Views and Templates

Separation of Concerns

A user interface like an HTML page should be independent of the underlying PHP code related to routing and business logic. This is fundamental to the MVC paradigm. A basic revision like converting `<h3>` to `<p>` should not demand a change in your application code. In the same manner, transforming a simple route like `GET /about` to `GET /about-us` should not have any effect on your user interface and business logic, (the view and model in MVC, or representation and method in RMR).

Mixing programming constructs and user interface components in a single file, like spaghetti coding, makes future application maintenance a nightmare.

PHP as a Template Engine

F3 supports PHP as a template engine. Take a look at this HTML fragment saved as `template.htm`:

```
<p>Hello, <?php echo $name; ?>!</p>
```

Regardless, if short tags are enabled on your server or not, this should work too:

```
<p>Hello, <?= $name ?></p>
```

The following PHP code utilizes the `View` class to render the PHP template stored in the `template.htm` file:

```
$f3=require('lib/base.php');
$f3->route('GET /',
    function($f3) {
        $f3->set('name','world');
        $view=new View;
        echo $view->render('template.htm');
        // Previous two lines can be shortened to:
        // echo View::instance()->render('template.htm');
    }
);
$f3->run();
```

The only issue with using PHP as a template engine, due to the embedded PHP code in these files, is the conscious effort needed to stick to the guidelines on separation of concerns and resist the temptation of mixing business logic with your user interface.

A Quick Look at the F3 Template Language

As an alternative to PHP templates, you can use F3's own template engine with the `Template` class. The above HTML fragment can be rewritten as:

```
<p>Hello, {{ @name }}!</p>
```

and the code needed to view this template:

```
$f3=require('lib/base.php');
$f3->route('GET /',
    function($f3) {
        $f3->set('name','world');
        echo \Template::instance()->render('template.htm');
    }
);
$f3->run();
```

> **Notice:** It is recommended to fetch the template instance with `\Template::instance()`, which ensures you always get back the same instance (like a singleton). This could be important for additional plugins to work properly, which might modify the global instance in the registry. Nevertheless you can also create a new instance with `new \Template();`, but it'll not include any registered filter, tag or event addons.

Like routing tokens used for catching variables in URLs (still remember the `GET /brew/@count` example in the previous section?), F3 template tokens begin with the `@` symbol followed by a series of letters and digits enclosed in curly braces. The first character must be alpha. Template tokens have a one-to-one correspondence with framework variables. The framework automatically replaces a token with the value stored in a variable of the same name.

In our example, F3 replaces the `@name` token in our template with the value we assigned to the name variable. At runtime, the output of the above code will be:

```
<p>Hello, world</p>
```

Worried about performance of F3 templates? At runtime, the framework parses and compiles/converts an F3 template to PHP code the first time it's displayed via `$template->render()`. The framework then uses this compiled code in all subsequent calls. Hence, performance should be the same as PHP templates, if not better due to code optimization done by the template compiler when more complex templates are involved.

Whether you use PHP's template engine or F3's own, template rendering can be significantly faster if you have APC, WinCache or XCache available on your server.

As mentioned earlier, framework variables can hold any PHP data type. However, usage of non-scalar data types in F3 templates may produce strange results if you're not careful. Expressions in curly braces will always be evaluated and converted to string. You should limit your user interface variables to simple scalars: `string`, `integer`, `boolean` or `float` data types.

But what about arrays? Fat-Free recognizes arrays and you can employ them in your templates. You can have something like:

```
<p>{{ @buddy[0] }}, {{ @buddy[1] }}, and {{ @buddy[2] }}</p>
```

And populate the `@buddy` array in your PHP code before serving the template:

```
$f3->set('buddy',array('Tom','Dick','Harry'));
```

However, if you simply insert `{{ @buddy }}` in your template, PHP will generate an `Array to string conversion` notice at runtime.

F3 allows you to embed expressions in templates. These expressions may take on various forms, like arithmetic calculations, boolean expressions, PHP constants, etc. Here are a few examples:

```
{{ 2*(@page-1) }}
{{ (int)765.29+1.2e3 }}
<option value="F" {{ @active?'selected="selected"':'' }}>Female</option>
{{ var_dump(@xyz) }}
<p>That is {{ preg_match('/Yes/i',@response)?'correct':'wrong' }}!</p>
{{ @obj->property }}
```

An additional note about array expressions: Take note that `@foo.@bar` is a string concatenation (`$foo.$bar`), whereas `@foo.bar` translates to `$foo['bar']`. If `$foo[$bar]` is what you intended, use the `@foo[@bar]` regular notation.

Framework variables may also contain anonymous functions:

```
$f3->set('func',
    function($a,$b) {
        return $a.', '.$b;
    }
);
```

The F3 template engine will interpret the token as expected, if you specify the following expression:

```
{{ @func('hello','world') }}
```

Notice: If you have an error of **UNDEFINED VARIABLE** or **UNDEFINED INDEX** during the rendering of your F3 Templates, it means the variable or array index was not previously declared in your function that is calling the template render() action. Two solutions:

- **Preferred:** Define all variables in your code, even if NULL;
  - $f3->set('myVar',NULL)
  - $f3->set('myArray.myIndex', 'First Item')
- **Quick but Dirty:** Use an extra AT symbol **@** in front of the token;
  - {{@@myArray.myIndex}}
  - to suppress the display of error message

Refer to the FatFree GitHub Issue #201 (https://github.com/bcosca/fatfree/issues/201) for more details.

Templates Within Templates

Simple variable substitution is one thing all template engines have. Fat-Free has more up its sleeves:

```
<include href="header.htm" />
```

The directive will embed the contents of the header.htm template at the exact position where the directive is stated. You can also have dynamic content in the form of:

```
<include href="{{ @content }}" />
// OR
<include href="{{ 'templates/layout/'.@content }}" />

// WRONG
<include href="templates/layout/{{ @content }}" />
```

A practical use for such template directive is when you have several pages with a common HTML layout but with different content. Instructing the framework to insert a sub-template into your main template is as simple as writing the following PHP code:

```
// switch content to your blog sub-template
$f3->set('content','blog.htm');
// in another route, switch content to the wiki sub-template
$f3->set('content','wiki.htm');
```

A sub-template may in turn contain any number of directives. F3 allows unlimited nested templates.

You can specify filenames with something other than .htm or .html file extensions, but it's easier to preview them in your Web browser during the development and debugging phase. The template engine is not limited to rendering HTML files. In fact you can use the template engine to render other kinds of files.

The `<include>` directive has an optional `if` attribute to let you specify a condition that needs to be fulfilled for the sub-template to be inserted:

```
<include if="{{ count(@items) >= 2 }}" href="items.htm" />
```

The current data hive is passed to the sub-template. You can however pass new variables or overwrite existing variables using the `with` attribute:

```
<!-- pass $a=2 to sub.htm -->
<include href="sub.htm" with="a=2" />

<!-- pass $b='something' and $c='something quoted' to sub.htm -->
<include href="sub.htm" with="b=something,c='something quoted'" />

<!-- pass uppercased value of $d to sub.htm -->
<set d="abc" />
<include href="sub.htm" with="d={{strtoupper($d)}}" /> // $d='ABC'
{{@d}} // $d='abc'
```

Notice: It's currently not possible to use a PHP constant like `TRUE` as a with-variable, since all attributes are parsed as string.

Exclusion of Segments

During the course of writing/debugging F3-powered programs and designing templates, there may be instances when disabling the display of a block of HTML may be handy. You can use the `<exclude>` directive for this purpose:

```
<exclude>
    <p>A chunk of HTML we don't want displayed at the moment</p>
</exclude>
```

That's like the `<!-- comment -->` HTML comment tag, but the `<exclude>` directive makes the HTML block totally invisible once the template is rendered.

Here's another way of excluding template content or adding comments:

```
{* <p>A chunk of HTML we don't want displayed at the moment</p> *}
```

Conditional Segments

Another useful template feature is the `<check>` directive. It allows you to embed an HTML fragment depending on the evaluation of a certain condition. Here are a few examples:

```
<check if="{{ @page=='Home' }}">
    <false><span>Inserted if condition is false</span></false>
</check>
<check if="{{ @gender=='M' }}">
    <true>
        <div>Appears when condition is true</div>
    </true>
    <false>
        <div>Appears when condition is false</div>
    </false>
</check>
```

You can have as many nested `<check>` directives as you need.

An F3 expression inside an if attribute that equates to `NULL`, an empty string, a boolean `FALSE`, an empty array or zero, automatically invokes `<false>`. If your template has no `<false>` block, then the `<true>` opening and closing tags are optional:

```
<check if="{{ @loggedin }}">
    <p>HTML chunk to be included if condition is true</p>
</check>
```

Repeating Segments

Fat-Free can also handle repetitive HTML blocks:

```
<repeat group="{{ @fruits }}" value="{{ @fruit }}">
    <p>{{ trim(@fruit) }}</p>
</repeat>
```

The `group` attribute `@fruits` inside the `<repeat>` directive must be an array and should be set in your PHP code accordingly:

```
$f3->set('fruits',array('apple','orange ',' banana'));
```

Nothing is gained by assigning a value to `@fruit` in your application code. Fat-Free ignores any preset value it may have because it uses the variable to represent the current item during iteration over the group. The output of the above HTML template fragment and the corresponding PHP code becomes:

```
<p>apple</p>
<p>orange</p>
<p>banana</p>
```

The framework allows unlimited nesting of `<repeat>` blocks:

```
<repeat group="{{ @div }}" key="{{ @ikey }}" value="{{ @idiv }}">
    <div>
        <p><span><b>{{ @ikey }}</b></span></p>
        <p>
        <repeat group="{{ @idiv }}" value="{{ @ispan }}">
            <span>{{ @ispan }}</span>
        </repeat>
        </p>
    </div>
</repeat>
```

Apply the following F3 command:

```
$f3->set('div',
    array(
        'coffee'=>array('arabica','barako','liberica','kopiluwak'),
        'tea'=>array('darjeeling','pekoe','samovar')
    )
);
```

As a result, you get the following HTML fragment:

```
<div>
    <p><span><b>coffee</b></span></p>
    <p>
        <span>arabica</span>
        <span>barako</span>
        <span>liberica</span>
        <span>kopiluwak</span>
    <p>
</div>
<div>
    <p><span><b>tea</b></span></p>
    <p>
        <span>darjeeling</span>
        <span>pekoe</span>
        <span>samovar</span>
    </p>
</div>
```

Amazing, isn't it? And the only thing you had to do in PHP was to define the contents of a single F3 variable `div` to replace the `@div` token. Fat-Free makes both programming and Web template design really easy.

The `<repeat>` template directive's `value` attribute returns the value of the current element in the iteration. If you need to get the array key of the current element, use the `key` attribute instead. The `key` attribute is optional.

`<repeat>` also has an optional counter attribute that can be used as follows:

```
<repeat group="{{ @fruits }}" value="{{ @fruit }}" counter="{{ @ctr }}">
    <p class="{{ @ctr%2?'odd':'even' }}">{{ trim(@fruit) }}</p>
</repeat>
```

Internally, F3's template engine records the number of loop iterations and saves that value in the variable/token `@ctr`, which is used in our example to determine the odd/even classification.

Embedding Javascript and CSS

If you have to insert F3 tokens inside a `<script>` or `<style>` section of your template, the framework will still replace them the usual way:

```
<script type="text/javascript">
    function notify() {
        alert('You are logged in as: {{ @userID }}');
    }
</script>
```

Embedding template directives inside your `<script>` or `<style>` tags requires no special handling:

```
<script type="text/javascript">
    var discounts=[];
    <repeat group="{{ @rates }}" value="{{ @rate }}">
        // whatever you want to repeat in Javascript, e.g.
        discounts.push("{{ @rate }}");
    </repeat>
</script>
```

Document Paths

There are some things to consider about public paths used in page links or style/script or image includes. You probably want to use relative paths:

```
<link href="ui/css/base.css" type="text/css" rel="stylesheet" />
<script src="ui/js/base.css"></script>
<a href="category-abc/article-xyz">read more</a>
<img src="ui/img/img1.jpg" alt="Image 1" />
```

These paths could work on your index page `GET /`, but will cause problems when you navigate to `GET /@category/@article`, or run your application from a sub-directory in your webspace public root (i.e. `http://domain.com/my-F3-App/`). To solve this you should either prepend all of your paths with the public base path like this:

```
<link href="{{@BASE}}/ui/css/base.css" type="text/css" rel="stylesheet" />
<script src="{{@BASE}}/ui/js/base.css"></script>
<a href="{{@BASE}}/category-abc/article-xyz">read more</a>
<img src="{{@BASE}}/ui/img/img1.jpg" alt="Image 1" />
```

Or use the HTML base Tag (http://www.w3schools.com/tags/tag_base.asp) to specify a document-wide default base path for relative links:

```
<base href="{{@SCHEME.'://'.@HOST.@BASE.'/'}}">
```

There are some side-effects (http://stackoverflow.com/questions/1889076/is-it-recommended-to-use-the-base-html-tag) using a `<base>` tag in your document: Mainly you need to prefix all page anchor links as well (`<a href="{{@PATH}}#top">go to top</a>`).

Document Encoding

Fat-Free uses the UTF-8 character set by default. You can override this behavior by setting the `ENCODING` (quick-reference#encoding) system variable like this:

```
$f3->set('ENCODING','ISO-8859-1');
```

Once you inform the framework of the desired character set, F3 will use it for all HTML and XML templates until altered again.

Please note that Fat-Free uses the `ENCODING` system variable not only for the document encoding but also for the database encoding. If your document encoding (e.g. ISO-8859-1) differs from your database encoding (e.g. latin1), make sure that you supply the divergent database encoding as PDO attribute for the `$options` parameter when establishing the database connection (sql#constructor).

All Kinds of Templates

As mentioned earlier in this section, the framework isn't limited to HTML templates. You can process XML templates just as well. The mechanics are pretty much similar. You still have the same `{{ @variable }}` and `{{ expression }}` tokens, `<repeat>` , `<check>` , `<include>` , and `<exclude>` directives at your disposal. Just tell F3 that you're passing an XML file instead of HTML:

```
echo Template::instance()->render('template.xml','application/xml');
```

The second argument represents the MIME type of the document being rendered.

The View component of MVC covers everything that doesn't fall under the Model and Controller, which means your presentation can and should include all kinds of user interfaces, like RSS, e-mail, RDF, FOAF, text files, etc. The example below shows you how to separate your e-mail presentation from your application's business logic:

```
MIME-Version: 1.0
Content-type: text/html; charset={{ @ENCODING }}
From: {{ @from }}
To: {{ @to }}
Subject: {{ @subject }}

<p>Welcome, and thanks for joining {{ @site }}!</p>
```

Save the above e-mail template as welcome.txt. The associated F3 code would be:

```
$f3->set('from','<no-reply@mysite.com>');
$f3->set('to','<slasher@throats.com>');
$f3->set('subject','Welcome');
ini_set('sendmail_from',$f3->get('from'));
mail(
    $f3->get('to'),
    $f3->get('subject'),
    Template::instance()->render('email.txt','text/html')
);
```

Tip: Replace the SMTP mail() function with imap_mail() if your script communicates with an IMAP server.

Now isn't that something? Of course, if you have a bundle of e-mail recipients, you'd be using a database to populate the firstName, lastName, and email tokens.

Here's an alternative solution using the F3's SMTP plug-in:

```
$mail=new SMTP('smtp.gmail.com',465,'SSL','account@gmail.com','secret');
$mail->set('from','<no-reply@mysite.com>');
$mail->set('to','"Slasher" <slasher@throats.com>');
$mail->set('subject','Welcome');
$mail->send(Template::instance()->render('email.txt'));
```

Multilingual Support

F3 supports multiple languages right out of the box.

First, create a dictionary file with the following structure (one file per language):

```
<?php
return array(
    'love'=>'I love F3',
    'today'=>'Today is {0,date}',
    'pi'=>'{0,number}',
    'money'=>'Amount remaining: {0,number,currency}'
);
```

Save it as `dict/en.php`. Let's create another dictionary, this time for German. Save the file as `dict/de.php`:

```
<?php
return array(
    'love'=>'Ich liebe F3',
    'today'=>'Heute ist {0,date}',
    'money'=>'Restbetrag: {0,number,currency}'
);
```

Dictionaries are nothing more than key-value pairs. F3 automatically instantiates framework variables based on the keys in the language files. As such, it's easy to embed these variables as tokens in your templates.

Notice: Dictionary key-value pairs become F3 variables once referenced. Make sure the keys do not conflict with any framework variable instantiated via `$f3->set()`, `$f3->mset()`, or `$f3->config()`. **In Addition, you can also use the PREFIX (quick-reference#PREFIX) var to set a common key that prepends every language key.**

Examples of multiple languages using the F3 template engine:

```
<h1>{{ @love }}</h1>
<p>
    {{ @today,time() | format }}.<br />
    {{ @money,365.25 | format }}<br />
    {{ @pi, 3.1415 | format }}
</p>
```

And the longer version that utilizes PHP as a template engine:

```
<?php $f3=Base::instance(); ?>
<h1><?php echo $f3->get('love'); ?></h1>
<p>
    <?php echo $f3->get('today',time()); ?>.<br />
    <?php echo $f3->get('money',365.25); ?><br />
    <?php echo $f3->get('pi', 3.1415); ?>
</p>
```

Next, we instruct F3 to look for dictionaries in the `dict/` folder:

```
$f3->set('LOCALES','dict/');
```

But how does the framework determine which language to use? F3 will detect it automatically by looking at the HTTP request headers first, specifically the `Accept-Language` header sent by the browser.

To override this behavior, you can trigger F3 to use a language specified by the user or application:

```
$f3->set('LANGUAGE','de');
```

Main language of your website

In the above example, the key `pi` exists only in the English dictionary. The framework will always use a fallback language to populate keys that are not present in the specified or detected language(s). In our example, the key `pi` present in the English dictionary has been picked up because English is the default fallback language set by F3 at startup. Hence it's important to make sure **all** the multi-languages texts used on your website *are defined in the dictionary of the fallback language*, otherwise an error, or at least a warning or notice, will be triggered at runtime, when the undefined variable will be referenced.

If the main language of your website is not English, and you haven't translated all the strings in the others languages, you must instruct F3 to use your native language reference dictionary as the fallback. You can easily do it thanks to the `FALLBACK` hive variable:

```
$f3->set('FALLBACK','it');  // Italiano as default fallback language
```

Language Variants

You may also create dictionary files for language variants like `en-US`, `es-AR`, etc. In this case, F3 will use the language variant first, e.g. `es-AR`. If there are keys that do not exist in the variant dictionary, the framework will look up the key in the root language, e.g. `es`, and, when still not found, use the defined fallback language file.

Did you notice the peculiar `'Today is {0,date}'` pattern in our previous example? F3's multilingual capability hinges on string/message formatting rules of the ICU project. The framework uses its own subset of the ICU string formatting implementation. There is no need for PHP's `intl` extension to be activated on the server.

One more thing: F3 can also load **.ini**-style formatted files as dictionaries:

```
love = I love F3
today = Today is {0,date}
pi = {0,number}
money = Amount remaining: {0,number,currency}
multiline = It's also possible to have language keys \
            spread over multiple lines

[module.user.validation]
name.required = Please enter your name.
mail.invalid = This mail address is not valid.
```

Save it as `dict/en.ini` so the framework can load it automatically.

> The LANGUAGE variable accepts the same kind of string as the HTTP Accept-Language header, which is a comma separated list of 2-letter language codes optionally followed by **a hypen** and a 2-letter country code.
>
> **NB:** Even though POSIX locales use an underscore as a separator (es_BR.UTF-8), you should define the LANGUAGE variable with a hyphen (es-BR). Dictionary files follow the same rule (es-BR.php / es-BR.ini).

For some more technical details, see the Base->language (base#language) method.

Data Sanitation

By default, both view handler and template engine escapes all rendered variables, i.e. converted to HTML entities to protect you from possible XSS and code injection attacks. On the other hand, if you wish to pass valid HTML fragments from your application code to your template:

```
$f3->set('ESCAPE',FALSE);
```

This may have undesirable effects. You might not want all variables to pass through unescaped. Fat-Free allows you to unescape variables individually. For F3 templates use the `raw` filter:

```
{{ @html_content | raw }}
```

In the case of PHP templates, use the raw method directly:

```
<?php echo $this->raw($html_content); ?>
```

As an addition to auto-escaping of F3 variables, the framework also gives you a free hand at sanitizing user input from HTML forms:

```
$f3->scrub($_GET,'p; br; span; div; a');
```

This command will strip all tags (except those specified in the second argument) and unsafe characters from the specified variable. If the variable contains an array, each element in the array is sanitized recursively. If an asterisk (*) is passed as the second argument, `$f3->scrub()` permits all HTML tags to pass through untouched and simply remove unsafe control characters.

Extending filters and custom tags

With the F3 template engine you can also setup own expression filters like `{{ @desc,100 | crop }}` and also combine them like `{{ @desc,100 | crop,raw }}`. Therefore you just need to register a new filter with the filter (preview#filter) method. For own html tag / element handlers that could render anything you want, check out the template->extend (template#extend) method. For more detailed descriptions about custom filters and how the whole templating system works, see the extended templating (extended-templating) part of the user guide.

← 3. Framework Variables (framework-variables)

5. Databases → (databases)