Template

F3's own lightning fast and extendable template engine gives you all the flexibility you need for modern and clean templating. This Template plugin extends the Preview (preview) class and is especially made for XML-style templates. Make sure you have read the user guide section about templating (views-and-templates#a-quick-look-at-the-f3-template-language).

> If you are looking for a summarizing overview of the available template directives and its syntax, see the Quick Reference (quick-reference#template-directives)

Namespace: `\`
File location: `lib/template.php`

Instantiation

Return unique class instance

```
$template = \Template::instance();
```

The Template class uses the Prefab (prefab-registry) factory wrapper, so you can, and you should, grab the same instance of that Template class at any point of your code.

extend

Extend F3 template engine with a custom tag

```
void extend ( string $tag, callback $func )
```

The `extend` function allows you to create your own custom template tag. It can be seen as a hook to the F3 template engine.

`$tag` is the name of your custom tag (without the `<` and `>` indeed). Don't name your tag after an existing F3 template directive (quick-reference#include) as your tag handler won't be called. (Read: To date, you can't use `extend` to override an existing F3 template directive)

`$func` is the name of your custom function that F3 will callback when it finds your custom `$tag` during the rendering of a template.

So, to define a new custom tag & register/hook its custom handler, it gives:

```
\Template::instance()->extend('my_new_special_tag','MyImageViewHelper::my_tag_rendere
r');
```

But let's have a look at a functional example to see how it basically works:

The idea is to create a new HTML `<image>` tag that would have the ability to resize images on the fly according to the `width` and `height` attributes found in the markup.

Let's do that now:

```
class ImageViewHelper {

    static public function render($args) {
        // retrieve the attributes of the template tag, as found in the template
        // in our case, we expect 'src', 'width' and 'height', and optionally 'crop'
        $attr = $args['@attrib']; // provided by the F3 template engine

        $imagepath = $attr['src'];
        $imgObj = new \Image($imagepath);
        $imgObj->resize(
            $attr['width'],
            $attr['height'],
            ((isset($attr['crop']) && $attr['crop']=='true') ? true : false)
        );
        $f3 = \Base::instance();
        // to avoid clash, build a unique name for the new generated image
        $file_name = $f3->hash($imagepath.$attr['width'].$attr['height']).'.png';
        // save it for example in TEMP
        $imagepath = $f3->get('TEMP').$file_name;
        // convert it to PNG and save it to a file
        $f3->write($imagepath, $imgObj->dump('png'));
        // done! return the HTML markup
        return
            sprintf ('<img src="/%s" width="%u" height="%u" />',
                        $imagepath,$attr['width'],$attr['height']);
    }
}

// register the tag renderer either in index.php or in your view controller
\Template::instance()->extend('image','ImageViewHelper::render');
```

The basic snippet above takes all html tags that look like this:

```
<image src="images/south-park.jpg" width="60" height="60" crop="true" />
```

and scales the image given by the `src` attribute to the specified `width="60" height="60"` dimensions, then copies the new scaled image to the `TEMP` folder and finally generates an HTML output similar to `<img src="/tmp/0dgsnl2kmnb.png" width="60" height="60" />` in your template.

Knowing that F3 templates are all pre-rendered and cached. This way our Image Tag Renderer will only process the file once and not on every request.

Combine your tag processing by using the token (preview#token) method to add some support for dynamic values in your tag attributes and write some php calls to the result to generate a fully flexible view helper.

parse

Parse string for template directives and tokens

```
string|array parse ( string $text )
```

This method parses the template string and builds a tree structure of handled nodes found in the template.

In example:

```
<div>
    <h1>My favorite books</h1>
    <ul>
    <F3 group="{{ @books }}" value="{{ @book }}">
        <li>{{ @book.title }}</li>
    </F3>
    </ul>
</div>
```

returns:

```
array (size=3)
    0 => string '<div>
        <h1>My favorite books</h1>
        <ul>
        ' (length=53)
    1 =>
        array (size=1)
            'repeat' => array (size=2)
                '@attrib' => array (size=2)
                    'group' => string '{{ @books }}' (length=12)
                    'value' => string '{{ @book }}' (length=11)
        0 => string '<li>{{ @book.title }}</li>' (length=38)
        2 => string '</ul>
  </div>' (length=19)
```