date_helper.php       inflector_helper.php  typography_helper.php
directory_helper.php  language_helper.php   url_helper.php
download_helper.php   number_helper.php     xml_helper.php
email_helper.php      path_helper.php
file_helper.php       security_helper.php
```

Writing Helpers

Each helper file is meant to hold a set of *related* functions.

One good practice is to ensure that helpers cannot be loaded outside the context of CodeIgniter. This is done by making sure that the BASEPATH constant is defined.

Another good practice, with functions, is to make sure they are not already defined (which would trigger a PHP error). This is done using the `function_exists` function built into PHP.

A typical helper excerpt...

```
<?php
defined('BASEPATH') OR exit('No direct
script access allowed');

if ( !
function_exists('do_something_useful'))
{
  function do_something_useful(...) {
    ...
  }
}
```

Extending Built-In Helpers

You can replace functions in a built-in helper, or add your own to them, by "extending" the helper.

Create a helper named the same as the one you want to enahnce, but with the subclass prefix (MY_), and in your application/helpers folder.

It will automatically be loaded before the built-in helper, and any functions you define will be used in preference to the built-in ones.

In application/helpers/MY_foo_helper.php:

```
function a() {...}
function b() {...}
```

In system/helpers/foo_helper.php:

```
function a() {...}
function c() {...}
```

In your controller using "foo":

`$this->load->helper('foo');`

functions a and b come from your code, while function c comes from the built-in.

Library Basics

Libraries are files that contain class definitions - usually one per file.

Libraries can be explicitly loaded, as in `$this->load->library('whatever');`, or they can be autoloaded in `application/config/autoload.php`, for instance `$autoload['libraries'] = array('parser');`.

Library functions are O-O.

```
[jim@new-host-16 libraries]$ ls
Cache           Form_validation.php  Pagination.php  Unit_test.php
Calendar.php    Ftp.php              Parser.php      Upload.php
Cart.php        Image_lib.php        Profiler.php    User_agent.php
Driver.php      index.html           Session         Xmlrpc.php
Email.php       Javascript           Table.php       Xmlrpcs.php
Encryption.php  Javascript.php       Trackback.php   Zip.php
Encrypt.php     Migration.php        Typography.php
[jim@new-host-16 libraries]$
```

Writing Libraries

Each library is meant to hold a single useful class.

One good practice is to ensure that libraries cannot be loaded outside the context of CodeIgniter. This is done by making sure that the `BASEPATH` constant is defined.

You don't have to worry about libraries being loaded twice - the CodeIgniter loader takes care of that.

A typical library excerpt...

```
<?php
defined('BASEPATH') OR exit('No direct
script access allowed');

class Whatever {
  function do_something_useful(...) {
    ...
  }
}
```

Extending Built-In Libraries

You can extend a built-in library, replacing or adding methods in an "O-O" fashion. Create a library named the same as the one you want to enahnce, but with the subclass prefix (`MY_`), and in your application/libraries folder.

It will automatically be loaded whenever you use the CI loader to pull in the "original" one.

If the source file contains other classes, they can be instantiated or extended in your logic, after loading. You've seen this with `Welcome extends Application`.

In application/libraries/MY_Foo.php:

```
class MY_Foo extends CI_Foo {   function
a() {...}
  function b() {...} }
```

In system/libraries/Foo.php:

```
class CI_Foo {
}
```

In your controller using "foo":

```
$this->load->library('foo');
```

Your controller will end up with a "foo" instance injected as a property, namely `$this->foo`

Package Basics

Packages are collections of related components organized in a file hierarchy similarly to `application/`.

Packages are not "loaded", in the conventional sense - their folders are added to the appropriate list of folders to search when the CI loader is invoked. Packages can be explicitly specified, as in `$this->load->add_package_path(APPPATH.'third_party' '/some_package');`, or they can be autoloaded in `application/config/autoload.php`, for instance `$autoload['packages'] = array(APPPATH.'third_party', '/some_package');`.

Packages contain most things that might be in an app, with one glaring exception ... can you spot it?

```
/application/third_party/some_package/

config/
helpers/
language/
libraries/
models/
views/
```

Writing Packages

The components in a package are written and referenced "normally". They become "visible" when the package's path is added to the loader.

Package paths are pre-pended to the list of folders to search, in the order that you add them, i.e. LIFO.

"Normal" component loading rules apply inside a package too.

Packages are meant to be self-contained.

Packages will often come from outside projects.

Naming collisions are possible, and not properly dealt with - namespaces in a future CodeIgniter will fix that.

Extending Built-In Packages?

No packages are built-in to CodeIgniter.

There are not a lot of good examples, at the moment.

A suggested best practice is to "install" a package by creating a folder for it, inside `application/third_party`, and crafting a "composer" file to manage pulling and updating the files that comprise it, from an external site.

Treat this notion of packages as CodeIgniter-specific.

A CODEIGNITER LIBRARY (CABOOSE)

*An **example** library, to help your understanding.*

Plain HTML doesn't cut it in your views :(

CSS frameworks improve presentation substantially :)

But ... they involve CSS, Javascript & often initialization

Caboose is a library that manages these frameworks & widgets

Frameworks & Widgets

In the context of presentation, a framework is a complete set of CSS and/or Javascript files that provide styling and behaviour for your webpages. Examples include BootStrap or jQuery.

These can be deployed from a content distribution network or as part of your webapp. If they are pervasive in your webapp, you would normally add links to them in your master template, so they are referenced on every page.

Caboose solves the problem of managing the individual add-on components, referred to as "widgets", or frameworks that are only used on some of your pages.

Caboose Perspective

What Caboose assumes to be a managed widget...

- Zero or more CSS files, to be included in your page's `<head>`
- Zero or more javascript files, to be included just before `</body>`
- An optional component template, holding a javascript snippet needed to initialize the library/widget

Caboose Assumptions

Caboose assumes that you will store the bits and pieces for a component in specific subfolders, `assets/css` and `assets/js` in your webapp's document root.

It further assumes that you will tailor any needed widget initialization using a parser template inside your `view/components` folder.

Caboose Configuration

Components recognized by Caboose are defined in its `$components` array, using names of your choice.

Each component name references an array of settings for that component.

The `css` and `js` settings can themselves contain an array of filenames to include, if appropriate.

Configuration for a sample datepicker component:

```
$components = array(
  ...
  'time' => array(
    'css' => 'bootstrap-timepicker.css',
    'js' => 'bootstrap-timepicker.js',
    'template' => 'time'
  ),
...);
```

Caboose Usage - Field Level

Once the library is loaded, you basically tell it whenever you need to use a component...
`$this->caboose->needed('whatever','fieldname');`

"whatever" is the name of the widget, and "fieldname" is the ID of a component to be bound to that widget.

The library will keep track of all of the widgets referenced, and generate/accumulate the needed Javascript initialization for each of them.

An example, inside a controller method that will result in a webpage with a datepicker that we want associated with the input text field "asof_date":

```
$this->caboose->needed('time','asof_date')
```

Caboose Usage - Page Level

The Caboose class has three methods to return the constructed HTML/Javascript. In my base controller's `render()`, I use these to set view parameters:

```
// convert Caboose output into view parameters
$this->data['caboose_styles'] = $this->caboose->styles();
$this->data['caboose_scripts'] = $this->caboose->scripts();
$this->data['caboose_trailings'] = $this->caboose->trailings();
```

Inside my master template, I include those view parameters:

```
<html>
  <head>
    ...
    {caboose_styles}
  </head>
  <body>
    ...
    {caboose_scripts}
    {caboose_trailings}
  </body>
</html>
```

Caboose Notes

The Caboose library *is* a library, but there are some awkward things you might have noticed:

- Configuration is done inside the class. Wouldn't it be better to handle this inside a config file? That would make the library more generalized.
- The library relies on some view fragments in the `views` folder. This means we have several places in our webapp to change :(
- What if we wanted to store our assets somewhere other than in `assets`? Shouldn't that be a variable, constant, or configuration item?

Hmmmm - do you remember mention of something that would let us bundle related classes, configuration and views?

A CODEIGNITER HELPER (FORMFIELDS)

The **Formfields** helper is a collection of functions, and corresponding view fragments, to deal with the styling of HTML form fields.

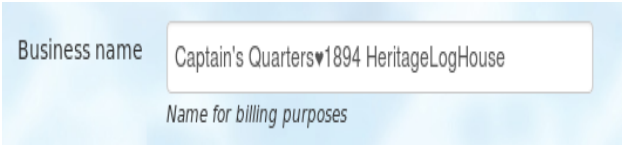
Each function builds a standard array of view parameters, and then invokes the template parser to style the form field for the CSS/JS framework(s) you are using.

So what? This adds a layer of abstraction, so that you don't worry about form styling in your controller logic. Further, should you wish to switch CSS or Javascript frameworks, all you have to do is update your set of view fragments :)

Sample Formfields Function

To the right is a typical function from the Formfields helper. It looks complicted for just a text field, but includes a number of standard parameters used in the field styling, such as standard help text.

The end result looks something like...




```
function makeTextField($label, $name,
$value, ...) {
  $CI = &get_instance();
  $parms = array(
    'label' => $label,
    'name' => $name,
    'value' =>
htmlentities($value,ENT_COMPAT,'UTF-8'),
    'explain' => $explain,
    'maxlen' => $maxlen,
    'size' => $size,
    'disabled' =>
($disabled?'disabled="disabled"':'')
  );
  return
$CI->parser->parse('_fields/textfield',
$parms, true);
}
```

Sample Formfields View Fragment

To the right is a view fragment that might correspond to the function on the previous slide.

It looks ugly, but only because of the Bootstrap styling.

```
<div class="control-group">
  <label class="control-label"
for="lesson06">{label}</label>
  <div class="controls">
    <input type="text" id="lesson06"
name="lesson06"
       value="{value}" maxLength="
{maxlen}"
      style="width:{size}em;height:2em"
{disabled}/>
    <br/><small><em>{explain}
</em></small>
  </div>
</div>
```

Formfields Usage

How do we use a helper like this? In your controller, you could construct the HTML for a given textfield something like...

```
$data['fuserid'] =
makeTextField('Userid', "userid",
    $member->userid,
    "Login ID for this member", 10, 10);
```

To use the result in your view, it might look something like...

```
<fieldset id="group3">
  <legend>(b) Listing Details</legend>
  {fuserid}
  {fbiz_email}
  {fxdesc_web}
</fieldset>
```

Formfields Helper - So What?

Separated control logic from presentation

Our views are *much* simpler, and do not have to involve PHP, Javascript, or customization!

Change the view fragments to suit the framework(s) you want to use (use the caboose, luke)

Add function/view fragment pairs to suit your application

Formfields Helper - Show Me!

Ok - here is the HTML for a file uploading widget. You might switch to outline view to see it better! ;p

```
<div class="control-group">
  <label class="control-label" for="lesson06">{label}</label>
  <div class="controls">
    <div class="fileupload fileupload-new" data-provides="fileupload">
      <div class="fileupload-new thumbnail" style="width: 120px; height: 80px;"><img
src="http://www.placehold.it/50x50/EFEFEF/AAAAAA" /></div>
      <div class="fileupload-preview fileupload-exists thumbnail" style="width:
120px; height: 80px;"></div>
      <span class="btn btn-file">
        <span class="fileupload-new">Select file</span>
        <span class="fileupload-exists">Change</span>
        <input type="file" name="lesson06"/>
      </span>
      <a href="#" class="btn fileupload-exists" data-dismiss="fileupload">Remove</a>
      <br/><small><em>{explain}</em></small>
      <br/>Any file selected above will be uploaded when you submit this form.<br/>
    </div>
  </div>
</div>
```

Formfields Helper - And...?

Make a simple image uploader function for that...

```
function makeImageUploader($label,
$name,$explain="") {
  $CI = &get_instance();
  $parms = array(
    'label' => $label,
    'name' => $name,
    'explain' => $explain
  );
  return
$CI->parser->parse('_fields/image_upload',
$parms, true);
}
```

And the controller code ends up like...

```
$this->data['simpler'] =
makeImageUploader($label,$name,'Any
file...');
```

And the view like...

```
{simpler}
```

A CODEIGNITER PACKAGE

To make one of the helpers or libraries into a package, set it up in its own folder, inside application/third_party, and autoload its config file.

Caboose as a Package

```
/caboose
  /config/caboose.php
  /libraries/caboose.php
  /views/_components
    Confirm.php, date.php, ...
/assets
  /css/
    bootstrap.css,datepicker.css,jquery.css,...
    style.css
  /images/glyph...
  /js/
    bootstrap.js,jquery.js,redactor.js
```

```
$autoload['config'] =
array('third_party/caboose/config
/caboose',...);
```

Formfields Helper as a Package

```
/formfields
  /config/formfields.php
  /helpers/formfields_helper.php
  /views/_fields/
    checkbox, date, textfield, ...
```

```
$autoload['config'] =
array('third_party/formfields/config
/formfields',...);
```

PACKAGE MANAGEMENT

Composer is the de facto standard for managing package dependencies.

CodeIgniter supports Composer and its autoloading. This is new for version 3, and I am not up to speed on it yet.

Things are further complicated, because Composer requires PHP 5.3.2 or better, and its integration into CI is a bit awkward as a result.

PLUGIN MANAGEMENT

A Netbeans Plugin is under development, to make it easier to manage the packages in a webapp, including those that come from external sources.

I am the mentor for that project, and trying to get up to speed as quickly as possible, on Composer and related goodies.

Gimme More!

Check the User Guide ...

- Libraries
- Helpers
- Packages

Congratulations!

You have completed lesson #lesson06: Helpers and Addons

If you would take a minute to provide some feedback, we would appreciate it!

The next activity in sequence is: assignment2 Functional Webapp writeup 2015.03.07 23:59

You can use your browser's back button to return to the page you were on before starting this activity, or you can jump directly to the course homepage, organizer, or reference page.