Twital Documentation

Release 1.0-alpha

Asmir Mustafic

Mar 30, 2017

Contents

Twital is a small "plugin" for Twig (a template engine for PHP) that adds some shortcuts and makes Twig's syntax more suitable for HTML based (XML, HTML5, XHTML, SGML) templates. Twital takes inspiration from PHPTal, TAL and AngularJS (just for some aspects), mixing their language syntaxes with the powerful Twig templating engine system.

To better understand the Twital's benefits, consider the following **Twital** template, which simply shows a list of users from an array:

```
<ul t:if="users">
    <li t:for="user in users">
        {{ user.name }}
    </li>
</ul>
```

To do the same thing using Twig, you need:

```
{% if users %}
    <ul>
        {% for user in users %}
            <li>
                {{ user.name }}
            </li>
        {% endfor %}
    </ul>
{% endif %}
```

As you can see, the Twital template is **more readable**, **less verbose** and and **you don't have to worry about opening and closing block instructions** (they are inherited from the HTML structure).

One of the main advantages of Twital is the *implicit* presence of control statements, which makes templates more readable and less verbose. Furthermore, it has all Twig functionalities, such as template inheritance, translations, looping, filtering, escaping, etc. Here you can find a *complete list of Twital attributes and nodes.*.

If some Twig functionality is not directly available for Twital, you can **freely mix Twig and Twital** syntaxes.

In the example below, we have mixed Twital and Twig syntaxes to use Twig custom tags:

```
<h1 t:if="users">
    {% custom_tag %}
        {{ someUnsafeVariable }}
    {% endcustom_tag %}
</h1>
```

CHAPTER 1

Installation

There are two recommended ways to install Twital via Composer:

- using the `composer require` command:

```
composer require 'goetas/twital:0.1.*'
```

- adding the dependency to your `composer.json` file:

```
"require": {
    ..
    "goetas/twital":"0.1.*",
    ..
}
```

CHAPTER 2

Getting started

First, you have to create a file that contains your template (named for example `demo.twital.html`):

```
<div t:if="name">
    Hello {{ name }}
</div>
```

Afterwards, you have to create a PHP script that instantiate the required objects:

```
<?php

require_once '/path/to/composer/vendor/autoload.php';
use Goetas\Twital\TwitalLoader;

$fileLoader = new Twig_Loader_Filesystem('/path/to/templates');
$twitalLoader = new TwitalLoader($fileLoader);

$twig = new Twig_Environment($twitalLoader);
echo $twig->render('demo.twital.html', array('name' => 'John'));
```

That's all!

Note: Since Twital uses Twig to compile and render templates, their performance is the same.

CHAPTER 3

Contents

Tags reference

if

The Twital instruction for Twig's `if` tag is the‘‘t:if‘‘ attribute.

```
<p t:if="online == false">
    Our website is in maintenance mode. Please, come back later.
</p>
```

`elseif` and `else` are not *well* supported, but you can always combine Twital with Twig.

```
<p t:if="online_users > 0">
    {%if online_users == 1%}
        one user
    {% else %}
        {{online_users}} users
    {% endif %}
</p>
```

But if you are really interested to use `elseif` and `else` tags with Twital you can do it anyway.

```
<p t:if="online">
    I'm online
</p>
<p t:elseif="invisible">
    I'm invisible
</p>
<p t:else="">
    I'm offline
</p>
```

This syntax will work if there are no non-space charachters between the `p` tags.

This example will not work:

```
<p t:if="online">
    I'm online
</p>
<hr />
<p t:else="">
    I'm offline
</p>

<p t:if="online">
    I'm online
</p>
some text...
<p t:else="">
    I'm offline
</p>
```

Note: To learn more about the Twig `if` tag, please refer to Twig official documentation.

for

The Twital instruction for Twig's `for` tag is the `t:for` attribute.

Loop over each item in a sequence. For example, to display a list of users provided in a variable called `users`:

```
<h1>Members</h1>
<ul>
    <li t:for="user in users">
        {{ user.username }}
    </li>
</ul>
```

Note: For more information about the `for` tag, please refer to Twig official documentation.

set

The Twital instruction for Twig `set` tag is the `t:set` attribute.

You can use `set` to assign variables. The syntax to use the `set` attribute is:

```
<p t:set=" name = 'tommy' ">Hello {{ name }}</p>
<p t:set=" foo = {'foo': 'bar'} ">Hello {{ foo.bas }}</p>
<p t:set=" name = 'tommy', surname='math' ">
    Hello {{ name }} {{ surname }}
</p>
```

Note: For more information about the `set` tag, please refer to Twig official documentation.

block

The Twital instruction for Twig `block` tag is `t:block` node.

To see how to use it, consider the following base template named `layout.html.twital`:

```
<html>
    <head>
        <title>Hello world!</title>
    </head>
    <body t:block="content">
        Hello!
    </div>
</html>
```

To improve the greeting message, we can extend it using the `t:textends` node, so we can create a new template called `hello.html.twital`.

```
<t:extends from="layout.html.twital">
    <t:block name="content">
        Hello {{name}}!
    </t:block>
</t:extends>
```

As you can see, we have overwritten the content of the `content` block with a new one. To do this, we have used a `t:block` node.

Of course, if needed, you can also **call the parent block** from inside. It is simple:

```
<t:extends from="layout.html.twital">
    <t:block name="content">
        {{parent()}}
        Hello {{name}}!
    </t:block>
</t:extends>
```

Note: To learn more about template inheritance, you can read the Twig official documentation

extends

The Twital instruction for Twig `extends` tag is `t:extends` node. To see how to use it, take a look at this example:

Consider the following base template named `layout.html.twital`. Here we are creating a simple page that says hello to someone.

With the *t:block* attribute we mark the body content as extensibile.

```
<html>
    <head>
        <title>Hello world!</title>
    </head>
    <body>
        <div t:block="content">
        Hello!
        </div>
```

```
    </div>
</html>
```

To improve the greating message, we can extend it using the `t:textends` node, so we can create a new template called `hello.html.twital`.

```
<t:extends from="layout.html.twital">
    <t:block name="content">
        Hello {{name}}!
    </t:block>
</t:extends>
```

As you can see, we have overwritten the content of the `content` block with a new one. To do this, we have used a `t:block` node.

You can also **extend a Twig Template**, so you can mix Twig and Twital Templates.

```
<t:extends from="layout.twig">
    <t:block name="content">
        Hello {{name}}!
    </t:block>
</t:extends>
```

Sometimes it's useful to obtain the layout **template name from a variable**: to do this you have to add the Twital namespace to attribute name:

```
<t:extends t:from="layoutVar">
    <t:block name="content">
        Hello {{name}}!
    </t:block>
</t:extends>
```

Now `hello.html.twital` can inherit dynamically from different templates. Now the tempalte name can be any valid Twig expression.

Note: To learn more about template inheritance, you can read the Twig official documentation.

---

embed

The Twital instruction for Twig `embed` tag is `t:embed` node.

The embed tag combines the behaviour of include and extends. It allows you to include another template's contents, just like include does. But, it also allows you to override any block defined inside the included template, like when extending a template.

To learn about the usefulness of *embed*, you can read the official documentation.

Now, let's see how to use it, take a look to this example:

```
<t:embed from="teasers_skeleton.html.twital">
    <t:block name="left_teaser">
        Some content for the left teaser box
    </t:block>
    <t:block name="right_teaser">
        Some content for the right teaser box
```

```
    </t:block>
<t:embed>
```

You can add additional variables by passing them after the `with` attribute:

```
<t:embed from="header.html" with="{'foo': 'bar'}">
    ...
</t:embed>
```

You can disable the access to the current context by using the `only` attribute:

```
<t:embed from="header.html" with="{'foo': 'bar'} only="true">
    ...
</t:embed>
```

You can mark an include with `ignore-missing` attribute in which case Twital will ignore the statement if the template to be included does not exist.

```
<t:embed from="header.html" with="{'foo': 'bar'} ignore-missing="true">
    ...
</t:embed>
```

`ignore-missing` can not be an expression; it has to be evaluated only at compile time.

To use Twig expressions as template name you have to use a namespace prefix on 'form' attribute:

```
<t:embed t:from="ajax ? 'ajax.html' : 'not_ajax.html' ">
    ...
</t:embed>
<t:embed t:from="['one.html','two.html']">
    ...
</t:embed>
```

Note: For more information about the `embed` tag, please refer to Twig official documentation.

See also:

include

include

The `include` statement includes a template and returns the rendered content of that file into the current namespace:

```
<t:include from="header.html"/>
    Body
<t:include from="footer.html"/>
```

A little bit different syntax to include a template can be:

```
<div class="content" t:include="news.html">
    <h1>Fake news content</h1>
    <p>Lorem ipsum</p>
</div>
```

In this case, the content of div will be replaced with the content of template 'news.html'.

You can add additional variables by passing them after the `with` attribute:

```
<t:include from="header.html" with="{'foo': 'bar'}"/>
```

You can disable the access to the current context by using the `only` attribute:

```
<t:include from="header.html" with="{'foo': 'bar'} only="true"/>
```

You can mark an include with the `ignore-missing` attribute in which case Twital will ignore the statement if the template to be included does not exist.

```
<t:include from="header.html" with="{'foo': 'bar'} ignore-missing="true"/>
```

`ignore-missing` can not be an expression; it has to be evauluated only at compile time.

To use Twig expressions as template name you have to use a namespace prefix on 'form' attribute:

```
<t:include t:from="ajax ? 'ajax.html' : 'not_ajax.html' " />
<t:include t:from="['one.html','two.html']" />
```

Note: For more information about the `include` tag, please refer to Twig official documentation.

import

The Twital instruction for Twig `import` tag is `t:import` node.

Since Twig supports putting often used code into *macros*. These macros can go into different templates and get imported from there.

There are two ways to import templates: (1) you can import the complete template into a variable or (2) request specific macros from it.

Imagine that we have a helper module that renders forms (called `forms.html`):

```
<t:macro name="input" args="name, value, type">
    <input type="{{ type|default('text') }}" name="{{ name }}" value="{{ value|e }}" /
↪>
</t:macro>

<t:macro name="textarea" args="name, value">
    <textarea name="{{ name }}">{{ value|e }}</textarea>
</t:macro>
```

To use your macro, you can do something like this:

```
<t:import from="forms.html" alias="forms"/>
<dl>
    <dt>Username</dt>
    <dd>{{ forms.input('username') }}</dd>
    <dt>Password</dt>
    <dd>{{ forms.input('password', null, 'password') }}</dd>
    {{ forms.textarea('comment') }}
</dl>
```

If you want to import your macros directly into your template (without referring to it with a variable):

```
<t:import from="forms.html" as="input as input_field, textarea"/>
<dl>
    <dt>Username</dt>
    <dd>{{ input_field('username') }}</dd>
    <dt>Password</dt>
    <dd>{{ input_field('password', '', 'password') }}</dd>
</dl>
<p>{{ textarea('comment') }}</p>
```

Tip: To import macros from the current file, use the special `_self` variable for the source.

Note: For more information about the `import` tag, please refer to Twig official documentation.

See also:

macro

macro

The Twital instruction for Twig `macro` tag is `t:macro` node.

To declare a macro inside Twital, the syntax is:

```
<t:macro name="input" args="value, type, size">
    <input type="{{ type|default('text') }}" name="{{ name }}" value="{{ value|e }}"␣
↪size="{{ size|default(20) }}" />
</t:macro>
```

To use a macro inside your Twital template, take a look at the :doc:`import<../tags/import>` attribute.

Note: For more information about the `macro` tag, please refer to Twig official documentation.

use

The Twital instruction for Twig `use` tag is `t:use` node.

This is a fature that allows a/the horizontal reuse of templates. To learn more about it, you can read the official documentation.

Let's see how does it work:

```
<t:use from="bars.html"/>

<t:block name="sidebar">
    ...
</t:block>
```

You can create some aliases for block inside "used" template to avoid name conflicting:

```
<t:extends from="layout.html.twig">
    <t:use from="bars.html" aliases="sidebar as sidebar_original, footer as old_footer
↪"/>

    <t:block name="sidebar">
        {{ block('sidebar_original') }}
    </t:block>
</t:extends>
```

Note: For more information about the `use` tag, please refer to Twig official documentation.

sandbox

The Twital instruction for Twig `import` tag is `t:sandbox` node or the `t:sandbox` attribute.

The `sandbox` tag can be used to enable the sandboxing mode for an included template, when sandboxing is not enabled globally for the Twig environment:

```
<t:sandbox>
    {% include 'user.html' %}
</t:sandbox>

<div t:sandbox="">
    {% include 'user.html' %}
</div>
```

Note: For more information about the `sandbox` tag, please refer to Twig official documentation.

autoescape

The Twital instruction for Twig `autoescape` tag is `t:autoescape` attribute.

Whether automatic escaping is enabled or not, you can mark a section of a template to be escaped or not by using the `autoescape` tag.

To see how to use it, take a look at this example:

```
<div t:autoescape="true">
    Everything will be automatically escaped in this block
    using the HTML strategy
</div>

<div t:autoescape="html">
    Everything will be automatically escaped in this block
    using the HTML strategy
</div>

<div t:autoescape="js">
    Everything will be automatically escaped in this block
    using the js escaping strategy
</div>
```

```
<div t:autoescape="false">
    Everything will be outputted as is in this block
</div>
```

When automatic escaping is enabled, everything is escaped by default, except for values explicitly marked as safe. Those can be marked in the template by using the Twig `raw` filter:

```
<div t:autoescape="false">
    {{ safe_value|raw }}
</div>
```

capture

This attribute acts as a `set` tag and allows to 'capture' chunks of text into a variable:

```
<div id="pagination" t:capture="foo">
    ... any content ..
</div>
```

All contents inside "pagination" div will be captured and saved inside a variable named *foo*.

Note: For more information about the `set` tag, please refer to Twig official documentation.

filter

The Twital instruction for Twig `filter` tag is `t:filter` attribute.

To see how to use it, take a look at this example:

```
<div t:filter="upper">
    This text becomes uppercase
</div>

<div t:filter="upper|escape">
    This text becomes uppercase
</div>
```

Note: To learn more about the *filter* tab, you can read the Twig official documentation.

spaceless

The Twital instruction for Twig `spaceless` tag is `t:spaceless` node or the `t:spaceless` attribute.

```
<t:spaceless>
    {% include 'user.html' %}
</t:spaceless>

<div t:spaceless="">
    {% include 'user.html' %}
</div>
```

Note: For more information about the `spaceless` tag, please refer to Twig official documentation.

omit

This attribute asks the Twital parser to ignore the elements' open and close tag, its content will still be evaluated.

```
<a href="/private" t:omit="false">
    {{ username }}
</a>

<t:omit>
    {{ username }}
</t:omit>
```

This attribute is useful when you want to create element optionally, e.g. hide a link if certain condition is met.

attr

Twital allows you to create HTML/XML attributes in a very simple way. You do not have to mess up with control structures inside HTML tags.

Let's see how does it work:

```
<div t:attr=" condition ? class='header'">
    My Company
</div>
```

Here we add conditionally an attribute based on the value of the *condition* expression.

You can use any Twig test expression as **condition** and **attribute value**, but the attribute name must be a litteral.

```
<div t:attr="
    users | length ? class='header'|upper ,
    item in array ? class=item">
    Here wins the last class that condition will be evaluated to true.
</div>
```

When not needed, you can omit the condition instruction.

```
<div t:attr="class='row'">
     Class will be "row"
</div>
```

Tip: *attr-append*

To set an HTML5 boolean attribute, just use booleans as `true` or `false`.

```
<option t:attr="selected=true">
    My Company
</option>
```

The previous template will be rendered as:

```
<option selected>
    My Company
</option>
```

Note: Since XML does not have the concept of "boolean attributes", this feature may break your output if you are using XML.

To to remove and already defined attribute, use `false` as attribute value

```
<div class="foo" t:attr="class=false">
    My Company
</div>
```

The previous template will be rendered as:

```
<div>
    My Company
</div>
```

attr-append

Twital allows you to create HTML/XML attributes in a very simple way.

t:attr-append is a different version of *t:attr*: it allows to append content to existing attributes instead of replacing it.

```
<div class="row" t:attr-append=" condition ? class=' even'">
     class will be "row even" if 'i' is odd.
</div>
```

In the same way of *t:attr*, *condition* and the value of attribute can be any valid Twig expression.

```
<div class="row"
    t:attr-append=" i mod 2 ? class=' even'|upper">
     class will be "row EVEN" if 'i' is odd.
</div>
```

When not needed, you can omit the condition instruction.

```
<div class="row" t:attr-append=" class=' even'">
     Class will be "row even"
</div>
```

content

This attribute allows to replace the content of a note with the content of a variable.

Suppose to have a variable `foo` with a value `My name is John` and the following template:

```
<div id="pagination" t:content="foo">
    This <b>content</b> will be removed
</div>
```

The output will be:

```
<div id="pagination">My name is John</div>
```

This can be useful to put come "test" content in your templates that will have a nice aspect on WYSIWYG editors, but at runtime will be replaced by real data coming from variables.

replace

This attribute acts in a similar way to `content` attribute, instead of replacing the content of a node, will replace the node itself.

Suppose to have a variable `foo` with a value `My name is John` and the following template:

```
<div id="pagination" t:replace="foo">
    This <b>content</b> will be removed
</div>
```

The output will be:

```
My name is John
```

This can be useful to put come "test" content in your templates that will have a nice aspect on WYSIWYG editors, but at runtime will be replaced by real data coming from variables.

Twital for Template Designers

This document gives you an overview on Twital principles: how to write a template and what to bear in mind for making it work well.

Introduction

A template is simply a text file. Twital can generate any HTML/XML format.

To make it work, your templates must match the configured file extension.

By default, Twital compiles only templates whose name ends with `.twital.xml`, `.twital.html`, `.twital.xhtml` (using respectively XML, HTML5 and XHTML rules to format the output).

A Twital template is basically a Twig template that takes advantage of the natural HTML/XML tree structure (avoiding redundant control flow instructions). All expressions are completely Twig compatible; control flow structures (Twig calls them *tags*) are just replaced by some Twital *tags* or *attributes*.

Here is a minimal template that illustrates a few basics:

```
<!DOCTYPE html>
<html>
    <head>
        <title>My Webpage</title>
    </head>
    <body>
        <ul id="navigation">
            <li t:for="item in navigation">
                <a href="{{ item.href }}">{{ item.caption }}</a>
            </li>
        </ul>
```

```
        <h1>My Webpage</h1>
        {{ a_variable }}
    </body>
</html>
```

Tip: See *here* how to use specific output formats as XML or XHTML and HTML5.

IDEs Integration

Any IDE that supports Twig syntax highlighting and auto-completion should be configured to support Twital.

Here you can find a list of IDEs that support Twig/Twital

Variables

To print the content of variables, you can use exactly the same Twig syntax, using Twig functions, filters etc.

```
{{ foo.bar }}
{{ foo['bar'] }}
{{ attribute(foo, 'data-foo') }}
```

Note: To learn more about Twig variables, you can read the Twig official documentation

Setting Variables

The `t:set` attribute acts in the same way as the Twig's `set` tag and allows you to set a variable from a template.

```
<div t:set="name = 'Tom'">
    Hello {{ tom }}
</div>

<t:omit t:set="numbers = [1,2], items = {'item':'one'}"/>
{# t:omit tag will not be output-ed, but t:set will work #}
```

Note: To learn more about Twig `set`, you can read the Twig official documentation

Filters

You can use all Twig filters directly into Twital. Here is just an example:

```
{{ name|striptags|title }}
{{ list|join(', ') }}
```

You can also use the Twital attribute `t:filter` to filter the content of an element.

```
<div t:filter="upper">
    This text becomes uppercase
</div>
```

Note: To learn more about Twig filters, you can read the Twig official documentation

Functions

You can use all Twig functions directly from Twital.

For instance, the `range` function returns a list containing an arithmetic progression of integers:

```
<div t:for="i in range(0, 3)">
    {{ i }},
</div>
```

Note: To learn more about Twig filters, you can read the Twig official documentation

Control Structure

Almost all Twig control structures have a Twital equivalent node or attribute.

For example, to display a list of users, provided in a variable called `users`, use the *for* attribute:

```
<h1>Members</h1>
<ul>
    <li t:for="user in users">
        {{ user.username|e }}
    </li>
</ul>
```

The *if* attribute can be used to test an expression:

```
<ul t:if="users|length">
    <li t:for="user in users">
        {{ user.username|e }}
    </li>
</ul>
```

Go to the *tags* page to learn more about the built-in attributes and nodes.

To learn more about Twig control structures, you can read the Twig official documentation

Attributes

To create HTML/XML attributes, you do not have to mess up HTML tags with control structures. Twital makes things really easy!

Take a look at the following example:

```
<div t:attr=" condition ? class='header'">
    My Company
</div>
```

Using the *t:attr* attribute, you can conditionally add an attribute depending on the value of the `condition` expression.

You can use any Twig expression as a condition or attribute value. The attribute name must be a literal.

```
<div t:attr="
    users | length ? class='header'|upper ,
    item in array ? class=item">
    Here wins the last condition, which will be evaluated as true.
</div>
```

You can also append some content to existing attributes using the *t:attr-append*.

```
<div class="row"
    t:attr-append=" i mod 2 ? class=' even'">
     class will be "row even" if 'i' is odd.
</div>
```

If not needed, you can omit the condition instruction.

```
<div t:attr="class='row'" t:attr-append=" class=' even'">
     Class will be "row even"
</div>
```

To remove an attribute:

```
<div t:attr=" condition ? class=null">
     Class will be "row even"
</div>
```

Comments

To comment-out part of a line in a template, you can use the Twig comment syntax `{# ... #}`.

Including other Templates

The *include* tag is useful for including a template and returning the rendered content of that template into the current one:

```
<t:include from="sidebar.html"/>
```

Inclusions work exactly as in Twig.

Note: To learn more about Twig inclusion techniques, you can read the Twig official documentation

Template Inheritance

Twital's template inheritance is almost identical to Twig. Twital adds just some features useful to define new blocks.

Here we define a base template, `base.html`, which defines a simple HTML skeleton document that you might use for a simple two-column page:

```
<!DOCTYPE html>
<html>
    <head t:block="head">
        <link rel="stylesheet" href="style.css" />
        <title t:block="title">My Webpage</title>
    </head>
    <body>
        <div id="content" t:block="content">
        </div>
        <div id="footer" t:block="footer">
            &copy; Copyright 2011 by <a href="http://domain.invalid/">you</a>.
        </div>
    </body>
</html>
```

In this example, the *t:block* attributes define four blocks that child templates can fill in. All the `t:block` attributes tell the template engine that a child template may override those portions of the template.

A child template might look like this:

```
<t:extends from="base.html">

    <t:block name="title">Index</t:block>

    <t:block name="head">
        {{ parent() }}
        <style type="text/css">
            .important { color: #336699; }
        </style>
    </t:block>

    <t:block name="content">
        <h1>Index</h1>
        <p class="important">
            Welcome to my awesome homepage.
        </p>
    </t:block>

</t:extends>
```

The *t:extends* node tells the template engine that the template "extends" another template. When the template system evaluates the template, it first locates the parent.

The *extends* tag should be the first tag in the template.

Note that, since the child template does not define the `footer` block, the value from the parent template is used instead.

To render the contents of the parent block, use the parent Twig function. The following template gives back the results of the parent block:

```
<t:block name="sidebar">
    <h3>Table Of Contents</h3>
    ...
    {{ parent() }}
</t:block>
```

Tip: The documentation page for the *extends* tag describes more advanced features like block nesting, scope, dynamic inheritance, and conditional inheritance.

Note: To learn more about Twig inheritance, you can read the Twig official documentation

Macros

Twital also supports Twig macros. A macro is defined via the *macro* tag.

Note: To learn more about Twig macros, you can read the Twig official documentation

Expressions, Literals and Operators

All expressions, literals and operators that can be used with Twig, can also be used with Twital.

Note: Pay attention to HTML/XML escaping rules (eg: < or > inside attributes).

Whitespace Control

Twital will try to respect almost all whitespaces that you type. To remove whitespaces between HTML tags, you can use the `t:spaceless` attribute:

```
<div t:spaceless="">
    <strong>foo bar</strong>
</div>

{# output will be <div><strong>foo bar</strong></div> #}
```

Twital behaves the same as Twig in whitespaces handling.

Extensions

Twital can be easily extended. To learn how to create your own extension, you can read the Extending Twital chapter.

Twital for Developers

This chapter describes the PHP API to Twital and not the template language. It is mostly aimed to developers who want to integrate Twital in their projects.

Basics

Twital is a Twig Loader that pre-compiles some templates before sending them back to Twig, which compiles and runs the templates.

The first step to using Twital is to configure a valid Twig instance. Later, we can configure the Twital object.

```
<?php
use Goetas\Twital\TwitalLoader;

$loader = new Twig_Loader_Filesystem('/path/to/templates');
$twitalLoader = new TwitalLoader($loader);

$twig = new Twig_Environment($twitalLoader, array(
    'cache' => '/path/to/compilation_cache',
));
```

By default, Twital compiles only templates whose name ends with *.twital.xml*, *.twital.html*, *.twital.xhtml* (by using the right source adapter). If you want to change it, adding more supported file formats, you can do something like this:

```
<?php

$twital = new TwitalLoader($loader);
$twital->addSourceAdapter('/\.wsdl$/', new XMLAdapter()); // handle .wsdl files as XML
$twital->addSourceAdapter('/\.htm$/', new HTML5Adapter()); // handle .htm files as␣
↪HTML5
```

Note: Built in adapters are: *XMLAdapter*, *XHTMLAdapter* and *HTML5Adpater*.

Note: To learn more about adapters, you can read the dedicated chapter :ref‘‘Creating a SourceAdapter‘‘.

Finally, to render a template with some variables, simply call the `render()` method on Twig instance:

```
<?php
echo $twig->render('index.twital.html', array('the' => 'variables', 'go' => 'here'));
```

How does Twital work?

Twital uses Twig to render templates, but before passing a template to Twig, Twital pre-compiles it in its own way.

The rendering of a template can be summarized into the following steps:

- **Load** the template (done by Twig): if the template has already been compiled, Twig loads it and goes to the *evaluation* step. Otherwise:
  - A *SourceAdapter* is chosen (from a set of configured adapters);
  - The **compiler.pre_load** event is fired; Here, listeners can transform the template source code before DOM loading;
  - The *SourceAdapter* will *load* the source code into a valid DOMDocument object;
  - The **compiler.post_load** event is fired.
  - The compiler transforms recognized attributes and nodes into the relative Twig code;

  - The **compiler.pre_dump** event is fired.
  - The *SourceAdapter* will *dump* the compiled *DOMDocument* into Twig source code;
  - The **compiler.post_dump** event is fired. Here, listeners can perform some non-DOM transformations to the new template source code;
  - Twital passes the final source code to Twig (Finally Twig compiles the Twig source code into PHP code)
- **Evaluate** the template: Twig calls the `display()` method of the compiled template by passing a context.

Extending Twital

As Twig, Twital is very extensible and you can hook into it. The best way to extend Twital is to create your own "extension" and provide your functionalities.

Creating a *SourceAdapter*

Source adapters adapt a resource representation (usually a file or a string) to something that can be converted into a PHP DOMDocument object. Note that, the same object has to be "re-adapted" into its original representation.

If you want to provide a source adapter, there is no need to create an extension; you can simply implement the `Goetas\Twital\SourceAdapter` interface and use it.

To enable an adapter, you have to add it to Twital's loader instance by using the `addSourceAdapter()` method:

```
<?php
use Goetas\Twital\TwitalLoader;

$twital = new TwitalLoader($fileLoader);
$twital->addSourceAdapter('/.*.xml$/i', new MyXMLAdapter());
```

A "naive" implementation of *MyXMLAdapter* can be:

```
<?php
use Goetas\Twital\SourceAdapter;
use Goetas\Twital\Template;

class MyXMLAdapter implements SourceAdapter
{
    public function load($source)
    {
        $dom = new \DOMDocument('1.0', 'UTF-8');
        $someMetadata = null; // you can also extract some metadata from original␣
↪source

        return new Template($dom, $someMetadata);
    }

    public function dump(Template $template)
    {
        $metedata = $template->getMetadata();
        $dom = $template->getDocument();

        return $dom->saveXML();
    }
}
```

- As you can see, `load` takes a string (containing the Twital template source code), and returns a `Goetas\Twital\Template` object.
- `Goetas\Twital\Template` is an object that requires a DOMDocument as first argument and a generic variable as second argument (useful to hold some metadata extracted from the original source, which can be used later during the "dump" phase).
- The `dump` method takes a `Goetas\Twital\Template` instance and returns a string. The returned string contains the template source code that will be passed to Twig.

Creating an *Extension*

An extension is simply a container of functionalities that can be added to Twital. The functionalities are node parsers, attribute parses and generic event listeners.

To create an extension, you have to implement the `Goetas\Twital\Extension` interface or extend the *GoetasTwitalExtensionAbstractExtension* class.

This is the `Goetas\Twital\Extension` interface:

To enable your extensions, you have to add them to your Twital instance by using the `Goetas\Twital\Twital::addExtension()` method:

```
<?php
use Goetas\Twital\Twital;
use Goetas\Twital\TwitalLoader;

$twital = new Twital($twig);
$twital->addExtension(new MyNewCustomExtension());

$fsLoader = new Twig_Loader_Filesystem('/path/to/templates');
$twitalLoader = new TwitalLoader($fsLoader, $twital);
```

Tip: The bundled extensions are great examples of how extensions work.

Note: In some special cases you may need to create a Twig extension instead of a Twital one. To learn how to create a Twig extension, you can read the Twig official documentation

Creating a *Node* parser

Node parsers are aimed at handling any custom XML/HTML tag.

Suppose that you want to create an extension to handle a tag `<my:hello>` that simply prints *"Hello {name}"*:

```
<div class="red" xmlns:my="http://www.example.com/namespace">
    <my:hello name="John"/>
</div>
```

First, you have to create your node parser, which handles this "new" tag. To do this, you have to implement the `Goetas\Twital\Node` interface.

The `HelloNode` class can be something like this:

```php
<?php
use Goetas\Twital\Node;
use Goetas\Twital\Compiler;

class HelloNode implements Node
{
    function visit(\DOMElement $node, Compiler $twital)
    {
        $helloNode = $node->ownerDocument->createTextNode("hello");
        $nameNode = $twital->createPrintNode(
           $node->ownerDocument,
           "'".$node->getAttribute("name")."'"
        );

        $node->parentNode->replaceChild($nameNode, $node);
        $node->parentNode->insertBefore($helloNode, $nameNode);
    }
}
```

Let's take a look at the `Goetas\Twital\Node::visit` method signature:

- `$node` gets the **'DOMElement'_** node of your `my:hello` tag;
- `$twital` gets the Twital compiler;
- No return value for the `visit` method is required.

The aim of the `Goetas\Twital\Node::visit` method is to transform the Twital template representation into the Twig template syntax.

Tip: `$compiler->applyTemplatesToChilds()`, `$compiler->applyTemplates()` or `$compiler->applyTemplatesToAttributes()` can be very useful when you need to process recursively the content of a node.

Finally, you have to create an extension that ships your node parser.

```php
<?php
class MyExtension extends AbstractExtension
{
    public function getNodes()
    {
        return array(
            'http://www.example.com/namespace'=>array(
                'hello' => new HelloNode()
            )
        );
    }
}
```

As you can see, the `getNodes` method has to return a two-level hash.

- The first level is the node namespace;
- The second level is the node name.

Of course, an extension can ship nodes that work with multiple namespaces.

Tip: To make the `xmlns:my` declaration optional, you can also use the event listener as

`Goetas\Twital\EventSubscriber\CustomNamespaceRawSubscriber`.

Creating an *Attribute* parser

An attribute parser aims at handling custom XML/HTML attributes.

Suppose that we want to create an extension to handle an attribute that simply appends some text inside a node, removing its original content.

```
<div class="red" xmlns:my="http://www.example.com/namespace">
    <p my:replace="rawHtmlVar">
        This text will be replaced with the content of the "rawHtmlVar" variable.
    </p>
</div>
```

To add your attribute parser, first you have to implement the `Goetas\Twital\Attribute` interface.

The `HelloAttribute` class can be something like this:

```
<?php
class HelloAttribute implements Attribute
{
    function visit(\DOMAttr $attr, Compiler $twital)
    {
        $printNode = $twital->createPrintNode($attr->ownerNode->ownerDocument, $attr.
↪" | raw");

        $attr->ownerNode->appendChild($printNode);
        $node->parentNode->insertBefore($helloNode, $nameNode);

        return Attribute::STOP_NODE;
    }
}
```

Let's take a look at the `Goetas\Twital\Attribute::visit` method:

- `$attr` gets the *DOMAttr* node of your attribute;
- `$twital` gets the Twital compiler.

The `visit` method has to transform the custom attribute into a valid Twig code.

The `visit` method can also return one of the following constants:

- `Attribute::STOP_NODE`: instructs the compiler to jump to the next node (go to next sibling), stopping the processing of possible node childs;
- `Attribute::STOP_ATTRIBUTE`: instructs the compiler to stop processing attributes of the current node (continue normally with child and sibling nodes).

Finally, you have to create an extension that ships your attribute parser.

```
<?php
class MyExtension extends AbstractExtension
{
  public function getAttributes()
  {
     return array(
         'http://www.example.com/namespace'=>array(
```

```
            'replace' => new HelloAttribute()
        )
    );
  }
}
```

As you can see, the `getAttributes` method has to return a two-level hash. - The first level is the attribute namespace; - The second level is the attribute name.

Of course, an extension can ship nodes that work with multiple namespaces.

Tip: To make the `xmlns:my` declaration optional, you can also use the event listener as `Goetas\Twital\EventSubscriber\CustomNamespaceRawSubscriber`.

Event Listeners

Another convenient way to hook into Twital is to create an event listener.

The possible entry points for listeners are:

- **compiler.pre_load**, fired before the source has been passed to the source adapter;
- **compiler.post_load**, fired after the source has been loaded into a DOMDocument;
- **compiler.pre_dump**, fired before the DOMDocument has been passed to the source adapter for the dumping phase;
- **compiler.post_dump**, fired after the DOMDocument has been dumped into a string by the source adapter.

A valid listener must implement the `Symfony\Component\EventDispatcher\EventSubscriberInterface` interface.

This is an example for a valid listener:

```
<?php
class MySubscriber implements EventSubscriberInterface {
    public static function getSubscribedEvents() {
        return array(
            'compiler.post_dump' => 'modifySource'
            'compiler.pre_dump' => 'modifyDOM'

            'compiler.post_load' => 'modifyDOM',
            'compiler.pre_load' => 'modifySource'
        );
    }
    public function modifyDOM(TemplateEvent $event) {
        $event->getTemplate(); // do something with template (returns a Template␣
↪instance)
    }
    public function modifySource(SourceEvent $event) {
        $event->getTemplate(); // do something with template (returns a string)
    }
}
```

Event `compiler.pre_load`

This event is fired just before a *SourceAdapter* tries to load the source code into a DOMDocument. Here you can modify the source code, adapting it for a source adapter.

Here an example:

```
<?php
class MySubscriber implements EventSubscriberInterface
{
    public static function getSubscribedEvents(){
        return array(
            'compiler.pre_load' => 'modifySource'
        );
    }
    public function modifySource(SourceEvent $event) {
        $str = $event->getTemplate();
        $str = str_replace(" ", " ", $str);

        $event->setTemplate($str);
    }
}
```

Tip: Take a look at `Goetas\Twital\EventSubscriber\CustomNamespaceRawSubscriber` to see what can be done using this event.

Event `compiler.post_load`

This event is fired just after a `Goetas\Twital\SourceAdapeter::load()` call. Here you can modify the DOMDocument object; it is a good point where to apply modifications that can't be done by node parsers. You can also add nodes that will be parsed by Twital (eg: `t:if` attribute, `t:include` nodes, etc).

Here an example:

```
<?php
class MySubscriber implements EventSubscriberInterface {
    public static function getSubscribedEvents() {
        return array(
            'compiler.post_load' => 'modifyDOM'
        );
    }
    public function modifyDOM(TemplateEvent $event) {
        $template = $event->getTemplate();
        $dom = $template->getTemplate();

        $nodes = $dom->getElementsByTagName('mynode');

        // do something with $nodes
    }
}
```

Tip: Take a look at `Goetas\Twital\EventSubscriber\CustomNamespaceSubscriber` to see what can be done using this event.

Event `compiler.pre_dump`

This event is fired when the Twital compilation process ends. It is similar to the `compiler.post_load` event, but you can not add elements that need to be parsed by Twital.

Here an example:

```
<?php
class MySubscriber implements EventSubscriberInterface
{
    public static function getSubscribedEvents(){
        return array(
            'compiler.pre_dump' => 'modifyDOM'
        );
    }
    public function modifyDOM(TemplateEvent $event) {
        $template = $event->getTemplate();
        $dom = $template->getTemplate();

        $body = $dom->getElementsByTagName('body')->item(0);
        // do something with body node...
    }
}
```

Event `compiler.post_dump`

This event is fired just after the `Goetas\Twital\SourceAdapeter::dump()` call. Here you can modify the final source code, which will be passed to Twig.

Here an example:

```
<?php
class MySubscriber implements EventSubscriberInterface {
    public static function getSubscribedEvents() {
        return array(
            'compiler.post_dump' => 'modifySource'
        );
    }
    public function modifySource(SourceEvent $event) {
        $str = $event->getTemplate();
        $str.=" {# generated by Twital #}";

        $event->setTemplate($str);
    }
}
```

Tip: Take a look at `Goetas\Twital\EventSubscriber\DOMMessSubscriber` to see what can be done using this event.

Ship your listeners

If you have created your listeners, add them to Tiwtal. To do this, you have to create an extension that ships your listeners.

```php
<?php
class MyExtension extends AbstractExtension
{
  public function getSubscribers()
  {
     return array(
         new MySubscriber(),
         new MyNewSubscriber()
     );
  }
}
```

Common mistakes and tricks

Since Twital internally uses XML, you need to pay attention to some aspects while writing a template. All templates must be XML valid (some exceptions are allowed...).

- All templates must have **one** root node. When needed, you can use the *t:omit* node to enclose other nodes.

```
<t:omit>
    <div>one</div>
    <div>two</div>
</t:omit>
```

- A template must be well formatted (opening and closing nodes, entities, DTD, etc...). Some aspects as namespaces, HTML5 & HTML entities, non-self closing tags can be "repaired", but it is recommended to be closer to XML as much as possible.

  The example below lacks the *br* self closing slash, but using the HTML5 source adapter it can be omitted.

```
<div>
    <br>
</div>
```

- The usage of *&* must follow XML syntax rules. .. code-block:: html

  ** ** & <!– to output "&" you have to write "&" –> < <!– to output "<" you have to write "<" –> > <!– to output ">" you have to write ">" –>

  <!– you can use all numeric entities –>

  <!– you should not use named entities (€)–>

- To be compatible with all browsers, the use of the *script* tag should be combined with *CDATA* sections and script comments.

```
<script>
//<![CDATA[
if ( 1 > 2 && 2 < 0){
    alert(' ok ')
}
//]]>
</script>
<style>
/*<![CDATA[*/
```

```
head {
    color: red;
}
/*]]>*/
</style>
```

Symfony2 Users

If you are a Symfony2 user, the most convenient way to integrate Twital into your project is using the TwitalBundle.

The bundle integrates all most common Synfony functionalities as Assetic, Forms, Translations etc.

CHAPTER 4

Contributing

This is an open source project: contributions are welcome. If your are interested, you can contribute to documentation, source code, test suite or anything else!

To start contributing right now, go to https://github.com/goetas/twital and fork it!

To improve your contributing experience, you can take a look into https://github.com/goetas/twital/blob/master/CONTRIBUTING.md inside the root directory of Twital GIT repository.

CHAPTER 5

Symfony2 Users

If you are a Symfony2 user, you can add Twital to your project using the TwitalBundle.

The bundle integrates all most common functionalities as Assetic, Forms, Translations, Routing, etc.

CHAPTER 6

Note

I'm sorry for the *terrible* english fluency used inside the documentation, I'm trying to improve it. Pull Requests are welcome.