Extended Templating

This guide tells you all about the F3 Template engine, how it technically works and how to extend it for highly flexible HTML templating, so you can get an idea of all the possibilities it offers to you.

The template engine of Fat-Free basically contains 3 different components - your template, the rendering process, which builds the pre-compiled cached template, and the sandbox. Many parts are extensible, which gives us the opportunity to create even more flexibility for our **HTML** templates. We will have a deep introspect to the View (view), Preview (preview) and Template (template) classes to make you understand the magic behind the scene.

Layers of templating

To make the explanation a bit more tangible, we'll go through several layers of how the final HTML is assembled and send to your browser.

1. Template file

A template file contains of HTML markup, inline tokens `{{@title}}` and special markup elements that add logical control structure to your templates, such as `check` or `repeat` blocks. These are easy for you to write and maintain and can give you a lot of flexibility. Let's take an example:

```
<h1>News</h1>
<repeat group="{{@articles}}" value="{{@article}}">
    <h2>{{@article.title}}</h2>
    <p>{{@article.text}}</p>
    <img src="{{'ui/images/'.@article.image}}" />
</repeat>
```

2. Template

The template class takes the HTML / XML template and checks it for known tag-handlers. Finally it builds a tree structure of nodes out of those registered handlers. For our example, it'll look like this:

```
array(2) {
  [0]=>
  string(14) "<h1>News</h1>
"
  [1]=>
  array(1) {
    ["repeat"]=>
    array(2) {
      ["@attrib"]=>
      array(2) {
        ["group"]=>
        string(13) "{{@articles}}"
        ["value"]=>
        string(12) "{{@article}}"
      }
      [0]=>
      string(106) "
    <h2>{{@article.title}}</h2>
    <p>{{@article.text}}</p>
    <img src="{{'ui/images/'.@article.image}}" />
"
    }
  }
}
```

As you can see, nodes are represented as multi-dimensional arrays and the simple HTML between is just a string. Each node handler is called with the given tag attributes and its inner content and can render final HTML markup for static template snippets, or PHP code, to leave some dynamic aspects in your final template document.

We are also able to define our own node handlers to whatever HTML tag we like (or create new ones). This way we could add a function that receives all `<img>` image tags and add additional attributes we could use for a JavaScript lightbox, or fallback image paths for responsive image rendering in the front-end.

Let's have a look at how to create a node handler:

```
\Template::instance()->extend('img',function($node){
    var_dump($node);
    /*
    array(1) {
        ["@attrib"]=> array(1) {
            ["src"]=> string(25) "{{'ui/images/'.@article.image}}"
        }
    }
    */
});
```

Our node handler receives the full tag description in the `$node` array. The `@attrib` array key contains all tag attributes that are present. All other keys in `$node` would represent some inner content within our tag. Well, `<img>` elements usually have no inner content, but in case of a `<h1>` it would include all content and nodes between the starting and closing tag.

Okay, so imagine you would like to resize all image files used by the `<img>` elements. What we need to do is to get the real image path, render it with a smaller size and put the new image link into the HTML tag. Well that sounds easy, but as you can see - trying to get the real image link is already the first problem. Since we used a token in the `src` attribute, we will never know the real image path at the time we are rendering this template tag. But this rendering process will result in a pre-compiled and cached PHP / HTML template. So what we are going to do is to create some PHP code, rather than the final image tag. This code get executed when the pre-compiled template is called and builds the final HTML. So in general we differ **static tag handlers** that just creates final static HTML code and **dynamic tag handlers** which creates dynamic code, that may execute additional other code to render the final HTML at runtime.

So in first place, we need to create a class that can be called from the inside of the pre-compiled template. The basic code could look like this:

```
class ImageViewHelper extends \Prefab {

  static public function render($node) {
    $attr = $node['@attrib'];
    $path = \Template::instance()->token($attr['src']);
    $out='<?php $imgPath = \ImageViewHelper::instance()->build('.$path.'); ?>'
        .'<img src="<?php echo $imgPath;?>" />';
    return $out;
  }
  function build($path) {
    $f3 = \Base::instance();
    $file_name = $f3->hash($path.'450x300').'.jpg';
    $file_path = $f3->get('TEMP').$file_name;
    if (!is_file($file_path)) {
      $imgObj = new \Image($path);
      $imgObj->resize(450,300,true,true);
      $file_data = $imgObj->dump('jpeg');
      $f3->write($file_path, $file_data);
    }
    return $file_path;
  }
}
```

We will register this new tag handler with

```
\Template::instance()->extend('img','ImageViewHelper::render');
```

The `render` method needs to be static for this purpose and is called during the pre-compiling, whereas `build` is called every time the pre-compiled template is loaded to create the live dynamic result of our image path.

Now, what's about having some additional attributes, like a class or anything else which also could have used dynamic tokens? Let's extend our template to see how to handle dynamic code:

```
<img src="{{'ui/images/'.@article.image}}" class="{{@article.imageType}}" {{@article.
lightbox?'rel="lightbox[]"':''}}/>
```

We've just added a `class` attribute that uses a token, which represents a dynamic value from the F3 hive. Furthermore we got an inline expression here, which is also passed to our tag-handler. If we inspect our tag handler's `$node` array now, we'll see something like this:

```
array(1) {
  ["@attrib"]=>
  array(3) {
    ["src"]=>
    string(25) "{{'ui/images/'.@article.image}}"
    ["class"]=>
    string(22) "{{@article.imageType}}"
    [0]=>
    string(45) "{{ @article.lightbox?'rel="lightbox[]"':'' }}"
  }
}
```

So we got 2 new keys here within our `@attrib` array. Notice that numeric keys represent inline expressions OR value-less attributes like `<input type="radio" checked />`. In order to get those working or to bypass these attributes, regardless if we need them or not in our handler itself, we need to convert these expressions and tokens to executable PHP code. Therefore we can use this function:

```
protected function resolveAttr(array $attr) {
    $tmp = \Template::instance();
    $out = '';
    foreach ($attr as $key => $value) {
        // build dynamic tokens
        if (preg_match('/{{(.+?)}}/s', $value))
            $value = $tmp->build($value);
        if (preg_match('/{{(.+?)}}/s', $key))
            $key = $tmp->build($key);
        // inline token
        if (is_numeric($key))
            $out .= ' '.$value;
        // value-less parameter
        elseif ($value == NULL)
            $out .= ' '.$key;
        // key-value parameter
        else
            $out .= ' '.$key.'="'.$value.'"';
    }
    return $out;
}
```

We should now use this function in our render method and use the new computed attributes to inject them in our final HTML element:

```
static public function render($node) {
    $attr = $node['@attrib'];
    $path = \Template::instance()->token($attr['src']);
    unset($attr['src']); // remove existing src key, we'll handle this on our own
    $attr = self::instance()->resolveAttr($attr); // assemble all remaining attribute
s
    $out='<?php $imgPath = \ImageViewHelper::instance()->build('.$path.'); ?>'
        .'<img src="<?php echo $imgPath;?>"'.$attr.' />';
    return $out;
}
```

And that's it. Congratulations! You just build your first own dynamic tag renderer. If we have an article array like this one:

```
array(
    'imageType'=>'big',
    'image'=>'wallpaper.jpg',
    'lightbox'=>true,
),
```

it would be rendered as `<img src="tmp/0fs02ehlcd7.jpg" class="big" rel="lightbox[]" />` now, where the new image path points to our resized image.

In case you want to create a tag handler that may contain additional content, you need to render this properly by yourself.

In your tag handler function, you'll receive the $node parameter. This includes an "@attrib" key. This key contains all attributes that were defined on your directive. Everything else in $node is the inner content of your directive. To proberly render this content, use the Template->build method like this:

```
static public function render($node) {
    $attr = $node['@attrib'];
    unset($node['@attrib']);
    // do things
    // ...
    $content = (isset($node[0])) ? \Template::instance()->build($node) : '';
    return '<div>'.$content.'</div>';
}
```

3. Preview

The Preview class mainly takes care about converting your template and its expressions to PHP code - the so called pre-compiled template. So any tokens that are echo'd `{{ }}` or just executed `{~ ~}` can be used for dynamic templating, not just with HTML markup, but also with non-XML compatible template system like HAML, YAML, markdown or simple text files.

It takes all those little tokens and renders them into PHP code that is cached in the next step. So if you have a token like `{{@title}}` it'll become `<?php echo $title?>` . Rendering tokens can also be extended with filters (or also called modifiers). Some filters that are already included are esc (view#esc), raw (view#raw) and format (base#format). Filters can be applied to any expression using a pipe char, i.e. `{{ @text | raw }}` .

This would render the token as the following:

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

So we are able to push all template variables through some filters before sending them into the output. We can also defined our own filters. For this purpose we need to register a handler or callback method as our new filter for the template class (or just the preview class, if you only use that). Let's try this:

```
class TemplateFilter extends \Prefab {
    public function badwords($val) {
        return str_replace(array(
            'damn',
            'asshole'
        ),array(
            'cute',
            'guy'
        ),$val);
    }
}
// register a filter for the Preview engine:
\Preview::instance()->filter('badwords','\TemplateFilter::instance()->badwords');
// or using the Template engine:
\Template::instance()->filter('badwords','\TemplateFilter::instance()->badwords');
```

If we now use `\Preview::instance()->render` to render our templates, we can now use the new custom filter. So if you have a variable like `$f3->set('text','you are a damn asshole!');` and send this to our new filter `{{ @text | badwords}}` you'll get the cleaned result `you are a cute guy!`.

We can also add additional parameters to our filters. Let's see this in a new filter - add this crop-function to the class:

```
function crop($val,$len) {
    return substr($val,0,$len);
}
```

Now you can crop your text to a max of 100 chars, right from the inside of the template: `{{ @text, 100 | crop }}`

You can also chain multiple filters, where the first filter receives the parameters and sends its output to the next filter. In example `{{@text,100|crop,raw}}` renders as:

```
<?php echo $this->raw($this->crop($text,100)); ?>
```

With these custom expression filters you can add some interesting and useful extensions for even more flexibility.

4. Pre-Compiled PHP

This is just another layer of your original template file. Using the template class and its descendant, the template was converted into a raw PHP template file. If you are fine with a simple PHP based template, you can just write your own PHP / HTML mixed templates and build them with the View class. Pre-compiled or also called pre-rendered templates are saved in the F3 TEMP (quick-reference#temp) directory and checked against the last modified time of the original template file. Therefore this cached template only builds upon a change on the original file, or if the temp-dir was cleared.

5. View

This View class is used to create a sandbox where the PHP templates are rendered in. Sandbox means that it takes all hive variables and sanitizes its data, so all dynamic data that get included into your template won't echo any bad or insecure content like HTML tags or scripts that might break your layout or lead to XSS attacks (controlled by ESCAPE (quick-reference#escape) var).

Executing dynamic code from extensions or filters in the sandbox also means that changing variables in the real F3 hive (using set (base#set) method) will not effect the variables that are in the current sandbox scope. That means, if you change the hive var `title` from a custom function that was called from the inside of the view-template (i.e. using `\Base::instance()->set`), any token like `{{@title}}` will still contain the original value that was passed into the sandbox.

6. Final HTML

If the PHP templates are executed in the sandbox, they will create and return the final HTML content.

That's all about the templating magic.