PHIL STURGEON

CATAPULT INTO PYROCMS

efendi books

Phil Sturgeon

EFENDI BOOKS

Catapult into PyroCMS

Catapult into PyroCMS

First Published: 2013

eBook Version: 1.0.0

Published by **Efendi Books**

Table Of Contents

About The Author

Phil Sturgeon is the Founder and Project Manager of PyroCMS and has been coding for years using Ruby, Python and Node but always comes back to PHP. He is an active open-source developer, former CodeIgniter Reactor Engineer and ex-FuelPHP developer, web-tech consultant, avid writer, terrible mountain biker and excellent kayaker.

…with contributions from

Adam Fairholm started working on PyroCMS after the module he developed for it, PyroStreams, was integrated into the core in version 2.1. He now works on making flexible and easy content and data structures available throughout PyroCMS to developers and designers. In his off time Adam is building a rocket ship out of old vending machine parts.

An Introduction To Catapult into PyroCMS

PyroCMS is an awesome open-source CMS, used by individuals, small businesses and large organisations across the globe. Its flexibility, ease of use, fantastic code base and ever-growing community are touted as just a few of the reasons why tens of thousands of people, worldwide, swear by it and use it each day to build the next generation of web applications.

In this book - from PyroCMS creator Phil Sturgeon - you'll learn the core concepts underpinning Pyro, create malleable templates and themes, utilise Streams for powering content-heavy sites and examine the basics of writing addons for the Pyro platform. You'll learn the theory behind the CMS, how to mould the system around your content and how to use the powerful, designer-friendly template layer.

Who Should Read This Book?

Catapult into PyroCMS is intended for designers and developers with little experience using PyroCMS. You should be comfortable with the basic ideas of content management systems, how to install PHP scripts and how to set up a MySQL database. You should also understand the basics of writing HTML, CSS and simple PHP.

We won’t be doing anything too advanced, so don’t worry if you don’t feel too comfortable with PHP or CodeIgniter. This book is written for people who want to get up to speed with *using* PyroCMS, not developing advance addons for it.

This book was written for PyroCMS 2.2.0. Any code examples here are tested to work with this PyroCMS version, not any prior or future versions. With the rapid pace of development, I can make no guarantee that code will work on future versions. If you’d like to follow along with the code samples in this book, please make sure you use this version.

Chapter One

Core Concepts

Themes

Themes are the way that Pyro collects and organises front-end code. They're the outer container for the dynamic pages; the shell of your website. Your theme organises all the pieces of your site (CSS, JS, images, layouts, partials, and more) into a single theme folder; a coherent package that Pyro can understand.

Open up the **/addons** folder at the root of your Pyro install and you'll see a folder called **default**. This is the site reference; a per-site name Pyro uses to organise multiple websites with the multi-site manager. Inside your site folder (**default**) you'll see a **themes** folder.

Yep, you guessed it, this is the home of your PyroCMS themes. Themes can also be found in **addons/shared_addons/themes**–anything inside **shared_addons** will be available to every site in the multi-site manager. Inside you'll see a few folders - **base**, **minimal** and **conjunction** - that contain some sample themes that are shipped with Pyro. It can be helpful to dig through these for a quick reference when building your own theme, so keep them around for the time being.

I'd also like to draw the **base** theme specifically to your attention. This theme contains the basic structure of an HTML5 website. It can be great to use when you're starting out as it contains the essentials all set up and ready to go. In this book, we'll build a theme from scratch so we can learn what goes on under the hood, but in the future, when creating a new Pyro theme, **base** can be a great starting point.

Themes are set up and installed in Pyro by logging in to the admin panel and heading to *Design -> Themes*. You'll see our previously observed three themes - **base**, **minimal** and **conjunction** - as well as a fourth, active theme. This is Pyro's default theme, and its files are found in **system/cms/themes/default**;

Pyro uses the same templating system under the hood as the system we're about to get to grips with.

You'll see an *Options* button next to the *PyroCMS Theme* name. Click on it. You'll be redirected to a theme settings page, where you can configure specific bits about your theme, such as a 'Show Breadcrumbs?' boolean and a layout dropdown.

What is this useful for?

It's ideal when you're writing your own base theme and you'd like to re-use your code across multiple websites; you can have a stylesheet dropdown, different types of layouts available and a series of components you can switch on and off. You can even use the settings page to create customisable themes that can be sold or given away for other developers online.

Themes contain two major divisions of presentation files: layouts and partials.

Layouts

Layouts are the wrappers for your pages. Your layout is the outer container for specific view files; templates for your pages or blog are embedded inside the layout. Layouts allow you to contain the HTML structure, common page elements like sidebars, headers and footers, and, importantly, embedding partials. They're usually used to wrap the page content with a header, footer and a sidebar, but could be anything you like.

You can create as many layouts as you like, and this can be *really* useful when you want to have specially branded marketing pages, for instance, or a mobile-only version of your site.

Partials

Partials are the individual parts of your layout. They're re-usable snippets of view code that you can use across different layouts. You can also use them to simply clean things up: using multiple, smaller files means your templates are easier to scan through quickly and make small adjustments.

Usually, you'll find that you'll want to separate your header and footer away from your layouts, as well as sections like sidebars. Partials can also be useful for sections of metadata or the area in your `<head>` tag where you link to your assets (CSS and JavaScript files).

A simple rule for a partial is this: if it contains HTML that is either re-used, or getting too unwieldy to handle in one file, it probably should go in a partial.

Pages

Pyro's page system is the mechanism for entering, displaying and managing both static and dynamic content. Static pages allow your clients to create new content and modify existing content in a predictable and conventional editing system. Page types allow pages to be tied to streams–giving you an utterly customisable editing form powered by a robust set of template tags.

Pages are even more useful in 2.2, because pages can now be linked to your streams. Let's say you have a site of products, where each product has its own dedicated page. You could create a project display page in the page tree, and then display that product using streams tags in the page layout, pulling the data from a separate stream. You still may want to do that, but it's the messier option; it's now possible to represent stream entries *directly in the page tree*.

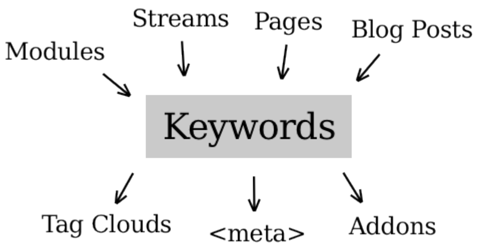
You tie your streams to a page type, so when you create a page, you fill out the data for a stream entry as well as the page fields. Then, you can simply use the fields from the page/entry in your layout. No pulling data from the URL or messing around with stream tags.

Plus, your data is still a stream, so you can display it in a cycle loop elsewhere, create entry forms for it, do anything you can do normally with streams. You get the powerful content management of streams - which we'll find out more about soon - baked straight into the page system.

We can customise the way that pages are outputted through the page type's layout. A page's layout is the way of customising template at an individual level. They allow you to structure a specific page in the master layout in a specific way–overriding any default layout that may be set up.

Keywords

PyroCMS also contains a centralised list of keywords that enable the categorisation and labelling of content. Keywords are supported across the entire system, from generating tag clouds on the blog through to setting `<meta>` keywords for static content.



A keywords field type exists that allows you to add keywords to any stream entries. Keywords are perfect for adding tags to posts, categories to shopping cart products or generating a tag cloud based on custom data entered by your clients. You can access the central list by heading to *Data -> Keywords* in the control panel.

Streams

The idea of streams may be familiar - they're known as channels or sections in other systems - and they're one of the most powerful elements inside PyroCMS. A stream is essentially a series of entries; multiple items of content. This differs from a blog insofar as streams are totally customisable. You can cherry-pick different fields of different types to make sure Pyro suits your website's content, not the other way round.

An example: your client runs a series of events and they'd like to display them on their website. They've got a name and a description, but they've also got an attached date, a number of available tickets and a venue. This custom data is too complex to put into the blog module, so instead, we can create a stream and set up exactly the right fields for this type of content. This is a typical example of a need to display custom content in a dynamic way, and this is precisely what streams are used for.

Streams aren't only used by your website: they allow you to add custom fields to page types, user profiles, and the blog. They're also available to developers via the Streams API, so third-party addons you use with PyroCMS may have the ability to add custom fields.

Streams are customized by assigning fields to them, and each fields has a different field type. There are a bunch of core field types to choose from: WYSIWYG editors, date pickers, numbers, entry relationships and keywords. There are also a number of other addon field types available.

Once you've added some fields to a stream, you can then add and modify entries within that stream, much like you do in other content entry areas in Pyro. Like we mentioned a moment ago, pages can be tied to streams through page types.

Here's another good rule: if you have custom content directly related to a page, use a stream and a page type. If you've got custom content not directly linked to a page and its URL, use a stream and a template.

When we get to writing some code - soon, I promise! - we'll take a look at how to use streams in your templates.

Addons

Addons are the way that custom developer code integrates into the core of PyroCMS. They lie at the centre of Pyro and allow you, the developer, to modify the CMS and extend it to do whatever you wish. With addons you can extend Pyro's support for third-party services like Twitter and Facebook, create complex booking and invoicing systems, hook into events to effect the behaviour of internal modules and – well, pretty much anything you (or your clients) can dream of.

While writing addons is simple, there are a few different types of addon components to learn about. Even if you don't plan on writing much custom code, it's a good idea to acquaint yourself with the different types and get a feel for how the addon system works.

Plugins

Plugins are used in your templates to embed simple content. They can represent files, assets, streams, entries, whatever you like. They use a simplistic tag syntax much like Smarty or Mustache, with the simplicity of single tags and the ability to display more complex, involved data with tag pairs and conditionals.

Plugins can be used on their own or complimenting a larger module. They're just a basic PHP class - Pyro's templating system takes all the tag parsing work out of your hands - so are very easily to build and extend.

Plugins sit in a **plugin.php** file at the root of the addon. They're comprised of a class called `Plugin_Something` (where 'Something' is replaced with the name of your addon) and they extend from `Plugin`. Like this:

```
class Plugin_Math extends Plugin
{
    public function random_number()
    {
        return round(rand(0,100));
    }
}
```

The `random_number()` method corresponds to the function called in the template tag, and whatever's returned from that function is displayed in the template:

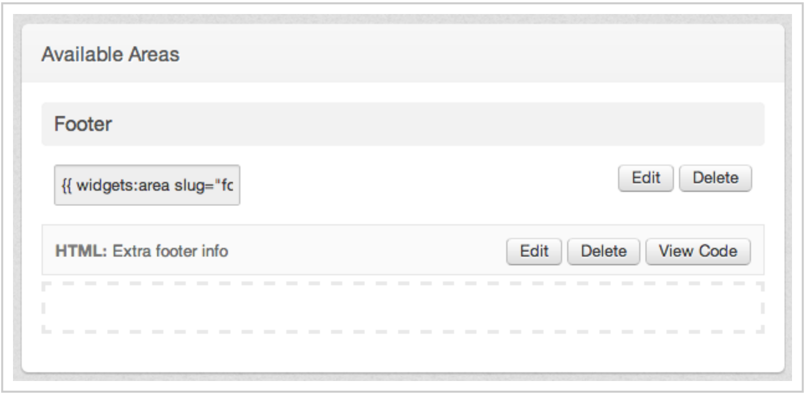
```
<p>Your random number is: {{ test:random_number }}</p>
```

Plugins can do much more than this, of course, but we'll look more at the way they work later when we write our first proper addon.

Widgets

I like to view widgets as “intelligent partials”. They accept a few options and give you back some pre-defined HTML output. The reason why they're so clever is that you can define widget areas for your widgets to sit in. Your client can then easily manage intelligent, dynamic content without having to call you for help.

Let's illustrate this with an example: the “Twitter Feed” widget built into the core of Pyro is configured with a Twitter username, is placed in a widget area, then displays a list of tweets in your template:



Widget areas aren't necessary - you can call a widget directly with a template tag - but they're very useful, because they give your client the ability to move things around to their specifications with a clean, simple, drag-and-drop admin interface.

You can write your own widgets, or use one of the several built in - including RSS feeds, Twitter, generic HTML and social bookmarking.

At first Widgets seem a little confusing, but once you get the hang of them you will come to the only logical conclusion: they are epic.

Modules

Modules are the most powerful, and most complex type of addon. Modules support a frontend interface as well as a full backend admin interface, which makes them perfect for more involved systems like shopping carts and download managers. The admin panels you interact with most of the time - blog and pages, for instance - will be modules.

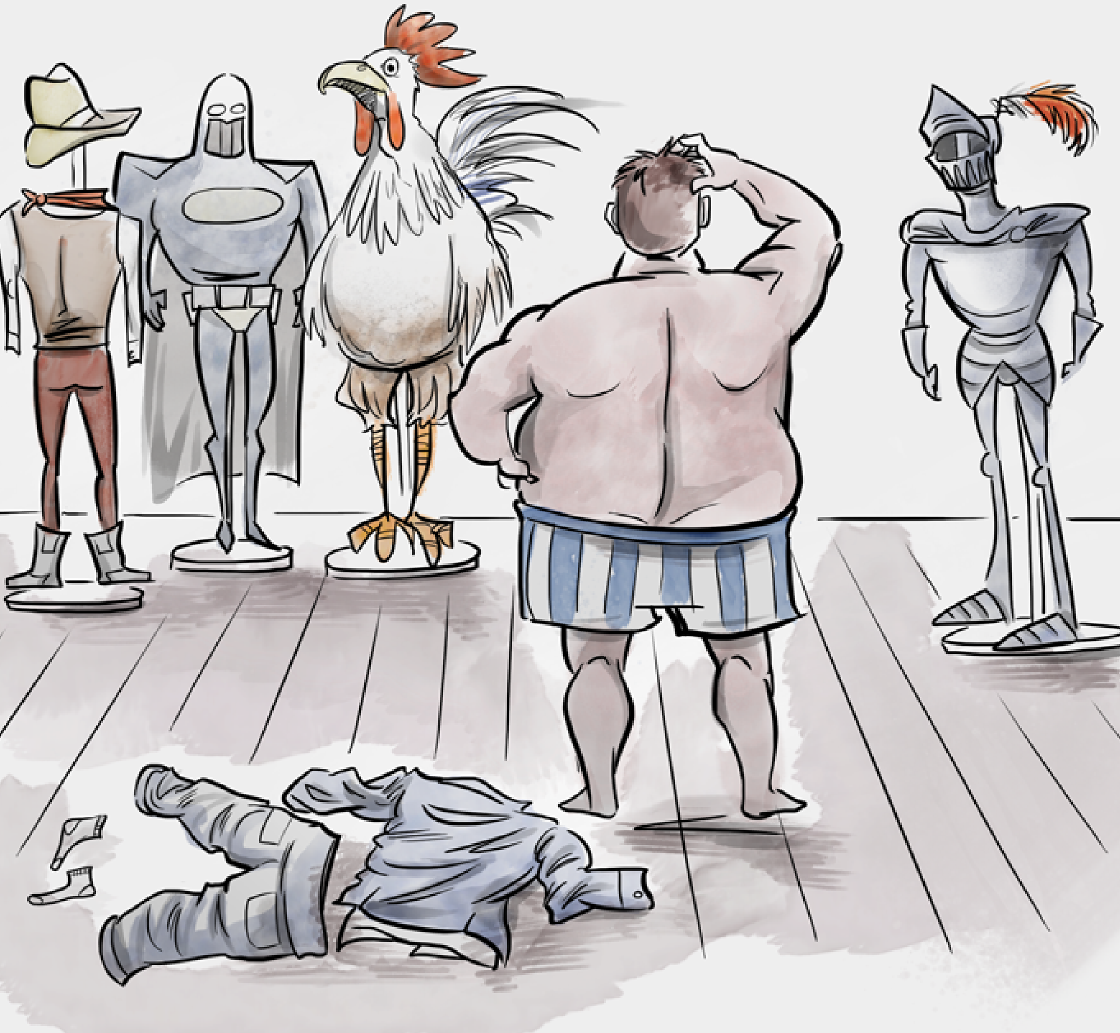
Modules are built like mini-CodeIgniter applications and have their own controllers, models and views, which means that anybody with CodeIgniter experience will be able to dive straight into developing them. A big part of the MVC (Model View Controller) separation is that your module code is very organised and clean.

Modules aren't just comprised of the MVC stack, though. They can have their own libraries and config files too… as well as plugins and widgets. Plugins work with modules to provide a simple template interface - although modules have their own interface too - and using widgets lets your module piggyback on the Pyro drag-and-drop widget system for your clients to display and rearrange dynamic content in their templates.

The core operates through modules (check out **system/cms/modules** to see more) and any custom modules are placed in your addon folder (**addons/default/modules** in our case). You'll need to create a couple of module files in order to get a module working properly, but we'll come onto that later.

Chapter Two

Themes



Getting Started

We've already looked at the basic concepts of themes, layouts and partials… but how do these actually translate into building a Pyro theme?

Create a new folder inside **addons/default/themes** called **catapult**. Themes usually comprise of layouts, partials, CSS files, images and JavaScript, but really all you need is a layout file and a theme information file; the rest is optional.

Create a **theme.php** file in the root of the catapult folder which needs to contain a basic class of metadata:

```
class Theme_Catapult extends Theme
{
    public $name = 'All About PyroCMS';
    public $author = 'Phil Sturgeon';
    public $author_website = 'http://philsturgeon.co.uk';
    public $website = 'https://efendibooks.com/books/
catapult-into-pyrocms';
    public $description = "The Catapult Into PyroCMS mini-site
theme";
    public $version = '1.0';
}
```

We define some very basic information about the theme here: its name, author and a description of the theme, along with links to websites and a version number. None of this is that important, it's just what shows up when we view the theme in the admin panel.

Which reminds me… jump into the Pyro admin and click on *Design -> Themes*. On that list, you should see a *Catapult into PyroCMS* theme with the description we wrote. If you try to enable the theme - select the radio button and click save. At this point it will error, because we've not set up a default layout for our theme.

To set a default layout simply create a **views/layouts** folder, and create a new file called **default.html** inside that.

We'll build a very simple layout to begin with; a basic HTML structure and nothing more:

```
<!doctype html>
<html>
<head>
    <title>All About PyroCMS</title>
</head>
<body>
    <header>
        <h1>All About PyroCMS</h1>
    </header>

    <div id="outer-content">
        <div id="content">
            Content goes here
        </div>
    </div>

    <footer>
        <p>Copyright &copy;2013 Efendi Books. This is a sample
website from the Phil Sturgeon book, <a
href="https://efendibooks.com/">Catapult into PyroCMS</a></p>
    </footer>
```

```
</body>
</html>
```

BUT, how are we going to show our pages? Right now this is a static file, so how do we get Pyro to display our content?

Pyro gives us access to a `{{ template }}` variable that contains the essential content for displaying the current page. We'll use `template:body` to access the body inside the content `<div>`:

```
<div id="outer-content">
    <div id="content">
        {{ template:body }}
    </div>
</div>
```

The template body is a single string containing all output from whichever module is currently running, be it all of the page, blog or anything else. In Ruby on Rails this would be your **yield** keyword.

Another with this basic layout is that we have the site name hard-coded - which isn't good - so let's swap that out for a variable:

```
<title>{{ settings:site_name }} | {{ template:title }}</title>
```

...and the header:

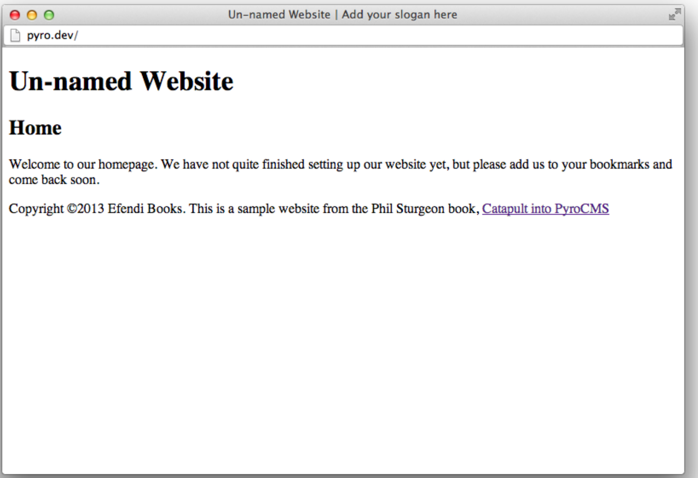
```
<h1><a href="{{ url:site }}">{{ settings:site_name }}</a></h1>
```

By using `{{ url:site }}` we can do crazy things - like move PyroCMS into a sub-folder - without breaking anything. Lots of other CMS's get rather confused if you try to do that, but using these tags you'll have no problems.

This template also has a hard-coded date, so let's change that too. We'll use the `{{ helper }}` tag for this. `helper:date` uses a tag parameter to pass through a date format. This is the same format as in PHP's native `date()` function:

```
<p>Copyright &copy;{{ helper:date format="Y" }}
```

Save the template, go back into the admin panel and enable the theme. Now load the front-end of the site. You should see something like this:


Un-named Website | Add your slogan here

pyro.dev/

Un-named Website

Home

Welcome to our homepage. We have not quite finished setting up our website yet, but please add us to your bookmarks and come back soon.

Copyright ©2013 Efendi Books. This is a sample website from the Phil Sturgeon book, Catapult into PyroCMS

Hey, it works!

A couple of things… our title is now back to “Un-named Website”. Let's change that. In the admin panel, head over to *Settings* and give our site a name. Refresh and it should update in the template. Set a slogan too–if you view source and take a look at the `<title>` tag you'll see “Add your slogan here”.

Let's break the layout up into partials. Extract the HTML up to the closing `</head>` and put it into a **views/partials/header.html**, replacing it with a call to `theme:partial`:

```
{{ theme:partial name="header" }}
```

Do the same with the footer and closing HTML tag:

```
{{ theme:partial name="footer" }}
```

…which gives us a nice, clean layout file:

```
{{ theme:partial name="header" }}

    <div id="outer-content">
        <div id="content">
            {{ template:body }}
        </div>
    </div>

{{ theme:partial name="footer" }}
```

If we refresh, we should get the same output. As you can see, a combination of layouts and partials is a great way of organising your template code and keeping the presentation layer of your website clean.

Layouts are a little bit cleverer than just a default file. You can create multiple layouts, and a layout named after a module will automatically be loaded when that module is used. Layouts can also be used for different page types.

Assets

Assets are the bane of many developers' lives. To begin with, they're easy to handle; basic `<link>` tags and `<script>` tags to include the files you need. Then, suddenly, you need to include separate stylesheets depending on the page. Then you need to include vendor stylesheets from other modules. There are other issues too: where should these files reside on the file system? How does caching work?

Suddenly, what was once simple becomes very complicated. Managing your images, stylesheets and javascript files elegantly is a tricky problem. Thankfully, PyroCMS contains a robust asset plugin to handle this with grace.

Your assets live in their own folders inside your theme:

- **themes/<theme-name>/css**
- **themes/<theme-name>/img**
- **themes/<theme-name>/js**

There is no special trick to how these files have to be made, they're just pure CSS, JavaScript and images copied straight from wherever your designer left them. They can then be accessed through the asset plugin in your templates.

To load these assets in your theme you can do it one of two ways: through the asset plugin, or by directly linking the asset.

Directly linking asset files is the simplest route. This involves using some basic template tags to output an HTML tag with a direct link to your file. Your web server then handles the output process.

It looks something like this:

```
{{ theme:css file="style.css" }}
{{ theme:js file="jquery.js" }}
```

Which will output HTML similar to this:

```
<link href="http://example.com/system/cms/themes/default/css/
style.css" rel="stylesheet" type="text/css" />
<script src="http://example.com/system/cms/themes/default/js/
jquery.js" type="text/javascript"></script>
```

Whilst this is simple, it's not particularly optimal. Many of the issues mentioned above still exist, and each file needs to be required and loaded separately.

If we want to be a little more clever about this we can let PyroCMS assemble, concatenate and minify our assets. We specify which assets we need in our theme and place a call to `asset:render`. The asset plugin will then bring in all the files it needs to, concatenate them together and minify them into one, optimised asset file.

This gives us many benefits. Our code is still clean and template friendly. We get a much smaller file and our page loads are much more lightweight because of that. PyroCMS is clever about the caching too - it will include cache-buster timestamps on the file name. It won't cache locally either - you'll be making changes regularly during development! - so your local workflow won't be affected.

So, let's open up our **partials/header.html** and add in the Twitter bootstrap stylesheet, which we'll use to help style our site:

```
{{ asset:css file="theme::bootstrap.css" }}
```

We'll also want to add in our own custom styles, so let's add a separate stylesheet:

```
{{ asset:css file="theme::style.css" }}
```

Now, we need to process these files and get the required HTML spat out into our theme. That's easy:

```
{{ asset:render }}
```

The CSS files we've specified don't exist yet, so if we try to load our site we'll get an uncaught exception. Creating them is easy. We need a **css/ bootstrap.css** file with the Twitter bootstrap source in it. You can get this from the Twitter Bootstrap website (version 2.2.2)[1] or from this book's GitHub repository[2].

We also need our own website's stylesheet. Like above, you can grab this from the GitHub repository linked to this book.

1. http://twitter.github.com/bootstrap/
2. https://github.com/efendibooks/catapult-into-pyrocms

Modular View Overloading

Modules - both core and third-party - can output their own templates. This saves you having to build out an entire interface for every new site you work on. These templates use generic HTML that you can style easily, using just CSS. There will be times when you need to override the output of a specific view, however.

Thankfully, that's easy. The folder structure of your custom theme can mirror that of the core PyroCMS system. This includes core modules.

Core modules exist inside **system/cms/modules**. If you examine the **blog** module, you'll see a **views** folder. These views can all be overridden inside your custom theme.

Create a **modules** folder in your theme's **views** folder. Then, all you have to do is create a **blog** folder - or any other module folder, for that matter - inside and you'll be able to override any views the module uses.

This is fantastic, because it means you can customise your application completely without having to hack around in the core, but without the expense of having to rewrite basic blog code every time.

Since we've got the beginnings of a good theme here, let's override some of the blog code and get a custom blog layout. Head over to `/blog` and you'll see a 'There are no posts at the moment' message.

Let's change that. Head into the *Content -> Blog* and add a new post. If we now load the blog page again, we'll see our post, but it won't look good.

Instead, let's override the blog code in our theme and customise the output. Create a new file inside our theme **views/modules/blog/posts.php**. Normally, you'll copy and paste the existing module code like I mentioned above, but we'll simplify the view substantially, so we'll write the blog code from scratch.

We'll open a container `<div>`, check for posts and begin to loop:

```
<div id="posts">
    {{ if posts }}
        {{ posts }}
            <div class="post">
```

Next we want to display the date. If you check out the stylesheet you'll see how we make this hang to the side of the post. We can use the `{{ helper:date }}` function to customise how we output the date:

```
<div class="date">
    <span class="day">{{ helper:date format="d"
timestamp=created_on }}</span>
    <span class="month">{{ helper:date format="M"
timestamp=created_on }}</span>
</div>
```

Next up, we want to display our title:

```
<h3><a href="{{ url }}">{{ title }}</a></h3>
```

This is the posts list page, so we'll need to display the preview and a read more link too:

```
<div class="preview">
    {{ preview }}
</div>

<p><a href="{{ url }}">{{ helper:lang
line="blog:read_more_label" }}</a></p>
```

We'll close off the posts loop and display out the pagination tag:

```
{{ /posts }}

{{ pagination }}
```

Finally, we'll display the “no posts” paragraph, just in case:

```
    {{ else }}
        {{ helper:lang line="blog:currently_no_posts" }}
    {{ endif }}
</div>
```

What are these `{{ helper:lang }}` tags about? All of PyroCMS's core modules are translated into fifteen languages; from Brazilian Portuguese to Finnish and Spanish to Hebrew. `{{ helper:lang }}` gives you access to the current module translations. The `line=""` parameter maps to the loaded language keys. Check out **system/cms/modules/blog/language/english** to see what these files look like.

Our implementation of the list page is rather simple. We'll do something very similar to the specific post view page. Create a new overloaded file in the theme: **views/modules/blog/view.php**. We'll begin it exactly like the list page:

```
<div id="posts">
    {{ post }}
        <div class="post">
            <div class="date">
                <span class="day">{{ helper:date format="d"
timestamp=created_on }}</span>
                <span class="month">{{ helper:date format="M"
timestamp=created_on }}</span>
            </div>
```

Under the title, we'll add a short notice about who wrote the post:

```
<h3><a href="{{ url }}">{{ title }}</a></h3>
<p class="author"><strong>- by {{ user:display_name
user_id=created_by }}</strong></p>
```

...and then display the body, rather than the preview:

```
<div class="body">
    {{ body }}
</div>
```

...finish off the tags:

```
</div>
    {{ /post }}
</div>
```

…and we're done!

PyroCMS's blog module supports a lot more than we've implemented. We've not looked at the comments system, we've not seen how keywords integrate with the blogging platform to give you tagged and organised posts. The blog module gives us categories and RSS feeds, and comes with several widgets too. The good news is, these more complex features aren't actually that complex; they're all designed to save you time, stress and sweat.

This is a “little bit” cleaner than the equivalent code in WordPress ey? Not a `while()` loop in site, nor any horrendous procedural functions littering our logic.

Don't forget you can overload more than the blog module too. You can customise the behaviour of most of the core modules. If it outputs to the front-end, you can customise it using the technique above. This includes commenting, site-wide search and user management.

Widget Areas

We've mentioned widgets already - small, reusable pieces of dynamic content that can be rearranged and reconfigured safely by your client - so how do we put them in our theme?

Widgets can be placed into your theme in a few ways; by far the most flexible way is through establishing widget areas. Generally, widget areas lie in places like a sidebar or a footer. By establishing widget areas in your theme, your end-user (your client) can mix and match all available widgets as they please.

The template tag for that is super-easy:

```
{{ widgets:area slug="sidebar" }}
```

Let's add an area in our layout. We don't have a sidebar, so let's put it in the footer partial instead:

```
<footer>
    {{ widgets:area slug="footer" }}
```

Next up, we need to create this widget area in the control panel. Head into the control panel (*Content -> Widgets*), tab onto *Areas* and delete the existing sidebar widget area. We then match the `slug=""` parameter in our template tag with our newly-created footer area's short name.

If you'd like to display a specific widget and *not* let the client/user/whoever change what it is, you can set up the widget in the CP like above, and then use the `instance` tag to display it anywhere in your template:

```
{{ widgets:instance id="3" }}
```

You'll be able to get the tag by hitting the *View Code* button next to the installed widget.

Any widgets we now assign to the widget area will be displayed on the website. It's that easy!

Routes and Routing

URLs in PyroCMS are structured like pretty much any other CodeIgniter application. It's based on a modular design, so we access each module with a base URL:

```
/<module-name>
```

By default, any module which expects to output to the front-end will be able to output with just its module name in the URL. To do this, you create a controller inside your module with a matching name. The `index()` method is then used as the 'base' method.

For example, a module named `booking` would have a controller named `booking.php` and an instance method called `index()`. We could access this method with this URL:

```
/booking
```

If we'd like to call the `process()` method on the `booking.php` controller of the `booking` module, we can call that directly too:

```
/booking/process
```

Your module might be more complex, and you might want to break the logic down into multiple controllers. That's easy:

```
/<module-name>/<controller-name>/<method-name>
```

Aside from this default routing behaviour, you can customise your module's routes with ease. Your custom routing rules go into `modules/\<module-name>/config/routes.php` and behave just like CodeIgniter's routing engine.

Here's an example from PyroCMS's core blog module:

```
$route['blog/rss/all.rss'] = 'rss/index';
```

This routes a request from `/blog/rss/all.rss` to the `rss.php` controller, and it calls the `index()` method.

Taking this a little further, you can use parameter matching (a regular expression) in the original route to pass through any custom parameters you may wish to. If you make a request to `/blog/rss/hotels.rss`, you'll be routed to the `rss.php` controller, the `category()` method will be called and `'hotels'` will be passed through as a parameter.

It they're CodeIgniter routes, so you can use a simple matcher:

```
$route['blog/rss/(:any).rss'] = 'rss/category/$1';
```

...or you could use a regex:

```
$route['blog/rss/([a-zA-Z0-9_\-]+).rss'] = 'rss/category/$1';
```

This parameter is then passed through as a parameter to the controller method:

```
public function category($slug)
{
    echo $slug;
}
```

Chapter Three

Managing Your Content

Managing Pages

Pages, like we mentioned earlier, are static bits of content, managed through the admin panel, in a predictable and comfortable editing environment. They're perfect for when you want no-nonsense, no-frills content editing in a hierarchical structure.

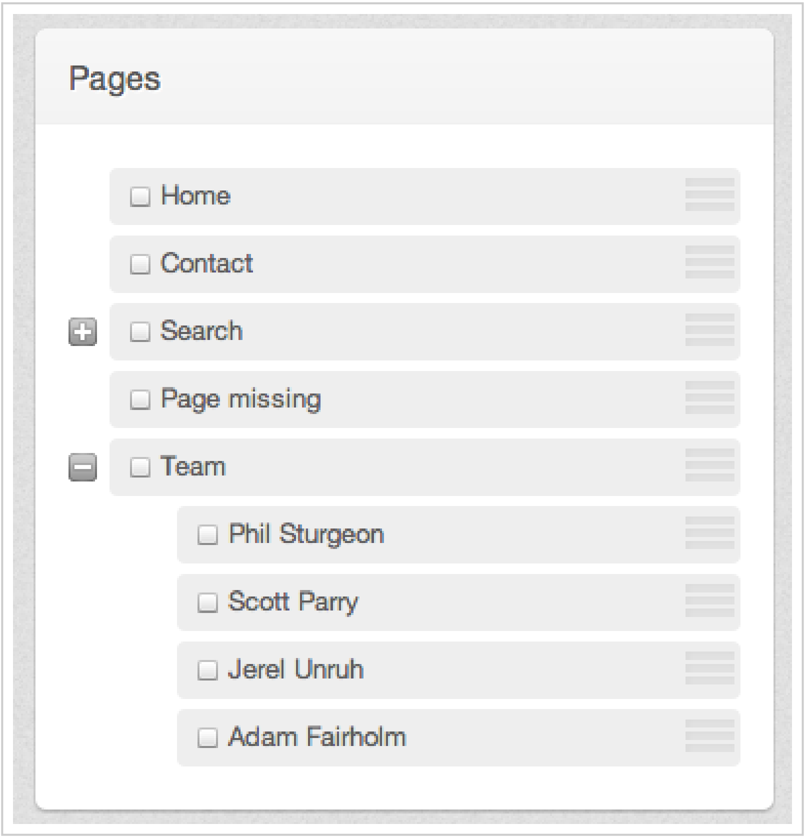
Head into the admin panel and click on *Content -> Pages*. You'll see a list of the pages in your install to the left. Click on one of pages. You'll see the page's details, including its slug.

The slug is the URL mapping; the way Pyro routes a request to that page. Creating a page called “About” with a slug of `about` will be available at the URL `http://example.com/about`.

Pages become really useful when used to build a hierarchy of content. You can create child pages which are nested under a parent page. This parent-child relationship is a great way to organise content into sub-groups.

An example of a child page from “About” would be something like “The Team”. Given a slug of `team`, the child page will nest its URL underneath the parent, so you'd access the team page at `http://example.com/about/team`.

Here's an example of what nested pages look like in the control panel:



Page Types

Not all pages in a site are created equal. They have different pieces of information and require different ways to structure and display that information. Page Types give you the ability to change the structure of the content and allow you to define things what fields and metadata the page uses.

PyroCMS comes with a simple default page type that contains a single WYSIWYG editor – the traditional page editing flow that we all know and love to hate. This is great for creating very simple pages with flat content.

Creating a Page Type

Let's say, however, that we are creating a website for a restaurant with three locations. Each location needs to be on a separate page, and the managers of the site want to change the location information on their own. Letting a site manager edit a WYSIWYG content area for this would be extremely limiting; clients are (usually) bad with content. How can you extract the specific bits of information from a free-form content box? How do you display this information precisely on the front-end?

This is precisely what page types are designed for. You'd create a new “Restaurant” page type. You'd “Create New Stream for this Page Type”, allowing you to use the underlying stream system to give your content flexibility. Creating the page type will place it right next to the default page type, where it will be selectable when creating a new page.

With a page type created, we can add some custom fields. These fields give our site managers the desired structure to follow when entering in data. It also means that, when we come to displaying our data, we have access to it in a predictable, raw form.

Our pretend restaurant page type might use the following:

- *Location Address*: A textarea field that will hold the location address.
- *Location Picture*: An image field that will hold an image of the location.
- *Phone Number*: A text field that will hold the contact number of the location.

We could also specify if any of these pieces of information are required or unique. To make things even easier on our site managers, we can also write some instructions that will show up on the page form.

If any of this is confusing, don't worry, we'll go through a real example in Chapter 4 as we build out the *All About PyroCMS* website.

On the Front-end

Once we have the information for our restaurant in a page, we can display it. We customise this using the layouts attached to our page type. If we want to display our location address field, we could display it with a simple `{{ location_address }}` template tag – just like building themes and overloading our blog content.

Metadata

While we are determining how we will display custom field data in our page type layout, we can take advantage of another feature: defining the structure of our metadata.

In our restaurant page example, we know that the title of each of those page should always be the title metadata tag. When we edit the page type, right next to the *Layout* tab is an extra tab called *Metadata*.

The cool thing is, we can use tags right inside these form fields. In our example, we'd like the `<meta>` description to be the address of the restaurant. That's easy: all we have to do is put `{{ location_address }}` into the description box and we're done.

On each page, these tags will be populated with data from the page. We can override these at a page-by-page level if we want to customise it individually, but they'll use the page content here by default. Sensible defaults make sensible content, and sensible content makes happy search engines and happy clients.

URL Matching

Another useful feature of Pyro's page system is strict URI matching. By default, the “Strict URI” checkbox under the *Options* tab on the edit page is checked.

This means that strict URI matching is enabled, and that the request in the address bar needs to match *exactly* what's in the slug (URL) field.

Having a page with a slug of `about` with Strict URI enabled would mean the address bar would need to be:

```
http://example.com/about
```

If you disable this however, extra segments can be added to the URL to change the content.

Let's say we create another child page, this time under “The Team”. Give it a slug of `profile`, and we can display each team member's info on an individual page:

```
http://example.com/about/team/profile/adam
http://example.com/about/team/profile/jerel
http://example.com/about/team/profile/joshua
```

Now, if your profile page only contained static HTML then this would be pretty useless, but if you use streams - which we'll take a look at in the next chapter - you can have a totally dynamic page using a familiar URL structure.

The use of loose URL matching is a really powerful technique in PyroCMS, and will be core to the way you build sites for the platform.

Blogs

Blogs share a similar interface to pages, but they have a few more article-specific features like categories and the ability to close comments after a certain amount of time. Navigate to *Content -> Blog* and you'll see a list of the blog articles in the system, and who they were written by. The blog module is a purpose-built module for blogging; if you'd like arbitrary dynamic content, check out Chapter 4 - Streams.

URLs

Blog articles have a different URL structure than pages. Like pages, they use a slug, but in order to better categorise them, and because they're usually date-sensitive content, the URL is prepended with the year and the month of creation, as well as the blog module name.

A blog URL typically looks like this:

```
http://example.com/blog/2013/01/blog-title-in-slug-form
```

This is not exactly an SEO specific improvement - the benefits are trivial - but it does help with analytic reporting systems that support content drilldown features like Google Analytics. Within your usual reporting tools you can then easily see which are the most popular articles by year, or even by month.

Blog articles can be left in draft mode, where you can get a preview URL. You can share this URL with anybody you like, who will then be able to view the post. This is handy if you want to garner feedback from an article, need to send it to somebody who will spell check things over, or want to just see what it looks like inside the theme, with images and CSS styles, without having to post it live.

Custom Fields

We discussed earlier how streams could have custom fields attached to them. Well, the blog module is just another stream, and as such any custom fields can be attached. Want to add a “Featured Image” for your homepage slider? Go for it. Fancy adding a “Thumbnail”? That's easy as clicking a few things. Want to add a PDF copy of the article as a download? Easy.

Multiple Blogs

Currently - version 2.2.0 - the blog module only supports one blog. PyroCMS 2.3.0, however, will support multiple blogs out of the box, so instead of there being a single blog, you'll be able to create a reviews blog and a news blog, or a whitepaper blog with its own set of custom fields.

All of this can be done directly with the streams manager, but that module only exists in Professional - or as an optional download - so this 2.3.0 feature will open up similar functionality to everybody. Additionally, the blog module is set up specifically for a blog post environment, so it's usually quicker.

Navigation

Creating navigation for your site can be done with static HTML, but this is a really inflexible way of doing things. Instead of updating your theme layouts every time you add a page slug you can use the PyroCMS core navigation module to power your navigation instead.

Navigation breaks your links down into groups. These groups can be rendered with a simple tag:

```
{{ navigation:links group="header" }}
```

...which will generate a bunch of `<li>` items, complete with `first`, `last` and `current` classes, with `<a>` tags linking to the pages specified in that group.

These links can be customised in the *Structure -> Navigation* section of the PyroCMS admin panel. By default you'll see three groups set up: Header, Sidebar and Footer. These are all customisable, of course, and you can add as many groups as you like.

Each group has a series of links. Each link can point toward an arbitrary URL, a site link (any slug relative to the site's root), a module or a page. This is all done in a simple GUI, and gives you fine-grained control over where you links are and what they point to.

We'll add in our navigation tag just under the title in our layout, with the HTML5 `<nav>` element. Inside our theme's header partial:

```
<nav>
    <ul>{{ navigation:links group="header" }}</ul>
</nav>
```

We'll revisit the navigation admin panel to customise our navigation links later on, when we've added some more content to the website.

Caching

PyroCMS takes caching really seriously; it's a simple technique that makes a *huge* impact on how your site performs and how often people will use it.

The vast majority of PyroCMS's caching is handled under the hood. Layout rendering, navigation elements, pages, variables and keywords are all cached cleverly in the background. If you update a navigation element, for instance, it will automatically expire the cache so that your change will appear next page load. This goes a long way to improving the performance of your site.

Much of our content won't be changing regularly so we can afford to cache it. This speeds up page loads and reduces the amount of work being done by the server. With most CMS sites the slowest part is talking to the database, so the fewer queries the better. Then, when we make a change to our pages - or any of their children, or blogs, navigation, etc the cache will expire and our updates will be displayed.

From time to time, particularly when using streams, you will need to customise the behaviour and manually implement caching yourself, but these occasions are rare and out of the scope of this book.

Remember that the cache can be manually emptied too - head over to *Data -> Maintenance* to check out how to clean the cache up. If you're really interested in how it all works, take a look at **system/cms/libraries/Pyrocache.php** and the cache files found in **system/cms/cache** to get a deeper understanding of how PyroCMS handles its extensive caching.

Disk vs Memory Caching

PyroCMS has historically always used file-based caching, which is very much the standard for portable PHP-based CMS, but with the rise in popularity of the faster in-memory key-value stores like Redis and Memcached it does make the file-based system look rather antiquated.

The main reason for sticking with file-based caching for so long is that it does the job with minimum fuss. Permission one folder to write some cache files too, or ask users to install extra software–it's an easy choice to make.

That said, Redis support is already baked in to 2.3, which means that those with the know-how can get a speed boost from their system, and even use PyroCMS on multiple (load balanced) servers and share the same cache. Redis support is one of the many more advanced features to expect in 2.3 and beyond.

Search Engine Optimisation

Out of the box, PyroCMS works *very* hard to make Google interested in your content.

All URLs are structured with segments - like `/legend/man-bear-pig` - which is precisely the format they like. Your pages and blogs can accept keywords, which are fed into `<meta>` tags, and you can always write your own meta descriptions for any page or blog. This is all stuff that the end-user/client/johnny-with-a-mouse will be doing, but the tightness of the integration into the core of the system makes it very easy to create highly optimised sites.

Sitemaps

When the Googlebot and its buddies scour your site, one of the first things they will look for is a `sitemap.xml` file. A sitemap is an XML file which informs the search engine crawler bots about what URLs are available and what it should be looking for.

At the time of writing, if you navigate to `http://pyrocms.com/sitemap.xml` you'll see this sitemap:

```
<sitemapindex xmlns="http://www.sitemaps.org/schemas/sitemap/
0.9">
    <sitemap>
        <loc>https://www.pyrocms.com/blog/sitemap/xml</loc>
    </sitemap>
    <sitemap>
        <loc>https://www.pyrocms.com/pages/sitemap/xml</loc>
    </sitemap>
    <sitemap>
        <loc>https://www.pyrocms.com/videos/sitemap/xml</loc>
    </sitemap>
</sitemapindex>
```

It's rather empty, but don't worry! This is called a 'Sitemap Index', which means that this main sitemap is only there to delegate to any other module in your PyroCMS application which happens to have a **sitemap.php** controller.

These sitemap controllers are left entirely up to themselves when it comes to building the XML content, and it can be done however they like. This is important, because all content will be rather different from module to module.

Take a look at the blog module's **sitemap.php** controller to see how it works, and how you could implement one for your own module. Not developing a custom module? Don't worry then! The core PyroCMS modules are all rather clever, and where necessary, will already automatically generate a sitemap for you.

Thanks to this feature, if search engines know about your sitemap (either they discover it, or you notify them about it) then your new articles and pages will be indexes incredibly soon after they are added–not just whenever Google happens to fall over the link after clicking around on your site madly.

To write one of these sitemap controllers, you can use the blog module as an example:

```
/**
 * @author  PyroCMS Dev Team
 * @package PyroCMS\Core\Modules\Blog\Controllers
 */
class Sitemap extends Public_Controller
{
    /**
     * XML
     */
    public function xml()
    {
        $this->load->model('blog_m');

        $doc = new SimpleXMLElement('<?xml version="1.0"
encoding="UTF-8"?><urlset xmlns="http://www.sitemaps.org/
schemas/sitemap/0.9" />');

        // Get all pages
        $articles = $this->blog_m->get_many_by(array('status',
'live'));

        // send em to XML!
        foreach ($articles as $article)
        {
            $node = $doc->addChild('url');

            $loc = site_url('blog/'.date('Y/m/',
$article->created_on).$article->slug);

            $node->addChild('loc', $loc);

            if ($article->updated_on)
            {
                $node->addChild('lastmod', date(DATE_W3C,
$article->updated_on));
            }
        }

        $this->output
```

```
                    ->set_content_type('application/xml')
                    ->set_output($doc->asXML());
        }
    }
```

The controller is a basic `Public_Controller` like any frontend controller you build, with one method: `xml()`.

Some of the rest of this code looks a little crazy, but this is down to the SimpleXML system in PHP being a little… odd. You need to create an XML document with the correct XML namespace listed as `http://www.sitemaps.org/schemas/sitemap/0.9`.

We then hit up the blog module for all of the live articles, loop through them and add a new `<url>` node to the main article, along with `<loc>` and `<lastmod>`. The `<lastmod>` part is important–it stands for *last modified*, and is what lets the search engine bot know you've changed something.

The sitemap 0.9 protocol accepts two other nodes to be added, which are entirely optional but can be helpful:

```
<changefreq>monthly</changefreq>
<priority>0.8</priority>
```

These would be added inside the `foreach()` loop like this:

```
$node->addChild('changefreq', 'monthly');
$node->addChild('priority', 0.8);
```

An example of where this is useful is if you have old products in your database that have expired and are only really there for historic purposes. If you turn the change frequency down to monthly on these products, but have your latest products indexed every day, the newer products will be checked regularly for changes to price, description, availability, etc. You could also have your most important products listed with a higher priority, and your cheaper stuff less-so - if that was something management were interested in. Who knows.

Chapter Four

Streams

Fields

So we've already talked about Streams a few times, but how do we actually use them?

If we jump into the admin panel and hit on *Content -> Streams*, we can add a new stream using the form. The important field here is the *Stream Slug*, which we use to refer to the stream in our templates.

Once we've created a stream we can begin to add fields to it. Remember that these fields are utterly customisable - that's the point! - and so can be moulded to fit around the content you want to put into them. Need a date field or a WYSIWYG input? Fine. Need to plot a location on a map? Easy. Want to relate entries to other entries in other streams? Piece of cake.

As I've said the real power of streams lies in the variety of field types you can configure. At the time of writing, PyroCMS supports these kinds of fields out-of-the-box:

- **Simple text based inputs**, like textboxes, email fields, URL-friendly slug fields and WYSIWYG editors
- **A range of predictable HTML inputs**, including selection dropdowns, date pickers and file uploads
- **PyroCMS data inputs** that allow you to link to other bits of data in the system. These include user selections, relationships with other streams/entries and keywords
- **Miscellaneous fields** like countries, states and encrypted text fields

Third-party addons can also define custom field types, so you can download a pre-made solution or even build your own. A few examples of addons currently available in the PyroCMS store:

- Color Picker
- Decimal
- Geocoder Field

- Timezones

We'll chat about building your own field type in the next chapter.

Now we're sold on streams - because they're the best thing ever - we'll start work on a small stream-powered page. We're going to create our team profile page and use streams to power the data entry.

Create a `team_members` stream. In the *Menu Path* field, you can set up a link right in the navigation menu within the admin panel. I'll set it to `Content / Team Members`. We're also going to need to spec out our stream's content so we can decide what fields we need. Here's what I reckon we'll use:

- **Name** - a simple text field with the member's name. We'll use the title for this.
- **Picture** - an image upload field we can use to show an image of the team member
- **Role** - a WYSIWYG text editor describing the role that the team member plays in our fictional company

It's a simple example, but I hope you can see how flexible streams can be and how much they facilitate the creation of complex data entry forms in a swift and malleable way.

After stream creation, our menu item will appear. Tab onto it (*Team Members*) and click on the *Files* sub-navigation item. We will create both the **Picture** and **Role** fields–we'll use page types and titles to give the team member a name (more on that in a bit). As you select each field's type you'll see some configuration fields pop up – these allow you to decide the individual characteristics of each field.

We'll limit the picture upload to JPEG and PNG files using the *Allowed Types* configuration box. We'll also need to create a new file folder using the very swish file manager. If you go to *Content -> Files* you can create a new folder with the interface. Then, when adding the picture field, you'll be able to select the folder.

There's a big difference between file fields and image fields. If you'd like to display the image in the site, like we do, it's crucial to use the image type. The **file** field type handles caching differently - it tries to force a file download, which prevents it from being cached - so in order to speed up the site *remember to use the image type!*

A quick note on naming: remember that the names you give fields will usually be what the client sees when they enter content. If you're giving them obscure, technical names, they'll probably confuse the client. Use plain language and descriptive field names to prevent confusion and save you time! Also, when assigning fields to streams as we'll do in a moment, you get the option to set some field instructions. Writing something useful here is highly recommended.

We'll also need to assign the fields to the *Team Members* stream. PyroCMS doesn't attach the fields directly to the stream so that you can re-use them across multiple streams. If you hit the *Manage* button next to the stream on the streams list page you'll be able to click on a *Stream Field Assignments* link. This page will allow you to assign any fields you like to the stream, as well as set per-stream field settings, like requirements and uniqueness.

Set a required status on both fields. We can now get to work on getting our stream to be displayed in our templates.

We're going to create a custom page type for our team members page. Thanks to the awesome new features in 2.2, we can tie our team members stream directly to the page and have it appear as in the hierarchical page tree. In fact, this is exactly the kind of scenario the new page types system was designed for.

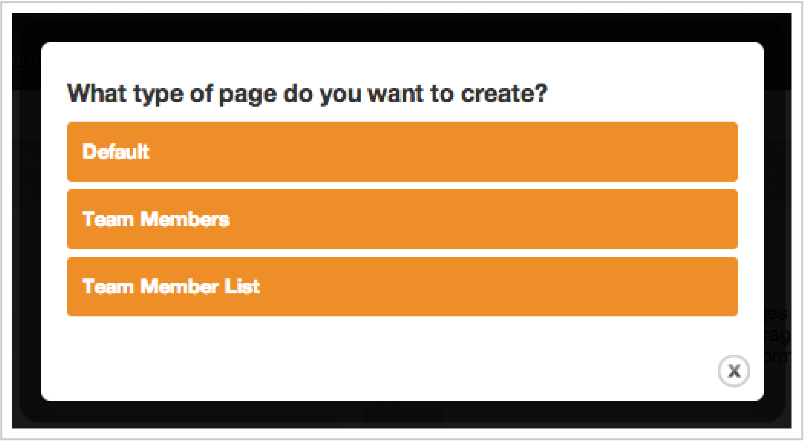
Head over to *Content -> Pages* and click on the *Page Types* sub-navigation item. We'll create a new page type called Team Members (predictably) and will select our team members stream as the assigned stream. You'll see in the streams dropdown that there's a pages stream–Pyro's page system uses streams under the hood itself!

I'll also use the *Title Label* field to specify that, as we configured earlier, the Team Member will be the title field. Save and exit the form.

Congratulations, you've just created your first page type!

We'll want to customise the team member list page too, so create a *Team Member List* page type too. Let it insert its data into the *Default* stream. We'll come back to this layout in a moment.

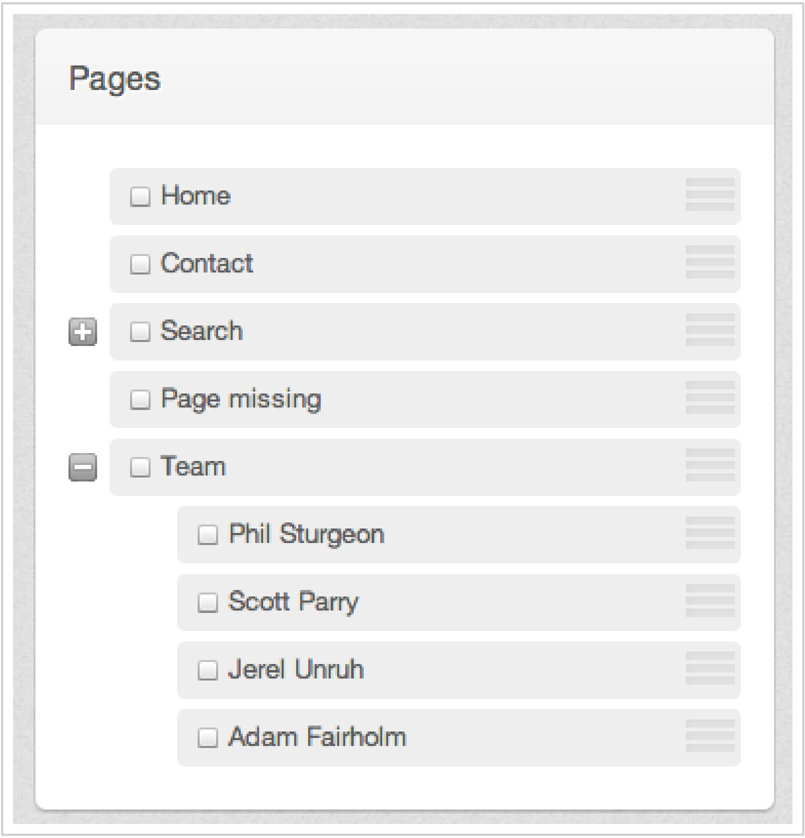
We'll create a `/team` page to be the parent page for each of our stream entries. When you click on *Add Page*, you'll get a snazzy modal window with the list of available page types:



We want our `/team` page to be our team member list page - it's the children of that page that will be our the Team Members type. Select default and create the page. We can edit the title and slug, as well as page content, metadata; anything else we can edit in the normal page editing system.

Now… here it gets interesting. Adding a new page under the newly created `/team` page - use the *Add Child* button - will include the stream's data. Instead of the traditional title and content boxes, you'll see the *Team Member*, *Picture* and *Role* inputs that we set up earlier.

Let's go through and add our team members:



Next up, we want to customise how these team members get displayed. Head back into the *Team Members* page type edit window and we'll adjust the *Layout*:

```
<div id="team-member">
    <h2>{{ title }}</h2>

    <div id="role">
        <img src="{{ picture:image }}" id="picture">
        {{ role }}
    </div>
</div>
```

We're accessing our custom stream data here - we use the `{{ picture:image }}` variable to get the full path to the image file and display it. We also use the `{{ role }}` variable to get our WYSIWYG content.

Currently, we can only access the page if we know the URL. Let's change that. Head back into the *Structure* -> *Navigation* section of the admin panel. Delete the contact link, we won't be using that. Hit *Add Link* in the header group. Add the Team link, connecting to the Team page.

If we refresh our homepage, the *Team* link should appear on our navigation bar. Click on that and you'll see the navigation will be selected and the page will load.

The CSS file I linked to earlier has made all this pretty–if you're interested take a look through there to see what it's doing. Since this isn't a book about design, I won't talk about the designs here.

…however, let's make a small change to the default page template. Since we've got the navigation bar showing what page we're on, the larger `<h2>` tag is unnecessary. Let's remove that from the layout.

We now want to display the children of the team page - all the entries in the *Team Members* stream - on the team page. We can do that by editing the page layout for our *Team Members List* page.

Let's use the `streams` plugin to output the contents of the stream in the layout:

```
{{ streams:cycle stream="team_members" }}
    {{ entries }}
        <img src="{{ picture:image }}" id="picture">
    {{ /entries }}
{{ /streams:cycle }}
```

This is great, BUT there's one problem. We can't access any data about the page (including its URL), so we can't link to the page. Not ideal.

Instead, let's go through the `pages` plugin and access the current page's children. We can then reach into the custom field data and grab the image, while linking to the page.

Here's how it works. We use `{{ pages:children }}` to get the children nested under the `/team` branch:

```
{{ pages:children id=page:id }}
```

We'll open up an anchor tag and set the link to the value of the child page's URL:

```
<a href="{{ url }}">
```

We'll then use the new `{{ custom_fields }}` tag to get access to the picture data:

```
{{ custom_fields }}<img src="{{ picture:image }}" />{{
/custom_fields }}</a>
```

...and close off the loop:

```
{{ /pages:children }}
```

You can now load the team page and see the team members. If we click on one of their pretty faces, we'll find ourselves on a member-specific profile page with their description. All this information can be edited easily through the admin system.

Next up, we need a more natural way of displaying the team/member hierarchy. We need the user to feel like they're on a sub-page, so let's add a link to the Team Members page layout, under the `<h2>`, to send the user back to the team page:

```
<p id="back-to-team"><small><a href="{{ url:site uri="team"
}}">&lt; back to team</a></small></p>
```

This gives us a back link to return to the team page, and means our site is really starting to take shape.

API

This next section is for the developers; it's a quick whirlwind tour through the Streams API and the underlying code. You'll need to be comfortable with PHP and CodeIgniter to get anything out of this bit!

The core logic of Streams (the Streams API) is built into PyroCMS Community in the **streams_core** module. If you're running PyroCMS Professional then you'll have access to a **streams** module too, which provides an interface to allow admins to build completely custom data structures.

Because **streams_core** is built in to PyroCMS Community, custom modules can use the powerful streams mechanism to hold their own content, allow custom fields on their entries and even build out most of the form logic for them on both the frontend and the admin panel.

This is all optional of course and developers can build modules however they like, but using these tools drastically reduces the amount of time it takes to build CRUD actions.

We can load the Streams API using CodeIgniter's driver loading mechanism:

```
$this->load->driver('streams');
```

It's designed as a driver: the streams system contains a fair chunk of code, and drivers are a great way of divvying up lengthy code into sub-classes.

When working with streams inside a custom module, it's usually a good idea to define a “Stream Namespace”. This could be a variable inside your module's model, for instance, and will usually be the name of your module. Since streams are used so extensively inside the core system, naming conflicts can occur. To counter this, most streams methods can use the namespace to put the request in the context of your module.

Creating and fetching streams themselves is easy. We'd like to use streams to store our custom module's data, so we'll begin by creating a new stream in the `install()` method of our module. First, our namespace:

```
public $namespace = 'shop';
```

Our hypothetical module will be a shopping cart system, so we'll need a stream to manage the products:

```
$this->streams->streams->add_stream('Products', 'products',
$this->namespace, $this->namespace.'_');
```

Here, we're adding a stream with the slug `'products'` and the full name `'Products'`, with the namespace and a DB table prefix.

We can fetch a stream simply with:

```
$stream = $this->streams->streams->get_stream('products',
$this->namespace);
```

If we want to get all the streams in the namespace, use the plural:

```
$streams =
$this->streams->streams->get_streams($this->namespace);
```

Updating and deleting is similarly easy:

```
$this->streams->streams->update_stream('products',
$this->namespace, array( 'stream_name' => 'Shop Products' ));
$this->streams->streams->delete_stream('products',
$this->namespace);
```

Streams need fields to be useful. Adding and deleting these through the API is simple, using predictably named methods:

```
$this->streams->fields->add_field(array(
    'name'          => 'Product Name',
    'slug'          => 'product_name',
    'namespace'     => $this->namespace,
    'type'          => 'text',
    'extra'         => array('max_length' => 200)
));
$this->streams->fields->delete_field('product_name',
$this->namespace);
```

One of the more powerful aspects of PyroStream's design is that fields are detached from streams. In order to link the two, we can use the assignment functions:

```
$this->streams->fields->assign_field($this->namespace,
'products', 'product_name', array('required' => true));
```

The last parameter is an array of options. These options correlate to the *New Field Assignment* form in the admin panel: required, unique, title_column and instructions. To save time, you can create and assign a field in one fell swoop:

```
$this->streams->fields->add_field(array(
    'name'          => 'Price',
    'slug'          => 'price',
    'namespace'     => $this->namespace,
    'type'          => 'integer',
    'assign'        => 'products',
    'required'      => true,
    'title_column'  => false,
    'unique'        => false
));
```

The `type` column refers to the field type. This could be one of the core field types (a comprehensive list with all available options available on the PyroCMS docs[3]) or a custom field type you develop yourself.

The *real* power of the Streams API becomes visible when we look at the control panel driver. The CP driver handles a big chunk of the repetitive CRUD stuff so you can focus on writing the important business logic of your addon and not on dull, monotonous control panel forms.

---

3. http://docs.pyrocms.com/2.1/manual/developers/tools/streams-api/core-field-types

The first useful method is `entries_table`. This generates a full HTML table, complete with pagination. It's also easily customisable, and comes with in-built support for drag-and-drop sorting. Cool, right?

We can use a few simple parameters to configure how the table is displayed:

```
$table_config = array(
    'title' => 'Products List',
    'sorting' => true
);
```

We can customise which columns are displayed by using the `columns` table config value, and filling it with the assigned field slugs:

```
$table_config['columns'] = array( 'product_name', 'price',
'in_stock' );
```

We can also add buttons to custom areas of our own modules. This makes building a more complicated custom content addon a breeze:

```
$table_config['buttons'] = array(
    array(
        'label' => 'Assign Stock',
        'url'   => 'admin/shop/assign_stock/-entry_id-'
    ),
    array(
        'label' => lang('global:edit'),
        'url' => 'admin/shop/edit/-entry_id-'
    ),
    array(
        'label' => lang('global:edit'),
        'url' => 'admin/shop/delete/-entry_id-',
        'confirm' => true
    ),
);
```

We're using the `-entry_id-` reference in the URL. PyroCMS will automatically populate your URL with the specific entry's ID for you when it generates the table. We also use the `'confirm' => true` key/value to trigger a JavaScript prompt for the user to confirm the deletion of our product.

Generating the table is then simple:

```
$table_html = $this->streams->cp->entries_table(
    'products',
    $this->namespace,
    15,
    'admin/shop/index',
    false,
    $table_config
);
```

We can enable pagination too here, by using the third and fourth parameters of `entries_table`. The third is the number of entries in the stream to paginate by, and the fourth is the URL to submit the pagination to. This will usually be a list page of some kind, and most commonly the page you serve the table up from.

The fourth parameter in the `entries_table` list is used to automatically output the HTML instead of returning it. If you set the fourth parameter to `true`, PyroCMS will spit out the generated table without you needing to create a new view. This is great when you're prototyping, or don't need any custom control over this page's template.

The second useful CP API function generates an HTML form to create an entry, including all of your custom assigned fields. This can literally save *hours* of work, so if you're using a stream to power your addon - if you're not you probably should be! - it's well worth reading the following bit and having a play with this function.

The `entry_form` function generates the entire form and provides an entry insertion action, which handles all of the inserting, validation and notifications. Like before, you use some basic configuration values to customise it:

```
$form_config = array(
    'return'             => 'admin/shop/index',
    'success_message'    => 'Successfully created new product',
    'title'              => 'Add Product'
);
```

The `return` value is the URI to the page you wish the user to return to *after successfully* creating the entry, and the `success_message` value is the flash message to display when that redirect takes place. If there's a problem with the validation, the system will automatically handle the validation messages for you.

We can then call the `entry_form` method to generate the form:

```
$form_html = $this->streams->cp->entry_form(
    'products',
    $this->namespace,
    'new',
    null,
    false,
    $form_config
);
```

Like before, we specify the stream slug, the namespace, the 'return-or-echo' value and the form config. You'll also notice `'new'` and `null` values. These allow you to set up edit forms as well as new entry forms:

```
$entry_id = $this->uri->segment(4);

$form_html = $this->streams->cp->entry_form(
    'products',
    $this->namespace,
    'edit',
    $entry_id,
    false,
    $form_config
);
```

The CP driver doesn't stop there; it can also generate full field and assignments tables, as well as forms for managing field creation and assignment for your custom addon stream. Check out the documentation on the driver for more about this[4].

4.http://docs.pyrocms.com/2.2/manual/developers/tools/streams-api/cp-driver

Locked Fields

Your module may depend on certain fields to exist in order for it to behave appropriately. You may want your users to be able to customise extra data, but prevent them from editing or deleting specific required fields. Thankfully, all fields support a locked boolean parameter to ensure that the fields you need stay where they're supposed to be.

When you add a new field, adding a `'locked' => true` parameter will lock the field into existence. The only way to remove it now is by deleting it through the API or manually in the database:

```
$this->streams->fields->add_field(array(
    'name'          => 'Price',
    'slug'          => 'price',
    'namespace'     => $this->namespace,
    'type'          => 'integer',
    'assign'        => 'products',
    'required'      => true,
    'title_column'  => false,
    'unique'        => false
    'locked'        => true
));
```

The CP methods, in particular, will pay attention to field locking. Check out the comprehensive documentation about field locking for more information[5].

5. http://docs.pyrocms.com/2.2/manual/developers/tools/streams-api/locked-fields

Chapter Five

Addons

Writing addons gives you almost limitless power. Using an addon gives you the opportunity to connect with third-party services (Twitter, Facebook, Flickr, whatever), develop a complex custom system with unlimited custom business logic and adjust the behaviour of how other aspects of PyroCMS behave.

There are several different components of addons with their own quirks, use cases and behaviours. Check out Chapter One of this book to find out more about them.

Most custom addons will take the shape of a module. Writing a custom module is a topic that deserves a whole book of its own, so in this chapter, we'll only be building both a small module and a small plugin just to demonstrate the basics of writing addons for the PyroCMS platform. And who knows, maybe there'll be a follow-up book for PyroCMS developers?

Modules

Modules sit at the very heart of the PyroCMS infrastructure. Pretty much every part of the system is built within a module; modules are the building blocks of the entire PyroCMS application. They are small MVC (Model-View-Controller) applications with their own routes, which makes them scalable solutions for installing and distributing custom code.

Modules are essentially just CodeIgniter applications, but they do require a custom file to tell Pyro that they exist and that they can be used by the system. This special file is called **details.php**, and it contains the root class for your module:

```
class Module_Test extends Module
{
    public $version = '1.0.0';
}
```

We can specify information about the module, including its name, description and some module settings, using the `info()` method. This method returns an array with all the module information:

```
public function info()
{
    return array(
        'name' => array(
            'en' => 'Test Module',
        ),
        'description' => array(
            'en' => 'This is a test module to do stuff.',
        ),
```

With this array we can also customise certain aspects of how PyroCMS handles the module once it has been installed. The following two booleans are used to enable front-end (user accessed) controllers and back-end (admin panel) controllers respectively:

```
'frontend' => true,
'backend' => true,
```

By default, PyroCMS will automatically run XSS (Cross-site scripting) checks against input values going into your module. This is great for security, but not ideal for certain occasions when you want to enable custom HTML or JavaScript, like in a blog post or page. You can turn it off easily:

```
'skip_xss' => true,
```

You can place the module in a menu item within the admin panel. In the following example, our test module will appear under the *Content* dropdown:

```
'menu' => 'content',
```

When you install or uninstall your module, you may want to run some custom code to create database tables, add a new stream or pre-populate some data. You can do this with the `install()` and `uninstall()` methods:

```
public function install()
{
    return true;
}

public function uninstall()
{
    return true;
}
```

Finally, when releasing a new version, you may want to add a new column to a database table or adjust something else structurally at upgrade time. You can do this with the `upgrade()` method, which will be called when a new version is detected.

The `upgrade()` method is passed the currently installed version as a variable. You can then compare this version against specific version numbers to provide an upgrade. It works something like this:

```
public function upgrade($old_version)
{
    if ($old_version == '1.0')
    {
        // Do stuff...
        $old_version = '1.1';
    }
    if ($old_version == '1.1')
    {
        // Do stuff...
        $old_version = '1.2';
    }
    return true;
}
```

Permissions

It's important to ensure that only users with permission to do certain actions can do them. PyroCMS supports a group-based permissions system, where you can assign certain users to groups, and then assign roles that members of that group can play. These roles are the actions you'd like to permit the user to do–they allow you to specify precisely what permissions to give to whom.

Remember that array we returned from `info()`? We can add a `roles` key to that with our list of roles:

```
'roles' => array(
    'put_live', 'edit_live', 'delete_live'
),
```

These are arbitrary roles, and you can have as many as you like. In *Users -> Permissions*, where we can configure the permissions for each user group, the roles will be displayed with a 'proper English' label. In order to specify what role corresponds to what label (in what language), we can use a language file.

Create a **modulename/language/english/permissions_lang.php** file to list the roles and their corresponding labels. Prepend the role name with the module name, a colon, and `role_`. For example, a module called `faqs` might have these permissions:

```
$lang['faqs:role_put_live'] = 'Put question live';
$lang['faqs:role_edit_live'] = 'Edit live questions';
$lang['faqs:role_delete_live']  = 'Delete live questions';
```

We can then use some helper methods to check if the current user has been assigned this role (and thus has the permission to do this action):

```
if (group_has_role('faqs', 'put_live'))
{
    echo anchor('admin/faqs/put_live', 'Put Live',
'class="btn"');
}
```

There's even a helper method for checking permissions and calling `die()` if they don't exist. This is great for controller methods where the user really has no business being if they don't have permission:

```
public function put_live($question_id)
{
    if (role_or_die('faqs', 'put_live'))
    {
        $this->question_m->put_live($question_id);
        redirect('admin/faqs');
    }
}
```

Permissions are totally optional, and you don't have to define roles for your module to work, but it's highly recommended that you give your user fine-grained control over who can use certain bits of functionality and what data they get access to.

Sections (Multiple Controllers)

Sections allow you to use multiple controllers in your modules. If you load *Content* -> *Blog*, beneath the *Blog* bar you'll see a sub-navigation bar, with *Posts* and *Categories*. These are two sections. Sections are optional, but are a great way of organising admin panel code.

Enable sections in - yep, you guessed it - the **details.php** `info()` array. The array takes the following structure:

```
'sections' => array(
    'questions' => array(
        'name' => 'Questions',
        'uri' => 'admin/faqs'
    ),
    'categories' => array(
        'name' => 'Categories',
        'uri' => 'admin/faqs/categories'
    )
),
```

This sets up the sections list. A section is then simply a new controller in the module. Within each section controller, it's important to specify the current active section. You can do this by defining a `$section` variable:

```
class Admin_Questions extends Admin_Controller
{
    public $section = 'questions';
}
```

Again, sections are entirely optional, but can give your module pages more structure in a conventional PyroCMS way.

Shortcuts

On the far right of the blog bar, above the section navigation, you'll notice a button. It reads + *Add Post*, and button is called a shortcut. Shortcuts are enabled much like sections or permissions; through a simple array in the `info()` method:

```
'shortcuts' => array(
    array(
        'name' => 'Add Question',
        'uri' => 'admin/faqs/questions/create',
        'class' => 'add'
    )
)
```

Each section can have its own shortcuts, just nest a shortcuts array within the section array. Shortcuts are very useful for calls to action relating to the current controller, and they're used across the PyroCMS system. It's a good rule of thumb to stick to PyroCMS conventions as much as you can when building custom modules.

Controllers and Models

You'll write your module much like any other CodeIgniter application. There's nothing special to it - you just take requests like you'd expect and respond using the simple templating library. A front-end controller must inherit from `Public_Controller`:

```
class Questions extends Public_Controller { }
```

Your front-end controllers can use their own views, but it's recommended that instead of loading the view with CodeIgniter's loader, you use the template library to parse the template with Lex (the PyroCMS templating engine) and thus allowing other custom themes to override, much like we did with the blog module earlier.

You can set the custom data with `set()` and then build the template with `build()`:

```
$questions = $this->question_m->get_all();

$this->template
        ->set('questions', $questions)
        ->build('questions');
```

Custom metadata can be set too:

```
$this->template
        ->set_metadata('description', $meta['description'])
        ->set_metadata('keywords', $meta['keywords']);
```

You can even set the page title:

```
$this->template->title('FAQs');
```

The call to `build()` will look for your template name in **modulename/views**, so be sure to create your **questions.php** template file in there. Remember you can use both PHP and PyroCMS template tags in there!

Admin controllers are equally simple. In the admin panel, when you access `admin/questions`, Pyro will look for an `Admin` controller in the **questions** module. Admin controllers must inherit from `Admin_Controller`:

```
class Admin extends Admin_Controller { }
```

Sections can be routed to normally, using a sub-directory or using a custom **modulename/config/routes.php** routing file.

Models in PyroCMS are powered by Jamie Rumbelow's rather brilliant MY_Model library[6], so a CRUD base is already set up. Create a model in **modulename/models** and have it inherit from `MY_Model`. Then you can load it and use it as you'd expect:

```
class Question_m extends MY_Model { }

$this->load->model('question_m');
$this->question_m->get(1);
```

6.https://github.com/jamierumbelow/codeigniter-base-model

We've written a sample module to help you get started[7]. The core modules in PyroCMS all behave in the same way too, so if you're serious about module development you can read through them to get a real handle on how module development works.

7. https://github.com/pyrocms/sample

Plugins

Plugins are the code that power template tags. As we've seen in this book, tags allow you to output dynamic data into your templates. Using a plugin, you can inject custom functionality or output data in a specific way straight into your template.

Plugins can be a part of a module, where they'll usually provide a template layer for the more complex functionality held in the module. Alternatively, they can be simple standalone files. We'll write a standalone plugin here, but if you'd like to add a plugin to your module, all it takes is to add a **plugin.php** file to your module directory instead of in the standalone **plugins** directory, as we'll do here.

Inside your **addons/default** directory, create a **plugins** folder, and inside that, a **random_team_member.php** file. This is the name of our plugin, and we'll use it to output a random element from the *team members* stream. The reason why this is a custom plugin is so we can also extract the page URL and provide that in our template too (and hopefully learn a fair amount about PyroCMS in the process!)

We'll open up our plugin class, which needs to extend from the `Plugin` class:

```
class Plugin_Random_team_member extends Plugin {}
```

The class has a “Plugin_” prefix to avoid conflicting with any other classnames, because PyroCMS 2.2 is PHP 5.2 compmatable and does not use namespaces. In 2.3 this will be switched to namespaces instead as we move to PHP 5.3.

Next our plugin needs some metadata, so let's add our name, version and description to the class. Note here that we use an array to specify the name and description in multiple languages.

The function that will do the bulk of the work will be the `pick` function. It'll be called from the template and will return data that we can then output in whatever way we like.

When writing custom plugins, it's usually a good idea to decide how it will be used from the front-end first. That way you try to make it as simple as usable as possible.

Our template code - which we'll place on the homepage, but more on that in a moment - will look something like this:

```
{{ random_team_member:pick }}
    <a class="team-member-list" href="{{ url }}">
        <img src="{{ picture:image }}" />
    </a>
{{ /random_team_member:pick }}
```

Excellent! Let's make it work.

The first thing we need to do is create the `pick()` function. This is a normal public function:

```
public function pick()
{
```

Due to the way that the PyroCMS database structure is designed, it makes most sense to withdraw the random team member from the context of the parent page. We'll be using our tag like above, but it'd be good to make the plugin more flexible.

Because of this, we'll be defining the parent page in our plugin, but we want to allow the user to override which page it is. Make sense? If it doesn't, don't worry, it'll become crystal clear in a few minutes.

This will be a `parent_slug` attribute on the tag. We can access attributes like so:

```
$parent_slug = $this->attribute('parent_slug', 'team');
```

The first parameter to `attribute()` is the name of the parameter to fetch. The second is a default parameter in case it's omitted from the template tag.

Next, we need to fetch the ID of the page named by `$parent_slug`. This is all CodeIgniter, remember, so we can use CodeIgniter's query builder to fetch that easily. We can use existing models, or instead we can just attack the database directly:

```
$parent_id = $this->db
  ->select('id')
  ->where('slug', $parent_slug)
  ->get('pages')
  ->row('id');
```

We'll then want to fetch the random page ID. I'm getting the random page ID here (rather than just getting the whole page) because we'll also need the custom stream data. Custom stream data is split into a few tables, and PyroCMS provides some models to do the heavy lifting for us.

We can get the random post by using CodeIgniter's query builder once again. We'll set the `parent_id` to the parent page's ID, and we'll use a combination of `ORDER BY RAND()` and `LIMIT 1` to get our random post:

```
$random_id = $this->db
  ->select('id')
  ->where('parent_id', $parent_id)
  ->order_by('RAND()')
  ->limit(1)
  ->get('pages')
  ->row('id');
```

Now we have the random ID, all it takes is to call Pyro's page model and get the full page object, complete with attached stream data, straight into our plugin.

```
$random_page = $this->page_m->get($random_id);
```

It's really that easy. Now we have the page, there's one more piece to the puzzle. We want a URL! We've got access to the page's URI, so we could use the `{{ url:site }}` helper tag in the template, but why give ourselves extra work to do in the template?

We can very easily add a full URL into the object using CodeIgniter's `site_url()` helper and the page's `uri` property:

```
$random_page->url = site_url($random_page->uri);
```

So, how do we get this into the template?

```
return array($random_page);
```

We return an array - we want to use a pair of tags, so we return the array to tell the template parser that we'll have embedded tag content - and put our `$random_page` object inside.

The template parser is then smart enough to loop through the elements of the array and provide all the data for our templates. We can then simply use the custom data as much as we please in the view. And our URL is there, too!

Building a plugin, as you can see, is really, *really* easy.

We'll put our random member on the homepage. If you click on the *View Source* button on a page's WYSIWYG editor you can add in custom PyroCMS tags. When viewing the source, the homepage looks like this:

```
<p>Welcome to All About PyroCMS, a very simple and frankly
useless site designed and built for the 2013 book, Catapult
into PyroCMS, written by PyroCMS creator Phil Sturgeon with
contributions from the PyroCMS core team. <a
href="https://efendibooks.com/books/catapult-into-pyrocms">Get
your copy at Efendi Books today!</a></p>

<h4>Random Team Member</h4>
```

```
{{ random_team_member:pick }}
    <a class="team-member-list" href="{{ url }}">
        <img src="{{ picture:image }}" />
    </a>
{{ /random_team_member:pick }}
```

And that's all there is to it!

Widgets

Widgets are the final main type of addon to PyroCMS. As we learnt earlier, they are simple drag-and-drop components that can provide dynamic content to your templates and can be rearranged or reorganised by your clients. But how do we write them?

Much like a plugin, widgets can sit in a module or be used as standalone components. If you choose the former, create a **widgets** folder in your module. If it's the latter, create a **widgets** folder inside your **addons/default** directory and place them in there.

Generally widgets are comprised of three files: the widget class, a backend settings view and the front-end display view.

widgetname/views/display.php is the code that is outputted to the browser when the widget is viewed in the front-end. It's usually a simple HTML file with a bit of PHP:

```
<ul class="navigation">
    <?php foreach($posts as $post): ?>
        <li><?php echo anchor('blog/'.date('Y/m',
$post->created_on) .'/'.$post->slug, $post->title) ?></li>
    <?php endforeach ?>
</ul>
```

Usually, widgets will have a few settings that can be changed at install time. These settings are selected in **widgetname/views/form.php**, which often looks something like this:

```
<ol>
    <li class="even">
        <label>Number of posts to display</label>
        <?php echo form_input('limit', $options['limit']) ?>
    </li>
</ol>
```

Don't worry about the `<form>` tag – that's automatically inserted for us and wrapped around the form view.

There's one more piece of the widget puzzle: the widget class. This class contains the core code that powers your widget, along with information about its settings. We define our widget class with some basic information about the widget:

```
class Widget_Blog_post_list extends Widgets
{
    public $title = 'Blog post list';
    public $description =  'Display a list of blog posts';
    public $author = 'Phil Sturgeon';
    public $website = 'http://example.com/';
    public $version = '1.0.0';
}
```

We then specify what settings our widget has:

```
public $fields = array(
    array(
        'field'  => 'limit',
        'label'  => 'Number of posts to display',
        'rules'  => 'required'
    )
);
```

When the widget is about to be outputted to the front-end, the `run()` method is invoked:

```
public function run($options)
{
    $this->load->model('blog/blog_m');

    $posts = $this->blog_m
                ->limit($options['limit'])
                ->get_many_by(array( 'status' => 'live' ));

    return array( 'posts' => $posts );
}
```

Finally, options are process through the `form()` method pre-save:

```
public function form($options)
{
    $options['limit'] = (!empty($options['limit'])) ?
$options['limit'] : 5;

    return array(
        'options' => $options
        );
    }
```

Widgets are simple classes to do simple things, but they can be really useful snippets of code and can make your clients' lives that much easier. Again, it's a good idea to check out some of the core widgets in **system/cms/widgets** to see how they work under the hood and get a real feel for how a real-world widget works.

Summary

In this book we've created a site using PyroCMS, from scratch. We've examined the core concepts of the system and seen how they're used to create a real system. We've used the powerful streams engine to see how the CMS can be crafted around the system, and not the other way round. We've built a theme from scratch, overloaded a core module and used the asset plugin to power our static assets. We've examined the addon architecture and written a small plugin to interact with the PyroStreams system.

PyroCMS has existed in one form or another for six years now, and in the last two years the uptake and interaction from the community has been amazing. I strongly believe that with help from the community, theme developers and other addon developers we can give some of the “big and nasty” systems a run for their money, while making a tool perfectly usable for building content heavy sites with the least pain possible. Hopefully this book has whet your appetite for more PyroCMS goodness, and I look forward to the next six years!

I hope you've enjoyed reading Catapult into PyroCMS.

If you spot any errors in this book, we'd sincerely appreciate it if you could submit it to the errata page at https://efendibooks.com/books/catapult-into-pyrocms/errata/. If you have any questions or suggestions how to improve this book, please don't hesitate to contact me at phil@pyrocms.com or on Twitter, at @philsturgeon.

Thanks for reading!