The Twig Book

generated on June 21, 2012

The Twig Book

Contents at a Glance

Chapter 1
Introduction

This is the documentation for Twig, the flexible, fast, and secure template engine for PHP.

If you have any exposure to other text-based template languages, such as Smarty, Django, or Jinja, you should feel right at home with Twig. It's both designer and developer friendly by sticking to PHP's principles and adding functionality useful for templating environments.

The key-features are...

- *Fast*: Twig compiles templates down to plain optimized PHP code. The overhead compared to regular PHP code was reduced to the very minimum.
- *Secure*: Twig has a sandbox mode to evaluate untrusted template code. This allows Twig to be used as a template language for applications where users may modify the template design.
- *Flexible*: Twig is powered by a flexible lexer and parser. This allows the developer to define its own custom tags and filters, and create its own DSL.

Prerequisites

Twig needs at least **PHP 5.2.4** to run.

Installation

You have multiple ways to install Twig. If you are unsure what to do, go with the tarball.

Installing from the tarball release

1. Download the most recent tarball from the *download page*[1]
2. Unpack the tarball
3. Move the files somewhere in your project

1. `https://github.com/fabpot/Twig/tags`

Installing the development version

1. Install Subversion or Git
2. For Git: `git clone git://github.com/fabpot/Twig.git`
3. For Subversion: `svn co http://svn.twig-project.org/trunk/ twig`

Installing the PEAR package

1. Install PEAR
2. `pear channel-discover pear.twig-project.org`
3. `pear install twig/Twig` (or `pear install twig/Twig-beta`)

Installing via Composer

1. Install composer in your project:

Listing 1-1
```
curl -s http://getcomposer.org/installer | php
```

2. Create a `composer.json` file in your project root:

Listing 1-2
```
{
    "require": {
        "twig/twig": "1.*"
    }
}
```

3. Install via composer

Listing 1-3
```
php composer.phar install
```

If you want to learn more about Composer, the `composer.json` file syntax and its usage, you can read the *online documentation*[2].

Installing the C extension

New in version 1.4: The C extension was added in Twig 1.4.

Twig comes with a C extension that enhances the performance of the Twig runtime engine. You can install it like any other PHP extension:

Listing 1-4
```
$ cd ext/twig
$ phpize
$ ./configure
$ make
$ make install
```

Finally, enable the extension in your `php.ini` configuration file:

Listing 1-5
```
extension=twig.so
```

2. `http://getcomposer.org/doc`

And from now on, Twig will automatically compile your templates to take advantage of the C extension. Note that this extension does not replace the PHP code but only provides an optimized version of the `Twig_Template::getAttribute()` method.

On Windows, you can also simply download and install a *pre-build DLL*[3].

Basic API Usage

This section gives you a brief introduction to the PHP API for Twig.

The first step to use Twig is to register its autoloader:

Listing 1-6

```
require_once '/path/to/lib/Twig/Autoloader.php';
Twig_Autoloader::register();
```

Replace the `/path/to/lib/` path with the path you used for Twig installation.

Twig follows the PEAR convention names for its classes, which means you can easily integrate Twig classes loading in your own autoloader.

Listing 1-7

```
$loader = new Twig_Loader_String();
$twig = new Twig_Environment($loader);

echo $twig->render('Hello {{ name }}!', array('name' => 'Fabien'));
```

Twig uses a loader (`Twig_Loader_String`) to locate templates, and an environment (`Twig_Environment`) to store the configuration.

The `render()` method loads the template passed as a first argument and renders it with the variables passed as a second argument.

As templates are generally stored on the filesystem, Twig also comes with a filesystem loader:

Listing 1-8

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

echo $twig->render('index.html', array('name' => 'Fabien'));
```

3. https://github.com/stealth35/stealth35.github.com/downloads

Chapter 2
Twig for Template Designers

This document describes the syntax and semantics of the template engine and will be most useful as reference to those creating Twig templates.

Synopsis

A template is simply a text file. It can generate any text-based format (HTML, XML, CSV, LaTeX, etc.). It doesn't have a specific extension, `.html` or `.xml` are just fine.

A template contains **variables** or **expressions**, which get replaced with values when the template is evaluated, and **tags**, which control the logic of the template.

Below is a minimal template that illustrates a few basics. We will cover the details later on:

Listing 2-1

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

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

There are two kinds of delimiters: `{% ... %}` and `{{ ... }}`. The first one is used to execute statements such as for-loops, the latter prints the result of an expression to the template.

IDEs Integration

Many IDEs support syntax highlighting and auto-completion for Twig:

- *Textmate* via the *Twig bundle*[1]
- *Vim* via the *Jinja syntax plugin*[2]
- *Netbeans* via the *Twig syntax plugin*[3]
- *PhpStorm* (native as of 2.1)
- *Eclipse* via the *Twig plugin*[4]
- *Sublime Text* via the *Twig bundle*[5]
- *GtkSourceView* via the *Twig language definition*[6] (used by gedit and other projects)
- *Coda* and *SubEthaEdit* via the *Twig syntax mode*[7]
- *Coda 2* via the *other Twig syntax mode*[8]
- *Komodo* and *Komodo Edit* via the Django highlight/syntax check mode

Variables

The application passes variables to the templates you can mess around in the template. Variables may have attributes or elements on them you can access too. How a variable looks like heavily depends on the application providing those.

You can use a dot (`.`) to access attributes of a variable (methods or properties of a PHP object, or items of a PHP array), or the so-called "subscript" syntax (`[]`):

Listing 2-2

```
{{ foo.bar }}
{{ foo['bar'] }}
```

It's important to know that the curly braces are *not* part of the variable but the print statement. If you access variables inside tags don't put the braces around.

If a variable or attribute does not exist, you will get back a `null` value when the `strict_variables` option is set to `false`, otherwise Twig will throw an error (see *environment options*).

1. https://github.com/Anomareh/PHP-Twig.tmbundle
2. http://jinja.pocoo.org/2/documentation/integration
3. http://plugins.netbeans.org/plugin/37069/php-twig
4. https://github.com/pulseOO/Twig-Eclipse-Plugin
5. https://github.com/Anomareh/PHP-Twig.tmbundle
6. https://github.com/gabrielcorpse/gedit-twig-template-language
7. https://github.com/bobthecow/Twig-HTML.mode
8. https://github.com/muxx/Twig-HTML.mode

Implementation

For convenience sake `foo.bar` does the following things on the PHP layer:

- check if `foo` is an array and `bar` a valid element;
- if not, and if `foo` is an object, check that `bar` is a valid property;
- if not, and if `foo` is an object, check that `bar` is a valid method (even if `bar` is the constructor - use `__construct()` instead);
- if not, and if `foo` is an object, check that `getBar` is a valid method;
- if not, and if `foo` is an object, check that `isBar` is a valid method;
- if not, return a `null` value.

`foo['bar']` on the other hand only works with PHP arrays:

- check if `foo` is an array and `bar` a valid element;
- if not, return a `null` value.

If you want to get a dynamic attribute on a variable, use the *attribute* function instead.

Global Variables

The following variables are always available in templates:

- `_self`: references the current template;
- `_context`: references the current context;
- `_charset`: references the current charset.

Setting Variables

You can assign values to variables inside code blocks. Assignments use the *set* tag:

Listing 2-3

```
{% set foo = 'foo' %}
{% set foo = [1, 2] %}
{% set foo = {'foo': 'bar'} %}
```

Filters

Variables can be modified by **filters**. Filters are separated from the variable by a pipe symbol (|) and may have optional arguments in parentheses. Multiple filters can be chained. The output of one filter is applied to the next.

The following example removes all HTML tags from the `name` and title-cases it:

Listing 2-4

```
{{ name|striptags|title }}
```

Filters that accept arguments have parentheses around the arguments. This example will join a list by commas:

Listing 2-5

```
{{ list|join(', ') }}
```

To apply a filter on a section of code, wrap it with the *filter* tag:

```
{% filter upper %}
  This text becomes uppercase
{% endfilter %}
```

Listing 2-6

Go to the *filters* page to learn more about the built-in filters.

Functions

Functions can be called to generate content. Functions are called by their name followed by parentheses (`()`) and may have arguments.

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

```
{% for i in range(0, 3) %}
    {{ i }},
{% endfor %}
```

Listing 2-7

Go to the *functions* page to learn more about the built-in functions.

Control Structure

A control structure refers to all those things that control the flow of a program - conditionals (i.e. `if`/`elseif`/`else`), `for`-loops, as well as things like blocks. Control structures appear inside `{% ... %}` blocks.

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

```
<h1>Members</h1>
<ul>
    {% for user in users %}
        <li>{{ user.username|e }}</li>
    {% endfor %}
</ul>
```

Listing 2-8

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

```
{% if users|length > 0 %}
    <ul>
        {% for user in users %}
            <li>{{ user.username|e }}</li>
        {% endfor %}
    </ul>
{% endif %}
```

Listing 2-9

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

Comments

To comment-out part of a line in a template, use the comment syntax `{# ... #}`. This is useful for debugging or to add information for other template designers or yourself:

```
{# note: disabled template because we no longer use this
    {% for user in users %}
        ...
```

Listing 2-10

```
    {% endfor %}
#}
```

Including other Templates

The *include* tag is useful to include a template and return the rendered content of that template into the current one:

Listing 2-11
```
{% include 'sidebar.html' %}
```

Per default included templates are passed the current context.

The context that is passed to the included template includes variables defined in the template:

Listing 2-12
```
{% for box in boxes %}
    {% include "render_box.html" %}
{% endfor %}
```

The included template `render_box.html` is able to access `box`.

The filename of the template depends on the template loader. For instance, the `Twig_Loader_Filesystem` allows you to access other templates by giving the filename. You can access templates in subdirectories with a slash:

Listing 2-13
```
{% include "sections/articles/sidebar.html" %}
```

This behavior depends on the application embedding Twig.

Template Inheritance

The most powerful part of Twig is template inheritance. Template inheritance allows you to build a base "skeleton" template that contains all the common elements of your site and defines **blocks** that child templates can override.

Sounds complicated but is very basic. It's easier to understand it by starting with an example.

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

Listing 2-14
```
<!DOCTYPE html>
<html>
    <head>
        {% block head %}
            <link rel="stylesheet" href="style.css" />
            <title>{% block title %}{% endblock %} - My Webpage</title>
        {% endblock %}
    </head>
    <body>
        <div id="content">{% block content %}{% endblock %}</div>
        <div id="footer">
            {% block footer %}
                &copy; Copyright 2011 by <a href="http://domain.invalid/">you</a>.
            {% endblock %}
        </div>
    </body>
</html>
```

In this example, the *block* tags define four blocks that child templates can fill in. All the `block` tag does is to tell the template engine that a child template may override those portions of the template.

A child template might look like this:

Listing 2-15

```
{% extends "base.html" %}

{% block title %}Index{% endblock %}
{% block head %}
    {{ parent() }}
    <style type="text/css">
        .important { color: #336699; }
    </style>
{% endblock %}
{% block content %}
    <h1>Index</h1>
    <p class="important">
        Welcome on my awesome homepage.
    </p>
{% endblock %}
```

The *extends* tag is the key here. It tells the template engine that this template "extends" another template. When the template system evaluates this template, first it locates the parent. The extends tag should be the first tag in the template.

Note that since the child template doesn't define the `footer` block, the value from the parent template is used instead.

It's possible to render the contents of the parent block by using the *parent* function. This gives back the results of the parent block:

Listing 2-16

```
{% block sidebar %}
    <h3>Table Of Contents</h3>
    ...
    {{ parent() }}
{% endblock %}
```

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

Twig also supports multiple inheritance with the so called horizontal reuse with the help of the *use* tag. This is an advanced feature hardly ever needed in regular templates.

HTML Escaping

When generating HTML from templates, there's always a risk that a variable will include characters that affect the resulting HTML. There are two approaches: manually escaping each variable or automatically escaping everything by default.

Twig supports both, automatic escaping is enabled by default.

Automatic escaping is only supported if the *escaper* extension has been enabled (which is the default).

Working with Manual Escaping

If manual escaping is enabled it's **your** responsibility to escape variables if needed. What to escape? If you have a variable that *may* include any of the following chars (`>`, `<`, `&`, or `"`) you **have to** escape it unless the variable contains well-formed and trusted HTML. Escaping works by piping the variable through the *escape* or `e` filter:

Listing 2-17
```
{{ user.username|e }}
{{ user.username|e('js') }}
```

Working with Automatic Escaping

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:

Listing 2-18
```
{% autoescape true %}
    Everything will be automatically escaped in this block
{% endautoescape %}
```

Escaping

It is sometimes desirable or even necessary to have Twig ignore parts it would otherwise handle as variables or blocks. For example if the default syntax is used and you want to use `{{` as raw string in the template and not start a variable you have to use a trick.

The easiest way is to output the variable delimiter (`{{`) by using a variable expression:

Listing 2-19
```
{{ '{{' }}
```

For bigger sections it makes sense to mark a block *raw*.

Macros

Macros are comparable with functions in regular programming languages. They are useful to reuse often used HTML fragments to not repeat yourself.

A macro is defined via the *macro* tag. Here is a small example (subsequently called `forms.html`) of a macro that renders a form element:

Listing 2-20
```
{% macro input(name, value, type, size) %}
    <input type="{{ type|default('text') }}" name="{{ name }}" value="{{ value|e }}" size="{{
size|default(20) }}" />
{% endmacro %}
```

Macros can be defined in any template, and need to be "imported" via the *import* tag before being used:

Listing 2-21
```
{% import "forms.html" as forms %}

<p>{{ forms.input('username') }}</p>
```

Alternatively, you can import individual macro names from a template into the current namespace via the *from* tag and optionally alias them:

Listing 2-22

```
{% from 'forms.html' import input as input_field %}

<dl>
    <dt>Username</dt>
    <dd>{{ input_field('username') }}</dd>
    <dt>Password</dt>
    <dd>{{ input_field('password', '', 'password') }}</dd>
</dl>
```

Expressions

Twig allows expressions everywhere. These work very similar to regular PHP and even if you're not working with PHP you should feel comfortable with it.

The operator precedence is as follows, with the lowest-precedence operators listed first: `b-and`, `b-xor`, `b-or`, `or`, `and`, `==`, `!=`, `<`, `>`, `>=`, `<=`, `in`, `..`, `+`, `-`, `~`, `*`, `/`, `//`, `%`, `is`, and `**`.

Literals

New in version 1.5: Support for hash keys as names and expressions was added in Twig 1.5.

The simplest form of expressions are literals. Literals are representations for PHP types such as strings, numbers, and arrays. The following literals exist:

- `"Hello World"`: Everything between two double or single quotes is a string. They are useful whenever you need a string in the template (for example as arguments to function calls, filters or just to extend or include a template).
- `42` / `42.23`: Integers and floating point numbers are created by just writing the number down. If a dot is present the number is a float, otherwise an integer.
- `["foo", "bar"]`: Arrays are defined by a sequence of expressions separated by a comma (`,`) and wrapped with squared brackets (`[]`).
- `{"foo": "bar"}`: Hashes are defined by a list of keys and values separated by a comma (`,`) and wrapped with curly braces (`{}`):

  Listing 2-23

  ```
  {# keys as string #}
  { 'foo': 'foo', 'bar': 'bar' }

  {# keys as names (equivalent to the previous hash) -- as of Twig 1.5 #}
  { foo: 'foo', bar: 'bar' }

  {# keys as integer #}
  { 2: 'foo', 4: 'bar' }

  {# keys as expressions (the expression must be enclosed into parentheses) -- as of
  Twig 1.5 #}
  { (1 + 1): 'foo', (a ~ 'b'): 'bar' }
  ```

- `true` / `false`: `true` represents the true value, `false` represents the false value.
- `null`: `null` represents no specific value. This is the value returned when a variable does not exist. `none` is an alias for `null`.

Arrays and hashes can be nested:

Listing 2-24
```
{% set foo = [1, {"foo": "bar"}] %}
```

Math

Twig allows you to calculate with values. This is rarely useful in templates but exists for completeness' sake. The following operators are supported:

- `+`: Adds two objects together (the operands are casted to numbers). `{{ 1 + 1 }}` is `2`.
- `-`: Substracts the second number from the first one. `{{ 3 - 2 }}` is `1`.
- `/`: Divides two numbers. The returned value will be a floating point number. `{{ 1 / 2 }}` is `{{ 0.5 }}`.
- `%`: Calculates the remainder of an integer division. `{{ 11 % 7 }}` is `4`.
- `//`: Divides two numbers and returns the truncated integer result. `{{ 20 // 7 }}` is `2`.
- `*`: Multiplies the left operand with the right one. `{{ 2 * 2 }}` would return `4`.
- `**`: Raises the left operand to the power of the right operand. `{{ 2 ** 3 }}` would return `8`.

Logic

You can combine multiple expressions with the following operators:

- `and`: Returns true if the left and the right operands are both true.
- `or`: Returns true if the left or the right operand is true.
- `not`: Negates a statement.
- `(expr)`: Groups an expression.

Twig also support bitwise operators (`b-and`, `b-xor`, and `b-or`).

Comparisons

The following comparison operators are supported in any expression: `==`, `!=`, `<`, `>`, `>=`, and `<=`.

Containment Operator

The `in` operator performs containment test.

It returns `true` if the left operand is contained in the right:

Listing 2-25
```
{# returns true #}

{{ 1 in [1, 2, 3] }}

{{ 'cd' in 'abcde' }}
```

You can use this filter to perform a containment test on strings, arrays, or objects implementing the `Traversable` interface.

To perform a negative test, use the `not in` operator:

```
{% if 1 not in [1, 2, 3] %}

{# is equivalent to #}
{% if not (1 in [1, 2, 3]) %}
```

Listing 2-26

Test Operator

The `is` operator performs tests. Tests can be used to test a variable against a common expression. The right operand is name of the test:

```
{# find out if a variable is odd #}

{{ name is odd }}
```

Listing 2-27

Tests can accept arguments too:

```
{% if loop.index is divisibleby(3) %}
```

Listing 2-28

Tests can be negated by using the `is not` operator:

```
{% if loop.index is not divisibleby(3) %}

{# is equivalent to #}
{% if not (loop.index is divisibleby(3)) %}
```

Listing 2-29

Go to the *tests* page to learn more about the built-in tests.

Other Operators

The following operators are very useful but don't fit into any of the other categories:

- `..`: Creates a sequence based on the operand before and after the operator (this is just syntactic sugar for the *range* function).
- `|`: Applies a filter.
- `~`: Converts all operands into strings and concatenates them. `{{ "Hello " ~ name ~ "!" }}` would return (assuming `name` is `'John'`) `Hello John!`.
- `.`, `[]`: Gets an attribute of an object.
- `?:`: The PHP ternary operator: `{{ foo ? 'yes' : 'no' }}`

String Interpolation

New in version 1.5: String interpolation was added in Twig 1.5.

String interpolation (*#{expression}*) allows any valid expression to appear within a string. The result of evaluating that expression is inserted into the string:

```
{{ "foo #{bar} baz" }}
{{ "foo #{1 + 2} baz" }}
```

Listing 2-30

Whitespace Control

New in version 1.1: Tag level whitespace control was added in Twig 1.1.

The first newline after a template tag is removed automatically (like in PHP.) Whitespace is not further modified by the template engine, so each whitespace (spaces, tabs, newlines etc.) is returned unchanged.

Use the `spaceless` tag to remove whitespace *between HTML tags*:

Listing 2-31
```
{% spaceless %}
    <div>
        <strong>foo</strong>
    </div>
{% endspaceless %}

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

In addition to the spaceless tag you can also control whitespace on a per tag level. By using the whitespace control modifier on your tags, you can trim leading and or trailing whitespace:

Listing 2-32
```
{% set value = 'no spaces' %}
{#- No leading/trailing whitespace -#}
{%- if true -%}
    {{- value -}}
{%- endif -%}

{# output 'no spaces' #}
```

The above sample shows the default whitespace control modifier, and how you can use it to remove whitespace around tags. Trimming space will consume all whitespace for that side of the tag. It is possible to use whitespace trimming on one side of a tag:

Listing 2-33
```
{% set value = 'no spaces' %}
<li>    {{- value }}    </li>

{# outputs '<li>no spaces    </li>' #}
```

Extensions

Twig can be easily extended.

If you are looking for new tags, filters, or functions, have a look at the Twig official *extension repository*[9].

If you want to create your own, read the *Creating an Extension* chapter.

9. `http://github.com/fabpot/Twig-extensions`

Chapter 3
Twig for Developers

This chapter describes the API to Twig and not the template language. It will be most useful as reference to those implementing the template interface to the application and not those who are creating Twig templates.

Basics

Twig uses a central object called the **environment** (of class `Twig_Environment`). Instances of this class are used to store the configuration and extensions, and are used to load templates from the file system or other locations.

Most applications will create one `Twig_Environment` object on application initialization and use that to load templates. In some cases it's however useful to have multiple environments side by side, if different configurations are in use.

The simplest way to configure Twig to load templates for your application looks roughly like this:

Listing 3-1

```
require_once '/path/to/lib/Twig/Autoloader.php';
Twig_Autoloader::register();

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

This will create a template environment with the default settings and a loader that looks up the templates in the `/path/to/templates/` folder. Different loaders are available and you can also write your own if you want to load templates from a database or other resources.

Notice that the second argument of the environment is an array of options. The `cache` option is a compilation cache directory, where Twig caches the compiled templates to avoid the parsing phase for sub-sequent requests. It is very different from the cache you might want to add for the evaluated templates. For such a need, you can use any available PHP cache library.

To load a template from this environment you just have to call the `loadTemplate()` method which then returns a `Twig_Template` instance:

Listing 3-2
```
$template = $twig->loadTemplate('index.html');
```

To render the template with some variables, call the `render()` method:

Listing 3-3
```
echo $template->render(array('the' => 'variables', 'go' => 'here'));
```

The `display()` method is a shortcut to output the template directly.

You can also load and render the template in one fell swoop:

Listing 3-4
```
echo $twig->render('index.html', array('the' => 'variables', 'go' => 'here'));
```

Environment Options

When creating a new `Twig_Environment` instance, you can pass an array of options as the constructor second argument:

Listing 3-5
```
$twig = new Twig_Environment($loader, array('debug' => true));
```

The following options are available:

- `debug`: When set to `true`, the generated templates have a `__toString()` method that you can use to display the generated nodes (default to `false`).
- `charset`: The charset used by the templates (default to `utf-8`).
- `base_template_class`: The base template class to use for generated templates (default to `Twig_Template`).
- `cache`: An absolute path where to store the compiled templates, or `false` to disable caching (which is the default).
- `auto_reload`: When developing with Twig, it's useful to recompile the template whenever the source code changes. If you don't provide a value for the `auto_reload` option, it will be determined automatically based on the `debug` value.
- `strict_variables`: If set to `false`, Twig will silently ignore invalid variables (variables and or attributes/methods that do not exist) and replace them with a `null` value. When set to `true`, Twig throws an exception instead (default to `false`).
- `autoescape`: If set to `true`, auto-escaping will be enabled by default for all templates (default to `true`). As of Twig 1.8, you can set the escaping strategy to use (`html`, `js`, `false` to disable, or a PHP callback that takes the template "filename" and must return the escaping strategy to use -- the callback cannot be a function name to avoid collision with built-in escaping strategies).
- `optimizations`: A flag that indicates which optimizations to apply (default to `-1` -- all optimizations are enabled; set it to `0` to disable).

Loaders

Loaders are responsible for loading templates from a resource such as the file system.

Compilation Cache

All template loaders can cache the compiled templates on the filesystem for future reuse. It speeds up Twig a lot as templates are only compiled once; and the performance boost is even larger if you use a PHP accelerator such as APC. See the `cache` and `auto_reload` options of `Twig_Environment` above for more information.

Built-in Loaders

Here is a list of the built-in loaders Twig provides:

- `Twig_Loader_Filesystem`: Loads templates from the file system. This loader can find templates in folders on the file system and is the preferred way to load them:

  ```
  $loader = new Twig_Loader_Filesystem($templateDir);
  ```

  Listing 3-6

  It can also look for templates in an array of directories:

  ```
  $loader = new Twig_Loader_Filesystem(array($templateDir1, $templateDir2));
  ```

  Listing 3-7

  With such a configuration, Twig will first look for templates in `$templateDir1` and if they do not exist, it will fallback to look for them in the `$templateDir2`.

- `Twig_Loader_String`: Loads templates from a string. It's a dummy loader as you pass it the source code directly:

  ```
  $loader = new Twig_Loader_String();
  ```

  Listing 3-8

- `Twig_Loader_Array`: Loads a template from a PHP array. It's passed an array of strings bound to template names. This loader is useful for unit testing:

  ```
  $loader = new Twig_Loader_Array($templates);
  ```

  Listing 3-9

When using the `Array` or `String` loaders with a cache mechanism, you should know that a new cache key is generated each time a template content "changes" (the cache key being the source code of the template). If you don't want to see your cache grows out of control, you need to take care of clearing the old cache file by yourself.

Create your own Loader

All loaders implement the `Twig_LoaderInterface`:

```
interface Twig_LoaderInterface
{
    /**
     * Gets the source code of a template, given its name.
     *
     * @param  string $name string The name of the template to load
     *
     * @return string The template source code
     */
    function getSource($name);

    /**
     * Gets the cache key to use for the cache for a given template name.
     *
     * @param  string $name string The name of the template to load
```

Listing 3-10

```
     *
     * @return string The cache key
     */
    function getCacheKey($name);

    /**
     * Returns true if the template is still fresh.
     *
     * @param string    $name The template name
     * @param timestamp $time The last modification time of the cached template
     */
    function isFresh($name, $time);
}
```

As an example, here is how the built-in `Twig_Loader_String` reads:

Listing 3-11
```
class Twig_Loader_String implements Twig_LoaderInterface
{
    public function getSource($name)
    {
      return $name;
    }

    public function getCacheKey($name)
    {
      return $name;
    }

    public function isFresh($name, $time)
    {
      return false;
    }
}
```

The `isFresh()` method must return `true` if the current cached template is still fresh, given the last modification time, or `false` otherwise.

Using Extensions

Twig extensions are packages that add new features to Twig. Using an extension is as simple as using the `addExtension()` method:

Listing 3-12
```
$twig->addExtension(new Twig_Extension_Sandbox());
```

Twig comes bundled with the following extensions:

- *Twig_Extension_Core*: Defines all the core features of Twig.
- *Twig_Extension_Escaper*: Adds automatic output-escaping and the possibility to escape/unescape blocks of code.
- *Twig_Extension_Sandbox*: Adds a sandbox mode to the default Twig environment, making it safe to evaluated untrusted code.
- *Twig_Extension_Optimizer*: Optimizers the node tree before compilation.

The core, escaper, and optimizer extensions do not need to be added to the Twig environment, as they are registered by default. You can disable an already registered extension:

Listing 3-13
```
$twig->removeExtension('escaper');
```

Built-in Extensions

This section describes the features added by the built-in extensions.

Read the chapter about extending Twig to learn how to create your own extensions.

Core Extension

The `core` extension defines all the core features of Twig:

- Tags:
    - `for`
    - `if`
    - `extends`
    - `include`
    - `block`
    - `filter`
    - `macro`
    - `import`
    - `from`
    - `set`
    - `spaceless`
- Filters:
    - `date`
    - `format`
    - `replace`
    - `url_encode`
    - `json_encode`
    - `title`
    - `capitalize`
    - `upper`
    - `lower`
    - `striptags`
    - `join`
    - `reverse`
    - `length`
    - `sort`
    - `merge`
    - `default`
    - `keys`
    - `escape`
    - `e`
- Functions:
    - `range`
    - `constant`
    - `cycle`
    - `parent`

    - `block`

- Tests:

    - `even`
    - `odd`
    - `defined`
    - `sameas`
    - `null`
    - `divisibleby`
    - `constant`
    - `empty`

Escaper Extension

The `escaper` extension adds automatic output escaping to Twig. It defines a tag, `autoescape`, and a filter, `raw`.

When creating the escaper extension, you can switch on or off the global output escaping strategy:

Listing 3-14
```
$escaper = new Twig_Extension_Escaper(true);
$twig->addExtension($escaper);
```

If set to `true`, all variables in templates are escaped (using the `html` escaping strategy), except those using the `raw` filter:

Listing 3-15
```
{{ article.to_html|raw }}
```

You can also change the escaping mode locally by using the `autoescape` tag (see the *autoescape* doc for the syntax used before Twig 1.8):

Listing 3-16
```
{% autoescape 'html' %}
    {{ var }}
    {{ var|raw }}      {# var won't be escaped #}
    {{ var|escape }}   {# var won't be double-escaped #}
{% endautoescape %}
```

The `autoescape` tag has no effect on included files.

The escaping rules are implemented as follows:

- Literals (integers, booleans, arrays, ...) used in the template directly as variables or filter arguments are never automatically escaped:

Listing 3-17
```
{{ "Twig<br />" }} {# won't be escaped #}

{% set text = "Twig<br />" %}
{{ text }} {# will be escaped #}
```

- Expressions which the result is always a literal or a variable marked safe are never automatically escaped:

Listing 3-18
```
{{ foo ? "Twig<br />" : "<br />Twig" }} {# won't be escaped #}

{% set text = "Twig<br />" %}
{{ foo ? text : "<br />Twig" }} {# will be escaped #}
```

```
{% set text = "Twig<br />" %}
{{ foo ? text|raw : "<br />Twig" }} {# won't be escaped #}

{% set text = "Twig<br />" %}
{{ foo ? text|escape : "<br />Twig" }} {# the result of the expression won't be
escaped #}
```

- Escaping is applied before printing, after any other filter is applied:

```
{{ var|upper }} {# is equivalent to {{ var|upper|escape }} #}
```

Listing 3-19

- The *raw* filter should only be used at the end of the filter chain:

```
{{ var|raw|upper }} {# will be escaped #}

{{ var|upper|raw }} {# won't be escaped #}
```

Listing 3-20

- Automatic escaping is not applied if the last filter in the chain is marked safe for the current context (e.g. `html` or `js`). `escaper` and `escaper('html')` are marked safe for html, `escaper('js')` is marked safe for javascript, `raw` is marked safe for everything.

```
{% autoescape true js %}
{{ var|escape('html') }} {# will be escaped for html and javascript #}
{{ var }} {# will be escaped for javascript #}
{{ var|escape('js') }} {# won't be double-escaped #}
{% endautoescape %}
```

Listing 3-21

Note that autoescaping has some limitations as escaping is applied on expressions after evaluation. For instance, when working with concatenation, `{{ foo|raw ~ bar }}` won't give the expected result as escaping is applied on the result of the concatenation, not on the individual variables (so, the `raw` filter won't have any effect here).

Sandbox Extension

The `sandbox` extension can be used to evaluate untrusted code. Access to unsafe attributes and methods is prohibited. The sandbox security is managed by a policy instance. By default, Twig comes with one policy class: `Twig_Sandbox_SecurityPolicy`. This class allows you to white-list some tags, filters, properties, and methods:

```
$tags = array('if');
$filters = array('upper');
$methods = array(
    'Article' => array('getTitle', 'getBody'),
);
$properties = array(
    'Article' => array('title', 'body'),
);
$functions = array('range');
$policy = new Twig_Sandbox_SecurityPolicy($tags, $filters, $methods, $properties, $functions);
```

Listing 3-22

With the previous configuration, the security policy will only allow usage of the `if` tag, and the `upper` filter. Moreover, the templates will only be able to call the `getTitle()` and `getBody()` methods on `Article` objects, and the `title` and `body` public properties. Everything else won't be allowed and will generate a `Twig_Sandbox_SecurityError` exception.

The policy object is the first argument of the sandbox constructor:

Listing 3-23
```
$sandbox = new Twig_Extension_Sandbox($policy);
$twig->addExtension($sandbox);
```

By default, the sandbox mode is disabled and should be enabled when including untrusted template code by using the `sandbox` tag:

Listing 3-24
```
{% sandbox %}
    {% include 'user.html' %}
{% endsandbox %}
```

You can sandbox all templates by passing `true` as the second argument of the extension constructor:

Listing 3-25
```
$sandbox = new Twig_Extension_Sandbox($policy, true);
```

Optimizer Extension

The `optimizer` extension optimizes the node tree before compilation:

Listing 3-26
```
$twig->addExtension(new Twig_Extension_Optimizer());
```

By default, all optimizations are turned on. You can select the ones you want to enable by passing them to the constructor:

Listing 3-27
```
$optimizer = new Twig_Extension_Optimizer(Twig_NodeVisitor_Optimizer::OPTIMIZE_FOR);

$twig->addExtension($optimizer);
```

Exceptions

Twig can throw exceptions:

- `Twig_Error`: The base exception for all errors.
- `Twig_Error_Syntax`: Thrown to tell the user that there is a problem with the template syntax.
- `Twig_Error_Runtime`: Thrown when an error occurs at runtime (when a filter does not exist for instance).
- `Twig_Error_Loader`: Thrown when an error occurs during template loading.
- `Twig_Sandbox_SecurityError`: Thrown when an unallowed tag, filter, or method is called in a sandboxed template.

Chapter 4
Extending Twig

Twig can be extended in many ways; you can add extra tags, filters, tests, operators, global variables, and functions. You can even extend the parser itself with node visitors.

The first section of this chapter describes how to extend Twig easily. If you want to reuse your changes in different projects or if you want to share them with others, you should then create an extension as described in the following section.

When extending Twig by calling methods on the Twig environment instance, Twig won't be able to recompile your templates when the PHP code is updated. To see your changes in real-time, either disable template caching or package your code into an extension (see the next section of this chapter).

Before extending Twig, you must understand the differences between all the different possible extension points and when to use them.

First, remember that Twig has two main language constructs:

- `{{ }}`: used to print the result of an expression evaluation;
- `{% %}`: used to execute statements.

To understand why Twig exposes so many extension points, let's see how to implement a *Lorem ipsum* generator (it needs to know the number of words to generate).

You can use a `lipsum` *tag*:

```
{% lipsum 40 %}
```

Listing 4-1

That works, but using a tag for `lipsum` is not a good idea for at least three main reasons:

- `lipsum` is not a language construct;
- The tag outputs something;
- The tag is not flexible as you cannot use it in an expression:

Listing 4-2
```
{{ 'some text' ~ {% lipsum 40 %} ~ 'some more text' }}
```

In fact, you rarely need to create tags; and that's good news because tags are the most complex extension point of Twig.

Now, let's use a `lipsum` *filter*:

Listing 4-3
```
{{ 40|lipsum }}
```

Again, it works, but it looks weird. A filter transforms the passed value to something else but here we use the value to indicate the number of words to generate.

Next, let's use a `lipsum` *function*:

Listing 4-4
```
{{ lipsum(40) }}
```

Here we go. For this specific example, the creation of a function is the extension point to use. And you can use it anywhere an expression is accepted:

Listing 4-5
```
{{ 'some text' ~ ipsum(40) ~ 'some more text' }}

{% set ipsum = ipsum(40) %}
```

Last but not the least, you can also use a *global* object with a method able to generate lorem ipsum text:

Listing 4-6
```
{{ text.lipsum(40) }}
```

As a rule of thumb, use functions for frequently used features and global objects for everything else.

Keep in mind the following when you want to extend Twig:

What?	Implementation difficulty?	How often?	When?
macro	trivial	frequent	Content generation
global	trivial	frequent	Helper object
function	trivial	frequent	Content generation
filter	trivial	frequent	Value transformation
tag	complex	rare	DSL language construct
test	trivial	rare	Boolean decision
operator	trivial	rare	Values transformation

Globals

A global variable is like any other template variable, except that it's available in all templates and macros:

Listing 4-7
```
$twig = new Twig_Environment($loader);
$twig->addGlobal('text', new Text());
```

You can then use the `text` variable anywhere in a template:

Listing 4-8
```
{{ text.lipsum(40) }}
```

Filters

A filter is a regular PHP function or an object method that takes the left side of the filter (before the pipe `|`) as first argument and the extra arguments passed to the filter (within parentheses `()`) as extra arguments.

Defining a filter is as easy as associating the filter name with a PHP callable. For instance, let's say you have the following code in a template:

```
{{ 'TWIG'|lower }}
```

Listing 4-9

When compiling this template to PHP, Twig looks for the PHP callable associated with the `lower` filter. The `lower` filter is a built-in Twig filter, and it is simply mapped to the PHP `strtolower()` function. After compilation, the generated PHP code is roughly equivalent to:

```
<?php echo strtolower('TWIG') ?>
```

Listing 4-10

As you can see, the `'TWIG'` string is passed as a first argument to the PHP function.

A filter can also take extra arguments like in the following example:

```
{{ now|date('d/m/Y') }}
```

Listing 4-11

In this case, the extra arguments are passed to the function after the main argument, and the compiled code is equivalent to:

```
<?php echo twig_date_format_filter($now, 'd/m/Y') ?>
```

Listing 4-12

Let's see how to create a new filter.

In this section, we will create a `rot13` filter, which should return the *rot13*[1] transformation of a string. Here is an example of its usage and the expected output:

```
{{ "Twig"|rot13 }}

{# should displays Gjvt #}
```

Listing 4-13

Adding a filter is as simple as calling the `addFilter()` method on the `Twig_Environment` instance:

```
$twig = new Twig_Environment($loader);
$twig->addFilter('rot13', new Twig_Filter_Function('str_rot13'));
```

Listing 4-14

The second argument of `addFilter()` is an instance of `Twig_Filter`. Here, we use `Twig_Filter_Function` as the filter is a PHP function. The first argument passed to the `Twig_Filter_Function` constructor is the name of the PHP function to call, here `str_rot13`, a native PHP function.

Let's say I now want to be able to add a prefix before the converted string:

```
{{ "Twig"|rot13('prefix_') }}

{# should displays prefix_Gjvt #}
```

Listing 4-15

As the PHP `str_rot13()` function does not support this requirement, let's create a new PHP function:

```
function project_compute_rot13($string, $prefix = '')
{
```

Listing 4-16

1. http://www.php.net/manual/en/function.str-rot13.php

```
    return $prefix.str_rot13($string);
}
```

As you can see, the `prefix` argument of the filter is passed as an extra argument to the `project_compute_rot13()` function.

Adding this filter is as easy as before:

Listing 4-17
```
$twig->addFilter('rot13', new Twig_Filter_Function('project_compute_rot13'));
```

For better encapsulation, a filter can also be defined as a static method of a class. The `Twig_Filter_Function` class can also be used to register such static methods as filters:

Listing 4-18
```
$twig->addFilter('rot13', new Twig_Filter_Function('SomeClass::rot13Filter'));
```

In an extension, you can also define a filter as a static method of the extension class.

Environment aware Filters

The `Twig_Filter` classes take options as their last argument. For instance, if you want access to the current environment instance in your filter, set the `needs_environment` option to `true`:

Listing 4-19
```
$filter = new Twig_Filter_Function('str_rot13', array('needs_environment' => true));
```

Twig will then pass the current environment as the first argument to the filter call:

Listing 4-20
```
function twig_compute_rot13(Twig_Environment $env, $string)
{
    // get the current charset for instance
    $charset = $env->getCharset();

    return str_rot13($string);
}
```

Automatic Escaping

If automatic escaping is enabled, the output of the filter may be escaped before printing. If your filter acts as an escaper (or explicitly outputs html or javascript code), you will want the raw output to be printed. In such a case, set the `is_safe` option:

Listing 4-21
```
$filter = new Twig_Filter_Function('nl2br', array('is_safe' => array('html')));
```

Some filters may have to work on already escaped or safe values. In such a case, set the `pre_escape` option:

Listing 4-22
```
$filter = new Twig_Filter_Function('somefilter', array('pre_escape' => 'html', 'is_safe' =>
array('html')));
```

Dynamic Filters

New in version 1.5: Dynamic filters support was added in Twig 1.5.

A filter name containing the special * character is a dynamic filter as the * can be any string:

```
$twig->addFilter('*_path', new Twig_Filter_Function('twig_path'));

function twig_path($name, $arguments)
{
    // ...
}
```

Listing 4-23

The following filters will be matched by the above defined dynamic filter:

- `product_path`
- `category_path`

A dynamic filter can define more than one dynamic parts:

```
$twig->addFilter('*_path_*', new Twig_Filter_Function('twig_path'));

function twig_path($name, $suffix, $arguments)
{
    // ...
}
```

Listing 4-24

The filter will receive all dynamic part values before the normal filters arguments. For instance, a call to `'foo'|a_path_b()` will result in the following PHP call: `twig_path('a', 'b', 'foo')`.

Functions

A function is a regular PHP function or an object method that can be called from templates.

```
{{ constant("DATE_W3C") }}
```

Listing 4-25

When compiling this template to PHP, Twig looks for the PHP callable associated with the `constant` function. The `constant` function is a built-in Twig function, and it is simply mapped to the PHP `constant()` function. After compilation, the generated PHP code is roughly equivalent to:

```
<?php echo constant('DATE_W3C') ?>
```

Listing 4-26

Adding a function is similar to adding a filter. This can be done by calling the `addFunction()` method on the `Twig_Environment` instance:

```
$twig = new Twig_Environment($loader);
$twig->addFunction('functionName', new Twig_Function_Function('someFunction'));
```

Listing 4-27

You can also expose extension methods as functions in your templates:

```
// $this is an object that implements Twig_ExtensionInterface.
$twig = new Twig_Environment($loader);
$twig->addFunction('otherFunction', new Twig_Function_Method($this, 'someMethod'));
```

Listing 4-28

Functions also support `needs_environment` and `is_safe` parameters.

Dynamic Functions

New in version 1.5: Dynamic functions support was added in Twig 1.5.

A function name containing the special * character is a dynamic function as the * can be any string:

Listing 4-29
```
$twig->addFunction('*_path', new Twig_Function_Function('twig_path'));

function twig_path($name, $arguments)
{
    // ...
}
```

The following functions will be matched by the above defined dynamic function:

- `product_path`
- `category_path`

A dynamic function can define more than one dynamic parts:

Listing 4-30
```
$twig->addFilter('*_path_*', new Twig_Filter_Function('twig_path'));

function twig_path($name, $suffix, $arguments)
{
    // ...
}
```

The function will receive all dynamic part values before the normal functions arguments. For instance, a call to `a_path_b('foo')` will result in the following PHP call: `twig_path('a', 'b', 'foo')`.

Tags

One of the most exciting feature of a template engine like Twig is the possibility to define new language constructs. This is also the most complex feature as you need to understand how Twig's internals work.

Let's create a simple `set` tag that allows the definition of simple variables from within a template. The tag can be used like follows:

Listing 4-31
```
{% set name = "value" %}

{{ name }}

{# should output value #}
```

The `set` tag is part of the Core extension and as such is always available. The built-in version is slightly more powerful and supports multiple assignments by default (cf. the template designers chapter for more information).

Three steps are needed to define a new tag:

- Defining a Token Parser class (responsible for parsing the template code);
- Defining a Node class (responsible for converting the parsed code to PHP);
- Registering the tag.

Registering a new tag

Adding a tag is as simple as calling the `addTokenParser` method on the `Twig_Environment` instance:

Listing 4-32

```
$twig = new Twig_Environment($loader);
$twig->addTokenParser(new Project_Set_TokenParser());
```

Defining a Token Parser

Now, let's see the actual code of this class:

Listing 4-33

```
class Project_Set_TokenParser extends Twig_TokenParser
{
    public function parse(Twig_Token $token)
    {
        $lineno = $token->getLine();
        $name = $this->parser->getStream()->expect(Twig_Token::NAME_TYPE)->getValue();
        $this->parser->getStream()->expect(Twig_Token::OPERATOR_TYPE, '=');
        $value = $this->parser->getExpressionParser()->parseExpression();

        $this->parser->getStream()->expect(Twig_Token::BLOCK_END_TYPE);

        return new Project_Set_Node($name, $value, $lineno, $this->getTag());
    }

    public function getTag()
    {
        return 'set';
    }
}
```

The `getTag()` method must return the tag we want to parse, here `set`.

The `parse()` method is invoked whenever the parser encounters a `set` tag. It should return a `Twig_Node` instance that represents the node (the `Project_Set_Node` calls creating is explained in the next section).

The parsing process is simplified thanks to a bunch of methods you can call from the token stream (`$this->parser->getStream()`):

- `getCurrent()`: Gets the current token in the stream.
- `next()`: Moves to the next token in the stream, *but returns the old one*.
- `test($type)`, `test($value)` or `test($type, $value)`: Determines whether the current token is of a particular type or value (or both). The value may be an array of several possible values.
- `expect($type[, $value[, $message]])`: If the current token isn't of the given type/value a syntax error is thrown. Otherwise, if the type and value are correct, the token is returned and the stream moves to the next token.
- `look()`: Looks a the next token without consuming it.

Parsing expressions is done by calling the `parseExpression()` like we did for the `set` tag.

Reading the existing `TokenParser` classes is the best way to learn all the nitty-gritty details of the parsing process.

Defining a Node

The `Project_Set_Node` class itself is rather simple:

Listing 4-34
```php
class Project_Set_Node extends Twig_Node
{
    public function __construct($name, Twig_Node_Expression $value, $lineno, $tag = null)
    {
        parent::__construct(array('value' => $value), array('name' => $name), $lineno, $tag);
    }

    public function compile(Twig_Compiler $compiler)
    {
        $compiler
            ->addDebugInfo($this)
            ->write('$context[\''.$this->getAttribute('name').'\'] = ')
            ->subcompile($this->getNode('value'))
            ->raw(";\n")
        ;
    }
}
```

The compiler implements a fluid interface and provides methods that helps the developer generate beautiful and readable PHP code:

- `subcompile()`: Compiles a node.
- `raw()`: Writes the given string as is.
- `write()`: Writes the given string by adding indentation at the beginning of each line.
- `string()`: Writes a quoted string.
- `repr()`: Writes a PHP representation of a given value (see `Twig_Node_For` for a usage example).
- `addDebugInfo()`: Adds the line of the original template file related to the current node as a comment.
- `indent()`: Indents the generated code (see `Twig_Node_Block` for a usage example).
- `outdent()`: Outdents the generated code (see `Twig_Node_Block` for a usage example).

Creating an Extension

The main motivation for writing an extension is to move often used code into a reusable class like adding support for internationalization. An extension can define tags, filters, tests, operators, global variables, functions, and node visitors.

Creating an extension also makes for a better separation of code that is executed at compilation time and code needed at runtime. As such, it makes your code faster.

Most of the time, it is useful to create a single extension for your project, to host all the specific tags and filters you want to add to Twig.

When packaging your code into an extension, Twig is smart enough to recompile your templates whenever you make a change to it (when the `auto_reload` is enabled).

Before writing your own extensions, have a look at the Twig official extension repository: *http://github.com/fabpot/Twig-extensions*[2].

An extension is a class that implements the following interface:

2. `http://github.com/fabpot/Twig-extensions`

Listing 4-35

```
interface Twig_ExtensionInterface
{
    /**
     * Initializes the runtime environment.
     *
     * This is where you can load some file that contains filter functions for instance.
     *
     * @param Twig_Environment $environment The current Twig_Environment instance
     */
    function initRuntime(Twig_Environment $environment);

    /**
     * Returns the token parser instances to add to the existing list.
     *
     * @return array An array of Twig_TokenParserInterface or Twig_TokenParserBrokerInterface
instances
     */
    function getTokenParsers();

    /**
     * Returns the node visitor instances to add to the existing list.
     *
     * @return array An array of Twig_NodeVisitorInterface instances
     */
    function getNodeVisitors();

    /**
     * Returns a list of filters to add to the existing list.
     *
     * @return array An array of filters
     */
    function getFilters();

    /**
     * Returns a list of tests to add to the existing list.
     *
     * @return array An array of tests
     */
    function getTests();

    /**
     * Returns a list of functions to add to the existing list.
     *
     * @return array An array of functions
     */
    function getFunctions();

    /**
     * Returns a list of operators to add to the existing list.
     *
     * @return array An array of operators
     */
    function getOperators();

    /**
     * Returns a list of global variables to add to the existing list.
     *
     * @return array An array of global variables
     */
    function getGlobals();
```

```
    /**
     * Returns the name of the extension.
     *
     * @return string The extension name
     */
    function getName();
}
```

To keep your extension class clean and lean, it can inherit from the built-in `Twig_Extension` class instead of implementing the whole interface. That way, you just need to implement the `getName()` method as the `Twig_Extension` provides empty implementations for all other methods.

The `getName()` method must return a unique identifier for your extension.

Now, with this information in mind, let's create the most basic extension possible:

Listing 4-36
```
class Project_Twig_Extension extends Twig_Extension
{
    public function getName()
    {
        return 'project';
    }
}
```

Of course, this extension does nothing for now. We will customize it in the next sections.

Twig does not care where you save your extension on the filesystem, as all extensions must be registered explicitly to be available in your templates.

You can register an extension by using the `addExtension()` method on your main `Environment` object:

Listing 4-37
```
$twig = new Twig_Environment($loader);
$twig->addExtension(new Project_Twig_Extension());
```

Of course, you need to first load the extension file by either using `require_once()` or by using an autoloader (see *spl_autoload_register()*[3]).

The bundled extensions are great examples of how extensions work.

Globals

Global variables can be registered in an extension via the `getGlobals()` method:

Listing 4-38
```
class Project_Twig_Extension extends Twig_Extension
{
    public function getGlobals()
    {
        return array(
            'text' => new Text(),
        );
    }
```

3. http://www.php.net/spl_autoload_register

```
    // ...
}
```

Functions

Functions can be registered in an extension via the `getFunctions()` method:

Listing 4-39

```
class Project_Twig_Extension extends Twig_Extension
{
    public function getFunctions()
    {
        return array(
            'lipsum' => new Twig_Function_Function('generate_lipsum'),
        );
    }

    // ...
}
```

Filters

To add a filter to an extension, you need to override the `getFilters()` method. This method must return an array of filters to add to the Twig environment:

Listing 4-40

```
class Project_Twig_Extension extends Twig_Extension
{
    public function getFilters()
    {
        return array(
            'rot13' => new Twig_Filter_Function('str_rot13'),
        );
    }

    // ...
}
```

As you can see in the above code, the `getFilters()` method returns an array where keys are the name of the filters (`rot13`) and the values the definition of the filter (`new Twig_Filter_Function('str_rot13')`).

As seen in the previous chapter, you can also define filters as static methods on the extension class:

Listing 4-41

```
$twig->addFilter('rot13', new Twig_Filter_Function('Project_Twig_Extension::rot13Filter'));
```

You can also use `Twig_Filter_Method` instead of `Twig_Filter_Function` when defining a filter to use a method:

Listing 4-42

```
class Project_Twig_Extension extends Twig_Extension
{
    public function getFilters()
    {
        return array(
            'rot13' => new Twig_Filter_Method($this, 'rot13Filter'),
        );
    }

    public function rot13Filter($string)
    {
        return str_rot13($string);
    }
```

```
    // ...
}
```

The first argument of the `Twig_Filter_Method` constructor is always `$this`, the current extension object. The second one is the name of the method to call.

Using methods for filters is a great way to package your filter without polluting the global namespace. This also gives the developer more flexibility at the cost of a small overhead.

Overriding default Filters

If some default core filters do not suit your needs, you can easily override them by creating your own core extension. Of course, you don't need to copy and paste the whole core extension code of Twig. Instead, you can just extends it and override the filter(s) you want by overriding the `getFilters()` method:

Listing 4-43

```
class MyCoreExtension extends Twig_Extension_Core
{
    public function getFilters()
    {
        return array_merge(parent::getFilters(), array(
            'date' => new Twig_Filter_Method($this, 'dateFilter'),
            // ...
        ));
    }

    public function dateFilter($timestamp, $format = 'F j, Y H:i')
    {
        return '...'.twig_date_format_filter($timestamp, $format);
    }

    // ...
}
```

Here, we override the `date` filter with a custom one. Using this new core extension is as simple as registering the `MyCoreExtension` extension by calling the `addExtension()` method on the environment instance:

Listing 4-44

```
$twig = new Twig_Environment($loader);
$twig->addExtension(new MyCoreExtension());
```

But I can already hear some people wondering how it can work as the Core extension is loaded by default. That's true, but the trick is that both extensions share the same unique identifier (`core` - defined in the `getName()` method). By registering an extension with the same name as an existing one, you have actually overridden the default one, even if it is already registered:

Listing 4-45

```
$twig->addExtension(new Twig_Extension_Core());
$twig->addExtension(new MyCoreExtension());
```

Tags

Adding a tag in an extension can be done by overriding the `getTokenParsers()` method. This method must return an array of tags to add to the Twig environment:

Listing 4-46

```
class Project_Twig_Extension extends Twig_Extension
{
    public function getTokenParsers()
    {
        return array(new Project_Set_TokenParser());
```

```
    }

    // ...
}
```

In the above code, we have added a single new tag, defined by the `Project_Set_TokenParser` class. The `Project_Set_TokenParser` class is responsible for parsing the tag and compiling it to PHP.

Operators

The `getOperators()` methods allows to add new operators. Here is how to add `!`, `||`, and `&&` operators:

Listing 4-47

```
class Project_Twig_Extension extends Twig_Extension
{
    public function getOperators()
    {
        return array(
            array(
                '!' => array('precedence' => 50, 'class' => 'Twig_Node_Expression_Unary_Not'),
            ),
            array(
                '||' => array('precedence' => 10, 'class' => 'Twig_Node_Expression_Binary_Or',
'associativity' => Twig_ExpressionParser::OPERATOR_LEFT),
                '&&' => array('precedence' => 15, 'class' =>
'Twig_Node_Expression_Binary_And', 'associativity' => Twig_ExpressionParser::OPERATOR_LEFT),
            ),
        );
    }

    // ...
}
```

Tests

The `getTests()` methods allows to add new test functions:

Listing 4-48

```
class Project_Twig_Extension extends Twig_Extension
{
    public function getTests()
    {
        return array(
            'even' => new Twig_Test_Function('twig_test_even'),
        );
    }

    // ...
}
```

Chapter 5
Twig Internals

Twig is very extensible and you can easily hack it. Keep in mind that you should probably try to create an extension before hacking the core, as most features and enhancements can be done with extensions. This chapter is also useful for people who want to understand how Twig works under the hood.

How Twig works?

The rendering of a Twig template can be summarized into four key steps:

- **Load** the template: If the template is already compiled, load it and go to the *evaluation* step, otherwise:
    - First, the **lexer** tokenizes the template source code into small pieces for easier processing;
    - Then, the **parser** converts the token stream into a meaningful tree of nodes (the Abstract Syntax Tree);
    - Eventually, the *compiler* transforms the AST into PHP code;
- **Evaluate** the template: It basically means calling the `display()` method of the compiled template and passing it the context.

The Lexer

The lexer tokenizes a template source code into a token stream (each token is an instance of `Twig_Token`, and the stream is an instance of `Twig_TokenStream`). The default lexer recognizes 13 different token types:

- `Twig_Token::BLOCK_START_TYPE`, `Twig_Token::BLOCK_END_TYPE`: Delimiters for blocks ({% %})
- `Twig_Token::VAR_START_TYPE`, `Twig_Token::VAR_END_TYPE`: Delimiters for variables ({{ }})
- `Twig_Token::TEXT_TYPE`: A text outside an expression;
- `Twig_Token::NAME_TYPE`: A name in an expression;
- `Twig_Token::NUMBER_TYPE`: A number in an expression;

- `Twig_Token::STRING_TYPE`: A string in an expression;
- `Twig_Token::OPERATOR_TYPE`: An operator;
- `Twig_Token::PUNCTUATION_TYPE`: A punctuation sign;
- `Twig_Token::INTERPOLATION_START_TYPE`, `Twig_Token::INTERPOLATION_END_TYPE` (as of Twig 1.5): Delimiters for string interpolation;
- `Twig_Token::EOF_TYPE`: Ends of template.

You can manually convert a source code into a token stream by calling the `tokenize()` of an environment:

```
$stream = $twig->tokenize($source, $identifier);
```

Listing 5-1

As the stream has a `__toString()` method, you can have a textual representation of it by echoing the object:

```
echo $stream."\n";
```

Listing 5-2

Here is the output for the `Hello {{ name }}` template:

```
TEXT_TYPE(Hello )
VAR_START_TYPE()
NAME_TYPE(name)
VAR_END_TYPE()
EOF_TYPE()
```

Listing 5-3

You can change the default lexer use by Twig (`Twig_Lexer`) by calling the `setLexer()` method:

```
$twig->setLexer($lexer);
```

Listing 5-4

The Parser

The parser converts the token stream into an AST (Abstract Syntax Tree), or a node tree (an instance of `Twig_Node_Module`). The core extension defines the basic nodes like: `for`, `if`, ... and the expression nodes.

You can manually convert a token stream into a node tree by calling the `parse()` method of an environment:

```
$nodes = $twig->parse($stream);
```

Listing 5-5

Echoing the node object gives you a nice representation of the tree:

```
echo $nodes."\n";
```

Listing 5-6

Here is the output for the `Hello {{ name }}` template:

```
Twig_Node_Module(
  Twig_Node_Text(Hello )
  Twig_Node_Print(
    Twig_Node_Expression_Name(name)
  )
)
```

Listing 5-7

The default parser (`Twig_TokenParser`) can be also changed by calling the `setParser()` method:

Listing 5-8
```
$twig->setParser($parser);
```

The Compiler

The last step is done by the compiler. It takes a node tree as an input and generates PHP code usable for runtime execution of the template.

You can call the compiler by hand with the `compile()` method of an environment:

Listing 5-9
```
$php = $twig->compile($nodes);
```

The `compile()` method returns the PHP source code representing the node.

The generated template for a `Hello {{ name }}` template reads as follows (the actual output can differ depending on the version of Twig you are using):

Listing 5-10
```
/* Hello {{ name }} */
class __TwigTemplate_1121b6f109fe93ebe8c6e22e3712bceb extends Twig_Template
{
    protected function doDisplay(array $context, array $blocks = array())
    {
        // line 1
        echo "Hello ";
        echo twig_escape_filter($this->env, $this->getContext($context, "name"), "ndex", null,
true);
    }

    // some more code
}
```

As for the lexer and the parser, the default compiler (`Twig_Compiler`) can be changed by calling the `setCompiler()` method:

Listing 5-11
```
$twig->setCompiler($compiler);
```

Chapter 6
Recipes

Making a Layout conditional

Working with Ajax means that the same content is sometimes displayed as is, and sometimes decorated with a layout. As Twig layout template names can be any valid expression, you can pass a variable that evaluates to `true` when the request is made via Ajax and choose the layout accordingly:

```
{% extends request.ajax ? "base_ajax.html" : "base.html" %}

{% block content %}
    This is the content to be displayed.
{% endblock %}
```

Listing 6-1

Making an Include dynamic

When including a template, its name does not need to be a string. For instance, the name can depend on the value of a variable:

```
{% include var ~ '_foo.html' %}
```

Listing 6-2

If `var` evaluates to `index`, the `index_foo.html` template will be rendered.

As a matter of fact, the template name can be any valid expression, such as the following:

```
{% include var|default('index') ~ '_foo.html' %}
```

Listing 6-3

Overriding a Template that also extends itself

A template can be customized in two different ways:

- *Inheritance*: A template *extends* a parent template and overrides some blocks;

- *Replacement*: If you use the filesystem loader, Twig loads the first template it finds in a list of configured directories; a template found in a directory *replaces* another one from a directory further in the list.

But how do you combine both: *replace* a template that also extends itself (aka a template in a directory further in the list)?

Let's say that your templates are loaded from both `.../templates/mysite` and `.../templates/default` in this order. The `page.twig` template, stored in `.../templates/default` reads as follows:

Listing 6-4
```
{# page.twig #}
{% extends "layout.twig" %}

{% block content %}
{% endblock %}
```

You can replace this template by putting a file with the same name in `.../templates/mysite`. And if you want to extend the original template, you might be tempted to write the following:

Listing 6-5
```
{# page.twig in .../templates/mysite #}
{% extends "page.twig" %} {# from .../templates/default #}
```

Of course, this will not work as Twig will always load the template from `.../templates/mysite`.

It turns out it is possible to get this to work, by adding a directory right at the end of your template directories, which is the parent of all of the other directories: `.../templates` in our case. This has the effect of making every template file within our system uniquely addressable. Most of the time you will use the "normal" paths, but in the special case of wanting to extend a template with an overriding version of itself we can reference its parent's full, unambiguous template path in the extends tag:

Listing 6-6
```
{# page.twig in .../templates/mysite #}
{% extends "default/page.twig" %} {# from .../templates #}
```

This recipe was inspired by the following Django wiki page: *http://code.djangoproject.com/wiki/ExtendingTemplates*[1]

Customizing the Syntax

Twig allows some syntax customization for the block delimiters. It's not recommended to use this feature as templates will be tied with your custom syntax. But for specific projects, it can make sense to change the defaults.

To change the block delimiters, you need to create your own lexer object:

Listing 6-7
```
$twig = new Twig_Environment();

$lexer = new Twig_Lexer($twig, array(
    'tag_comment'  => array('{#', '#}'),
    'tag_block'    => array('{%', '%}'),
    'tag_variable' => array('{{', '}}'),
));
$twig->setLexer($lexer);
```

Here are some configuration example that simulates some other template engines syntax:

1. `http://code.djangoproject.com/wiki/ExtendingTemplates`

Listing 6-8

```
// Ruby erb syntax
$lexer = new Twig_Lexer($twig, array(
    'tag_comment'  => array('<%#', '%>'),
    'tag_block'    => array('<%', '%>'),
    'tag_variable' => array('<%=', '%>'),
));

// SGML Comment Syntax
$lexer = new Twig_Lexer($twig, array(
    'tag_comment'  => array('<!--#', '-->'),
    'tag_block'    => array('<!--', '-->'),
    'tag_variable' => array('${', '}'),
));

// Smarty like
$lexer = new Twig_Lexer($twig, array(
    'tag_comment'  => array('{*', '*}'),
    'tag_block'    => array('{', '}'),
    'tag_variable' => array('{$', '}'),
));
```

Using dynamic Object Properties

When Twig encounters a variable like `article.title`, it tries to find a `title` public property in the `article` object.

It also works if the property does not exist but is rather defined dynamically thanks to the magic `__get()` method; you just need to also implement the `__isset()` magic method like shown in the following snippet of code:

Listing 6-9

```
class Article
{
    public function __get($name)
    {
        if ('title' == $name) {
            return 'The title';
        }

        // throw some kind of error
    }

    public function __isset($name)
    {
        if ('title' == $name) {
            return true;
        }

        return false;
    }
}
```

Accessing the parent Context in Nested Loops

Sometimes, when using nested loops, you need to access the parent context. The parent context is always accessible via the `loop.parent` variable. For instance, if you have the following template data:

Listing 6-10

```php
$data = array(
    'topics' => array(
        'topic1' => array('Message 1 of topic 1', 'Message 2 of topic 1'),
        'topic2' => array('Message 1 of topic 2', 'Message 2 of topic 2'),
    ),
);
```

And the following template to display all messages in all topics:

Listing 6-11
```
{% for topic, messages in topics %}
    * {{ loop.index }}: {{ topic }}
  {% for message in messages %}
      - {{ loop.parent.loop.index }}.{{ loop.index }}: {{ message }}
  {% endfor %}
{% endfor %}
```

The output will be similar to:

Listing 6-12
```
* 1: topic1
  - 1.1: The message 1 of topic 1
  - 1.2: The message 2 of topic 1
* 2: topic2
  - 2.1: The message 1 of topic 2
  - 2.2: The message 2 of topic 2
```

In the inner loop, the `loop.parent` variable is used to access the outer context. So, the index of the current `topic` defined in the outer for loop is accessible via the `loop.parent.loop.index` variable.

Defining undefined Functions and Filters on the Fly

When a function (or a filter) is not defined, Twig defaults to throw a `Twig_Error_Syntax` exception. However, it can also call a *callback*[2] (any valid PHP callable) which should return a function (or a filter).

For filters, register callbacks with `registerUndefinedFilterCallback()`. For functions, use `registerUndefinedFunctionCallback()`:

Listing 6-13
```php
// auto-register all native PHP functions as Twig functions
// don't try this at home as it's not secure at all!
$twig->registerUndefinedFunctionCallback(function ($name) {
    if (function_exists($name)) {
        return new Twig_Function_Function($name);
    }

    return false;
});
```

If the callable is not able to return a valid function (or filter), it must return `false`.

If you register more than one callback, Twig will call them in turn until one does not return `false`.

As the resolution of functions and filters is done during compilation, there is no overhead when registering these callbacks.

2. http://www.php.net/manual/en/function.is-callable.php

Validating the Template Syntax

When template code is providing by a third-party (through a web interface for instance), it might be interesting to validate the template syntax before saving it. If the template code is stored in a *$template* variable, here is how you can do it:

Listing 6-14

```
try {
    $twig->parse($twig->tokenize($template));

    // the $template is valid
} catch (Twig_Error_Syntax $e) {
    // $template contains one or more syntax errors
}
```

If you iterate over a set of files, you can pass the filename to the `tokenize()` method to get the filename in the exception message:

Listing 6-15

```
foreach ($files as $file) {
    try {
        $twig->parse($twig->tokenize($template, $file));

        // the $template is valid
    } catch (Twig_Error_Syntax $e) {
        // $template contains one or more syntax errors
    }
}
```

This method won't catch any sandbox policy violations because the policy is enforced during template rendering (as Twig needs the context for some checks like allowed methods on objects).

Refreshing modified Templates when APC is enabled and apc.stat = 0

When using APC with `apc.stat` set to `0` and Twig cache enabled, clearing the template cache won't update the APC cache. To get around this, one can extend `Twig_Environment` and force the update of the APC cache when Twig rewrites the cache:

Listing 6-16

```
class Twig_Environment_APC extends Twig_Environment
{
    protected function writeCacheFile($file, $content)
    {
        parent::writeCacheFile($file, $content);

        // Compile cached file into bytecode cache
        apc_compile_file($file);
    }
}
```

Reusing a stateful Node Visitor

When attaching a visitor to a `Twig_Environment` instance, Twig uses it to visit *all* templates it compiles. If you need to keep some state information around, you probably want to reset it when visiting a new template.

This can be easily achieved with the following code:

Listing 6-17
```
protected $someTemplateState = array();

public function enterNode(Twig_NodeInterface $node, Twig_Environment $env)
{
    if ($node instanceof Twig_Node_Module) {
        // reset the state as we are entering a new template
        $this->someTemplateState = array();
    }

    // ...

    return $node;
}
```

Using the Template name to set the default Escaping Strategy

New in version 1.8: This recipe requires Twig 1.8 or later.

The `autoescape` option determines the default escaping strategy to use when no escaping is applied on a variable. When Twig is used to mostly generate HTML files, you can set it to `html` and explicitly change it to `js` when you have some dynamic JavaScript files thanks to the `autoescape` tag:

Listing 6-18
```
{% autoescape js %}
    ... some JS ...
{% endautoescape %}
```

But if you have many HTML and JS files, and if your template names follow some conventions, you can instead determine the default escaping strategy to use based on the template name. Let's say that your template names always ends with `.html` for HTML files and `.js` for JavaScript ones, here is how you can configure Twig:

Listing 6-19
```
function twig_escaping_guesser($filename)
{
    // get the format
    $format = substr($filename, strrpos($filename, '.') + 1);

    switch ($format) {
        'js':
            return 'js';
        default:
            return 'html';
    }
}

$loader = new Twig_Loader_Filesystem('/path/to/templates');
$twig = new Twig_Environment($loader, array(
    'autoescape' => 'twig_escaping_guesser',
));
```

This dynamic strategy does not incur any overhead at runtime as auto-escaping is done at compilation time.

Chapter 7
Coding Standards

When writing Twig templates, we recommend you to follow these official coding standards:

- Put one (and only one) space after the start of a delimiter (`{{`, `{%`, and `{#`) and before the end of a delimiter (`}}`, `%}`, and `#}`):

  ```
  {{ foo }}
  {# comment #}
  {% if foo %}{% endif %}
  ```

  Listing 7-1

  When using the whitespace control character, do not put any spaces between it and the delimiter:

  ```
  {{- foo -}}
  {#- comment -#}
  {%- if foo -%}{%- endif -%}
  ```

  Listing 7-2

- Put one (and only one) space before and after the following operators: comparison operators (`==`, `!=`, `<`, `>`, `>=`, `<=`), math operators (`+`, `-`, `/`, `*`, `%`, `//`, `**`), logic operators (`not`, `and`, `or`), `~`, `is`, `in`, and the ternary operator (`?:`):

  ```
  {{ 1 + 2 }}
  {{ foo ~ bar }}
  {{ true ? true : false }}
  ```

  Listing 7-3

- Put one (and only one) space after the `:` sign in hashes and `,` in arrays and hashes:

  ```
  {{ [1, 2, 3] }}
  {{ {'foo': 'bar'} }}
  ```

  Listing 7-4

- Do not put any spaces after an opening parenthesis and before a closing parenthesis in expressions:

  ```
  {{ 1 + (2 * 3) }}
  ```

  Listing 7-5

- Do not put any spaces before and after string delimiters:

Listing 7-6
```
{{ 'foo' }}
{{ "foo" }}
```

- Do not put any spaces before and after the following operators: `|`, `.`, `..`, `[]`:

Listing 7-7
```
{{ foo|upper|lower }}
{{ user.name }}
{{ user[name] }}
{% for i in 1..12 %}{% endfor %}
```

- Do not put any spaces before and after the parenthesis used for filter and function calls:

Listing 7-8
```
{{ foo|default('foo') }}
{{ range(1..10) }}
```

- Do not put any spaces before and after the opening and the closing of arrays and hashes:

Listing 7-9
```
{{ [1, 2, 3] }}
{{ {'foo': 'bar'} }}
```

- Use lower cased and underscored variable names:

Listing 7-10
```
{% set foo = 'foo' %}
{% set foo_bar = 'foo' %}
```

- Indent your code inside tags (use the same indentation as the one used for the main language of the file):

Listing 7-11
```
{% block foo %}
    {% if true %}
        true
    {% endif %}
{% endblock %}
```

Chapter 8

for

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>
    {% for user in users %}
        <li>{{ user.username|e }}</li>
    {% endfor %}
</ul>
```

Listing 8-1

A sequence can be either an array or an object implementing the `Traversable` interface.

If you do need to iterate over a sequence of numbers, you can use the `..` operator:

```
{% for i in 0..10 %}
    * {{ i }}
{% endfor %}
```

Listing 8-2

The above snippet of code would print all numbers from 0 to 10.

It can be also useful with letters:

```
{% for letter in 'a'..'z' %}
    * {{ letter }}
{% endfor %}
```

Listing 8-3

The `..` operator can take any expression at both sides:

```
{% for letter in 'a'|upper..'z'|upper %}
    * {{ letter }}
{% endfor %}
```

Listing 8-4

The *loop* variable

Inside of a `for` loop block you can access some special variables:

Variable	Description
`loop.index`	The current iteration of the loop. (1 indexed)
`loop.index0`	The current iteration of the loop. (0 indexed)
`loop.revindex`	The number of iterations from the end of the loop (1 indexed)
`loop.revindex0`	The number of iterations from the end of the loop (0 indexed)
`loop.first`	True if first iteration
`loop.last`	True if last iteration
`loop.length`	The number of items in the sequence
`loop.parent`	The parent context

Listing 8-5

```
{% for user in users %}
    {{ loop.index }} - {{ user.username }}
{% endfor %}
```

The `loop.length`, `loop.revindex`, `loop.revindex0`, and `loop.last` variables are only available for PHP arrays, or objects that implement the `Countable` interface. They are also not available when looping with a condition.

New in version 1.2: The `if` modifier support has been added in Twig 1.2.

Adding a condition

Unlike in PHP, it's not possible to `break` or `continue` in a loop. You can however filter the sequence during iteration which allows you to skip items. The following example skips all the users which are not active:

Listing 8-6

```
<ul>
    {% for user in users if user.active %}
        <li>{{ user.username|e }}</li>
    {% endfor %}
</ul>
```

The advantage is that the special loop variable will count correctly thus not counting the users not iterated over. Keep in mind that properties like `loop.last` will not be defined when using loop conditions.

Using the `loop` variable within the condition is not recommended as it will probably not be doing what you expect it to. For instance, adding a condition like `loop.index > 4` won't work as the index is only incremented when the condition is true (so the condition will never match).

The *else* Clause

If no iteration took place because the sequence was empty, you can render a replacement block by using `else`:

Listing 8-7

```
<ul>
    {% for user in users %}
        <li>{{ user.username|e }}</li>
    {% else %}
        <li><em>no user found</em></li>
    {% endfor %}
</ul>
```

Iterating over Keys

By default, a loop iterates over the values of the sequence. You can iterate on keys by using the `keys` filter:

Listing 8-8

```
<h1>Members</h1>
<ul>
    {% for key in users|keys %}
        <li>{{ key }}</li>
    {% endfor %}
</ul>
```

Iterating over Keys and Values

You can also access both keys and values:

Listing 8-9

```
<h1>Members</h1>
<ul>
    {% for key, user in users %}
        <li>{{ key }}: {{ user.username|e }}</li>
    {% endfor %}
</ul>
```

Chapter 9

`if`

The `if` statement in Twig is comparable with the if statements of PHP.

In the simplest form you can use it to test if an expression evaluates to `true`:

Listing 9-1
```
{% if online == false %}
    <p>Our website is in maintenance mode. Please, come back later.</p>
{% endif %}
```

You can also test if an array is not empty:

Listing 9-2
```
{% if users %}
    <ul>
        {% for user in users %}
            <li>{{ user.username|e }}</li>
        {% endfor %}
    </ul>
{% endif %}
```

If you want to test if the variable is defined, use `if users is defined` instead.

For multiple branches `elseif` and `else` can be used like in PHP. You can use more complex `expressions` there too:

Listing 9-3
```
{% if kenny.sick %}
    Kenny is sick.
{% elseif kenny.dead %}
    You killed Kenny!  You bastard!!!
{% else %}
    Kenny looks okay --- so far
{% endif %}
```

Chapter 10

macro

Macros are comparable with functions in regular programming languages. They are useful to put often used HTML idioms into reusable elements to not repeat yourself.

Here is a small example of a macro that renders a form element:

Listing 10-1

```
{% macro input(name, value, type, size) %}
    <input type="{{ type|default('text') }}" name="{{ name }}" value="{{ value|e }}" size="{{
size|default(20) }}" />
{% endmacro %}
```

Macros differs from native PHP functions in a few ways:

- Default argument values are defined by using the `default` filter in the macro body;
- Arguments of a macro are always optional.

But as PHP functions, macros don't have access to the current template variables.

You can pass the whole context as an argument by using the special `_context` variable.

Macros can be defined in any template, and need to be "imported" before being used (see the documentation for the *import* tag for more information):

Listing 10-2

```
{% import "forms.html" as forms %}
```

The above `import` call imports the "forms.html" file (which can contain only macros, or a template and some macros), and import the functions as items of the `forms` variable.

The macro can then be called at will:

Listing 10-3

```
<p>{{ forms.input('username') }}</p>
<p>{{ forms.input('password', null, 'password') }}</p>
```

If macros are defined and used in the same template, you can use the special `_self` variable, without importing them:

Listing 10-4
```
<p>{{ _self.input('username') }}</p>
```

When you want to use a macro in another one from the same file, use the `_self` variable:

Listing 10-5
```
{% macro input(name, value, type, size) %}
  <input type="{{ type|default('text') }}" name="{{ name }}" value="{{ value|e }}" size="{{
size|default(20) }}" />
{% endmacro %}

{% macro wrapped_input(name, value, type, size) %}
    <div class="field">
        {{ _self.input(name, value, type, size) }}
    </div>
{% endmacro %}
```

When the macro is defined in another file, you need to import it:

Listing 10-6
```
{# forms.html #}

{% macro input(name, value, type, size) %}
  <input type="{{ type|default('text') }}" name="{{ name }}" value="{{ value|e }}" size="{{
size|default(20) }}" />
{% endmacro %}

{# shortcuts.html #}

{% macro wrapped_input(name, value, type, size) %}
    {% import "forms.html" as forms %}
    <div class="field">
        {{ forms.input(name, value, type, size) }}
    </div>
{% endmacro %}
```

from, import

Chapter 11

filter

Filter sections allow you to apply regular Twig filters on a block of template data. Just wrap the code in the special `filter` section:

Listing 11-1

```
{% filter upper %}
    This text becomes uppercase
{% endfilter %}
```

You can also chain filters:

Listing 11-2

```
{% filter lower|escape %}
    <strong>SOME TEXT</strong>
{% endfilter %}

{# outputs "&lt;strong&gt;some text&lt;/strong&gt;" #}
```

Chapter 12

set

Inside code blocks you can also assign values to variables. Assignments use the `set` tag and can have multiple targets:

Listing 12-1
```
{% set foo = 'foo' %}

{% set foo = [1, 2] %}

{% set foo = {'foo': 'bar'} %}

{% set foo = 'foo' ~ 'bar' %}

{% set foo, bar = 'foo', 'bar' %}
```

The `set` tag can also be used to 'capture' chunks of text:

Listing 12-2
```
{% set foo %}
  <div id="pagination">
    ...
  </div>
{% endset %}
```

If you enable automatic output escaping, Twig will only consider the content to be safe when capturing chunks of text.

Chapter 13

extends

The `extends` tag can be used to extend a template from another one.

Like PHP, Twig does not support multiple inheritance. So you can only have one extends tag called per rendering. However, Twig supports horizontal *reuse*.

Let's define a base template, `base.html`, which defines a simple HTML skeleton document:

Listing 13-1

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

In this example, the *block* tags define four blocks that child templates can fill in. All the `block` tag does is to tell the template engine that a child template may override those portions of the template.

Child Template

A child template might look like this:

Listing 13-2
```
{% extends "base.html" %}

{% block title %}Index{% endblock %}
{% block head %}
    {{ parent() }}
    <style type="text/css">
        .important { color: #336699; }
    </style>
{% endblock %}
{% block content %}
    <h1>Index</h1>
    <p class="important">
        Welcome on my awesome homepage.
    </p>
{% endblock %}
```

The `extends` tag is the key here. It tells the template engine that this template "extends" another template. When the template system evaluates this template, first it locates the parent. The extends tag should be the first tag in the template.

Note that since the child template doesn't define the `footer` block, the value from the parent template is used instead.

You can't define multiple `block` tags with the same name in the same template. This limitation exists because a block tag works in "both" directions. That is, a block tag doesn't just provide a hole to fill - it also defines the content that fills the hole in the *parent*. If there were two similarly-named `block` tags in a template, that template's parent wouldn't know which one of the blocks' content to use.

If you want to print a block multiple times you can however use the `block` function:

Listing 13-3
```
<title>{% block title %}{% endblock %}</title>
<h1>{{ block('title') }}</h1>
{% block body %}{% endblock %}
```

Parent Blocks

It's possible to render the contents of the parent block by using the *parent* function. This gives back the results of the parent block:

Listing 13-4
```
{% block sidebar %}
    <h3>Table Of Contents</h3>
    ...
    {{ parent() }}
{% endblock %}
```

Named Block End-Tags

Twig allows you to put the name of the block after the end tag for better readability:

Listing 13-5
```
{% block sidebar %}
    {% block inner_sidebar %}
        ...
    {% endblock inner_sidebar %}
{% endblock sidebar %}
```

Of course, the name after the `endblock` word must match the block name.

Block Nesting and Scope

Blocks can be nested for more complex layouts. Per default, blocks have access to variables from outer scopes:

```
{% for item in seq %}
    <li>{% block loop_item %}{{ item }}{% endblock %}</li>
{% endfor %}
```

Listing 13-6

Block Shortcuts

For blocks with few content, it's possible to use a shortcut syntax. The following constructs do the same:

```
{% block title %}
    {{ page_title|title }}
{% endblock %}
```

Listing 13-7

```
{% block title page_title|title %}
```

Listing 13-8

Dynamic Inheritance

Twig supports dynamic inheritance by using a variable as the base template:

```
{% extends some_var %}
```

Listing 13-9

If the variable evaluates to a `Twig_Template` object, Twig will use it as the parent template:

```
// {% extends layout %}

$layout = $twig->loadTemplate('some_layout_template.twig');

$twig->display('template.twig', array('layout' => $layout));
```

Listing 13-10

New in version 1.2: The possibility to pass an array of templates has been added in Twig 1.2.

You can also provide a list of templates that are checked for existence. The first template that exists will be used as a parent:

```
{% extends ['layout.html', 'base_layout.html'] %}
```

Listing 13-11

Conditional Inheritance

As the template name for the parent can be any valid Twig expression, it's possible to make the inheritance mechanism conditional:

```
{% extends standalone ? "minimum.html" : "base.html" %}
```

Listing 13-12

In this example, the template will extend the "minimum.html" layout template if the `standalone` variable evaluates to `true`, and "base.html" otherwise.

block, block, parent, use

Chapter 14

block

Blocks are used for inheritance and act as placeholders and replacements at the same time. They are documented in detail in the documentation for the *extends* tag.

Block names should consist of alphanumeric characters, and underscores. Dashes are not permitted.

block, parent, use, extends

Chapter 15

include

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

Listing 15-1
```
{% include 'header.html' %}
    Body
{% include 'footer.html' %}
```

Included templates have access to the variables of the active context.

If you are using the filesystem loader, the templates are looked for in the paths defined by it.

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

Listing 15-2
```
{# the foo template will have access to the variables from the current context and the foo one #}
{% include 'foo' with {'foo': 'bar'} %}

{% set vars = {'foo': 'bar'} %}
{% include 'foo' with vars %}
```

You can disable access to the context by appending the `only` keyword:

Listing 15-3
```
{# only the foo variable will be accessible #}
{% include 'foo' with {'foo': 'bar'} only %}
```

Listing 15-4
```
{# no variable will be accessible #}
{% include 'foo' only %}
```

When including a template created by an end user, you should consider sandboxing it. More information in the *Twig for Developers* chapter and in the *sandbox* tag documentation.

The template name can be any valid Twig expression:

Listing 15-5
```
{% include some_var %}
{% include ajax ? 'ajax.html' : 'not_ajax.html' %}
```

And if the expression evaluates to a `Twig_Template` object, Twig will use it directly:

Listing 15-6

```
// {% include template %}

$template = $twig->loadTemplate('some_template.twig');

$twig->loadTemplate('template.twig')->display(array('template' => $template));
```

New in version 1.2: The `ignore missing` feature has been added in Twig 1.2.

You can mark an include with `ignore missing` in which case Twig will ignore the statement if the template to be ignored does not exist. It has to be placed just after the template name. Here some valid examples:

Listing 15-7

```
{% include "sidebar.html" ignore missing %}
{% include "sidebar.html" ignore missing with {'foo': 'bar} %}
{% include "sidebar.html" ignore missing only %}
```

New in version 1.2: The possibility to pass an array of templates has been added in Twig 1.2.

You can also provide a list of templates that are checked for existence before inclusion. The first template that exists will be included:

Listing 15-8

```
{% include ['page_detailed.html', 'page.html'] %}
```

If `ignore missing` is given, it will fall back to rendering nothing if none of the templates exist, otherwise it will throw an exception.

Chapter 16

import

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. You can import the complete template into a variable or request specific macros from it.

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

Listing 16-1
```
{% macro input(name, value, type, size) %}
    <input type="{{ type|default('text') }}" name="{{ name }}" value="{{ value|e }}" size="{{
size|default(20) }}" />
{% endmacro %}

{% macro textarea(name, value, rows) %}
    <textarea name="{{ name }}" rows="{{ rows|default(10) }}" cols="{{ cols|default(40) }}">{{
value|e }}</textarea>
{% endmacro %}
```

The easiest and most flexible is importing the whole module into a variable. That way you can access the attributes:

Listing 16-2
```
{% import 'forms.html' as forms %}

<dl>
    <dt>Username</dt>
    <dd>{{ forms.input('username') }}</dd>
    <dt>Password</dt>
    <dd>{{ forms.input('password', null, 'password') }}</dd>
</dl>
<p>{{ forms.textarea('comment') }}</p>
```

Alternatively you can import names from the template into the current namespace:

Listing 16-3
```
{% from 'forms.html' import 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>
```

Importing is not needed if the macros and the template are defined in the same file; use the special `_self` variable instead:

Listing 16-4

```
{# index.html template #}

{% macro textarea(name, value, rows) %}
    <textarea name="{{ name }}" rows="{{ rows|default(10) }}" cols="{{ cols|default(40) }}">{{
value|e }}</textarea>
{% endmacro %}

<p>{{ _self.textarea('comment') }}</p>
```

But you can still create an alias by importing from the `_self` variable:

Listing 16-5

```
{# index.html template #}

{% macro textarea(name, value, rows) %}
    <textarea name="{{ name }}" rows="{{ rows|default(10) }}" cols="{{ cols|default(40) }}">{{
value|e }}</textarea>
{% endmacro %}

{% import _self as forms %}

<p>{{ forms.textarea('comment') }}</p>
```

macro, from

Chapter 17

`from`

The `from` tags import *macro* names into the current namespace. The tag is documented in detail in the documentation for the *import* tag.

macro, import

Chapter 18

use

New in version 1.1: Horizontal reuse was added in Twig 1.1.

Horizontal reuse is an advanced Twig feature that is hardly ever needed in regular templates. It is mainly used by projects that need to make template blocks reusable without using inheritance.

Template inheritance is one of the most powerful Twig's feature but it is limited to single inheritance; a template can only extend one other template. This limitation makes template inheritance simple to understand and easy to debug:

Listing 18-1

```
{% extends "base.html" %}

{% block title %}{% endblock %}
{% block content %}{% endblock %}
```

Horizontal reuse is a way to achieve the same goal as multiple inheritance, but without the associated complexity:

Listing 18-2

```
{% extends "base.html" %}

{% use "blocks.html" %}

{% block title %}{% endblock %}
{% block content %}{% endblock %}
```

The `use` statement tells Twig to import the blocks defined in `blocks.html` into the current template (it's like macros, but for blocks):

Listing 18-3

```
# blocks.html
{% block sidebar %}{% endblock %}
```

In this example, the `use` statement imports the `sidebar` block into the main template. The code is mostly equivalent to the following one (the imported blocks are not outputted automatically):

Listing 18-4
```
{% extends "base.html" %}

{% block sidebar %}{% endblock %}
{% block title %}{% endblock %}
{% block content %}{% endblock %}
```

The `use` tag only imports a template if it does not extend another template, if it does not define macros, and if the body is empty. But it can *use* other templates.

Because `use` statements are resolved independently of the context passed to the template, the template reference cannot be an expression.

The main template can also override any imported block. If the template already defines the `sidebar` block, then the one defined in `blocks.html` is ignored. To avoid name conflicts, you can rename imported blocks:

Listing 18-5
```
{% extends "base.html" %}

{% use "blocks.html" with sidebar as base_sidebar %}

{% block sidebar %}{% endblock %}
{% block title %}{% endblock %}
{% block content %}{% endblock %}
```

New in version 1.3: The `parent()` support was added in Twig 1.3.

The `parent()` function automatically determines the correct inheritance tree, so it can be used when overriding a block defined in an imported template:

Listing 18-6
```
{% extends "base.html" %}

{% use "blocks.html" %}

{% block sidebar %}
    {{ parent() }}
{% endblock %}

{% block title %}{% endblock %}
{% block content %}{% endblock %}
```

In this example, `parent()` will correctly call the `sidebar` block from the `blocks.html` template.

In Twig 1.2, renaming allows you to simulate inheritance by calling the "parent" block:

Listing 18-7
```
{% extends "base.html" %}
```

```
{% use "blocks.html" with sidebar as parent_sidebar %}

{% block sidebar %}
    {{ block('parent_sidebar') }}
{% endblock %}
```

You can use as many `use` statements as you want in any given template. If two imported templates define the same block, the latest one wins.

Chapter 19

spaceless

Use the `spaceless` tag to remove whitespace *between HTML tags*, not whitespace within HTML tags or whitespace in plain text:

Listing 19-1
```
{% spaceless %}
    <div>
        <strong>foo</strong>
    </div>
{% endspaceless %}

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

This tag is not meant to "optimize" the size of the generated HTML content but merely to avoid extra whitespace between HTML tags to avoid browser rendering quirks under some circumstances.

If you want to optimize the size of the generated HTML content, gzip compress the output instead.

If you want to create a tag that actually removes all extra whitespace in an HTML string, be warned that this is not as easy as it seems to be (think of `textarea` or `pre` tags for instance). Using a third-party library like Tidy is probably a better idea.

For more information on whitespace control, read the *dedicated* section of the documentation and learn how you can also use the whitespace control modifier on your tags.

Chapter 20

autoescape

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:

Listing 20-1

```
{# The following syntax works as of Twig 1.8 -- see the note below for previous versions #}

{% autoescape %}
    Everything will be automatically escaped in this block
    using the HTML strategy
{% endautoescape %}

{% autoescape 'html' %}
    Everything will be automatically escaped in this block
    using the HTML strategy
{% endautoescape %}

{% autoescape 'js' %}
    Everything will be automatically escaped in this block
    using the js escaping strategy
{% endautoescape %}

{% autoescape false %}
    Everything will be outputted as is in this block
{% endautoescape %}
```

Before Twig 1.8, the syntax was different:

Listing 20-2

```
{% autoescape true %}
    Everything will be automatically escaped in this block
    using the HTML strategy
{% endautoescape %}

{% autoescape false %}
    Everything will be outputted as is in this block
{% endautoescape %}
```

```
{% autoescape true js %}
    Everything will be automatically escaped in this block
    using the js escaping strategy
{% endautoescape %}
```

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 *raw* filter:

Listing 20-3

```
{% autoescape %}
    {{ safe_value|raw }}
{% endautoescape %}
```

Functions returning template data (like *macros* and *parent*) always return safe markup.

Twig is smart enough to not escape an already escaped value by the *escape* filter.

The chapter *Twig for Developers* gives more information about when and how automatic escaping is applied.

Chapter 21

raw

The `raw` tag marks sections as being raw text that should not be parsed. For example to put Twig syntax as example into a template you can use this snippet:

Listing 21-1

```
{% raw %}
    <ul>
    {% for item in seq %}
        <li>{{ item }}</li>
    {% endfor %}
    </ul>
{% endraw %}
```

Chapter 22

flush

New in version 1.5: The flush tag was added in Twig 1.5.

The `flush` tag tells Twig to flush the output buffer:

Listing 22-1
```
{% flush %}
```

Internally, Twig uses the PHP *flush*[1] function.

1. `http://php.net/flush`

Chapter 23

do

New in version 1.5: The do tag was added in Twig 1.5.

The `do` tag works exactly like the regular variable expression (`{{ ... }}`) just that it doesn't print anything:

```
{% do 1 + 2 %}
```

Listing 23-1

Chapter 24

sandbox

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:

Listing 24-1
```
{% sandbox %}
    {% include 'user.html' %}
{% endsandbox %}
```

The `sandbox` tag is only available when the sandbox extension is enabled (see the *Twig for Developers* chapter).

The `sandbox` tag can only be used to sandbox an include tag and it cannot be used to sandbox a section of a template. The following example won't work for example:

Listing 24-2
```
{% sandbox %}
    {% for i in 1..2 %}
        {{ i }}
    {% endfor %}
{% endsandbox %}
```

Chapter 25

embed

New in version 1.8: The `embed` tag was added in Twig 1.8.

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.

Think of an embedded template as a "micro layout skeleton".

Listing 25-1

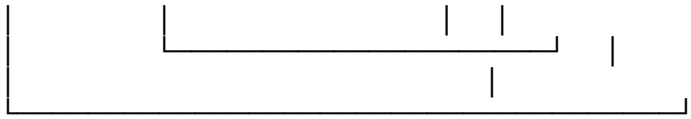
```
{% embed "teasers_skeleton.twig" %}
    {# These blocks are defined in "teasers_skeleton.twig" #}
    {# and we override them right here:                    #}
    {% block left_teaser %}
        Some content for the left teaser box
    {% endblock %}
    {% block right_teaser %}
        Some content for the right teaser box
    {% endblock %}
{% endembed %}
```

The `embed` tag takes the idea of template inheritance to the level of content fragments. While template inheritance allows for "document skeletons", which are filled with life by child templates, the `embed` tag allows you to create "skeletons" for smaller units of content and re-use and fill them anywhere you like.

Since the use case may not be obvious, let's look at a simplified example. Imagine a base template shared by multiple HTML pages, defining a single block named "content":

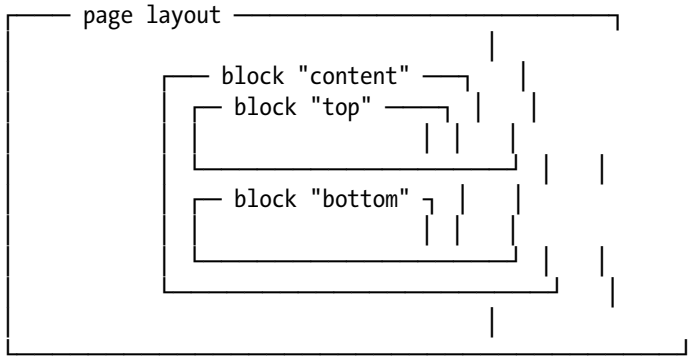
Listing 25-2

```
┌─── page layout ──────────────────────────┐
│                                          │
│           ┌── block "content" ──┐        │
│           │                     │        │
│           │                     │        │
│           │ (child template to  │        │
│           │  put content here)  │        │
│           │                     │        │
```



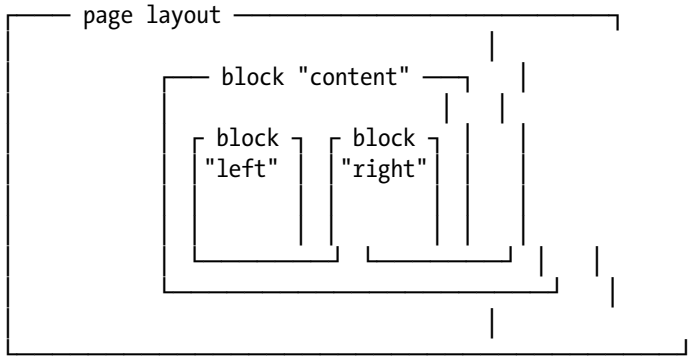
Some pages ("foo" and "bar") share the same content structure - two vertically stacked boxes:

Listing 25-3



While other pages ("boom" and "baz") share a different content structure - two boxes side by side:

Listing 25-4



Without the `embed` tag, you have two ways to design your templates:

- Create two "intermediate" base templates that extend the master layout template: one with vertically stacked boxes to be used by the "foo" and "bar" pages and another one with side-by-side boxes for the "boom" and "baz" pages.
- Embed the markup for the top/bottom and left/right boxes into each page template directly.

These two solutions do not scale well because they each have a major drawback:

- The first solution may indeed work for this simplified example. But imagine we add a sidebar, which may again contain different, recurring structures of content. Now we would need to create intermediate base templates for all occurring combinations of content structure and sidebar structure... and so on.
- The second solution involves duplication of common code with all its negative consequences: any change involves finding and editing all affected copies of the structure, correctness has to be verified for each copy, copies may go out of sync by careless modifications etc.

In such a situation, the `embed` tag comes in handy. The common layout code can live in a single base template, and the two different content structures, let's call them "micro layouts" go into separate templates which are embedded as necessary:

Page template `foo.twig`:

Listing 25-5

```
{% extends "layout_skeleton.twig" %}

{% block content %}
    {% embed "vertical_boxes_skeleton.twig" %}
        {% block top %}
            Some content for the top box
        {% endblock %}

        {% block bottom %}
            Some content for the bottom box
        {% endblock %}
    {% endembed %}
{% endblock %}
```

And here is the code for `vertical_boxes_skeleton.twig`:

Listing 25-6

```
<div class="top_box">
    {% block top %}
        Top box default content
    {% endblock %}
</div>

<div class="bottom_box">
    {% block bottom %}
        Bottom box default content
    {% endblock %}
</div>
```

The goal of the `vertical_boxes_skeleton.twig` template being to factor out the HTML markup for the boxes.

The `embed` tag takes the exact same arguments as the `include` tag:

Listing 25-7

```
{% embed "base" with {'foo': 'bar'} %}
    ...
{% endembed %}

{% embed "base" with {'foo': 'bar'} only %}
    ...
{% endembed %}

{% embed "base" ignore missing %}
    ...
{% endembed %}
```

As embedded templates do not have "names", auto-escaping strategies based on the template "filename" won't work as expected if you change the context (for instance, if you embed a CSS/JavaScript template into an HTML one). In that case, explicitly set the default auto-escaping strategy with the `autoescape` tag.

include

Chapter 26

date

New in version 1.1: The timezone support has been added in Twig 1.1.

New in version 1.5: The default date format support has been added in Twig 1.5.

New in version 1.6.1: The default timezone support has been added in Twig 1.6.1.

The `date` filter formats a date to a given format:

Listing 26-1
```
{{ post.published_at|date("m/d/Y") }}
```

The `date` filter accepts strings (it must be in a format supported by the *date*[1] function), *DateTime*[2] instances, or *DateInterval*[3] instances. For instance, to display the current date, filter the word "now":

Listing 26-2
```
{{ "now"|date("m/d/Y") }}
```

To escape words and characters in the date format use \\ in front of each character:

Listing 26-3
```
{{ post.published_at|date("F jS \\a\\t g:ia") }}
```

You can also specify a timezone:

1. http://www.php.net/date
2. http://www.php.net/DateTime
3. http://www.php.net/DateInterval

```
{{ post.published_at|date("m/d/Y", "Europe/Paris") }}
```
Listing 26-4

If no format is provided, Twig will use the default one: `F j, Y H:i`. This default can be easily changed by calling the `setDateFormat()` method on the `core` extension instance. The first argument is the default format for dates and the second one is the default format for date intervals:

```
$twig = new Twig_Environment($loader);
$twig->getExtension('core')->setDateFormat('d/m/Y', '%d days');
```
Listing 26-5

The default timezone can also be set globally by calling `setTimezone()`:

```
$twig = new Twig_Environment($loader);
$twig->getExtension('core')->setTimezone('Europe/Paris');
```
Listing 26-6

If the value passed to the `date` filter is null, it will return the current date by default. If an empty string is desired instead of the current date, use a ternary operator:

```
{{ post.published_at is empty ? "" : post.published_at|date("m/d/Y") }}
```
Listing 26-7

Chapter 27

format

The `format` filter formats a given string by replacing the placeholders (placeholders follows the *printf*[1] notation):

Listing 27-1
```
{{ "I like %s and %s."|format(foo, "bar") }}

{# returns I like foo and bar
   if the foo parameter equals to the foo string. #}
```

replace

1. http://www.php.net/printf

Chapter 28

replace

The `replace` filter formats a given string by replacing the placeholders (placeholders are free-form):

```
{{ "I like %this% and %that%."|replace({'%this%': foo, '%that%': "bar"}) }}

{# returns I like foo and bar
   if the foo parameter equals to the foo string. #}
```

Listing 28-1

format

Chapter 29

number_format

New in version 1.5: The number_format filter was added in Twig 1.5

The `number_format` filter formats numbers. It is a wrapper around PHP's *number_format*[1] function:

Listing 29-1
```
{{ 200.35|number_format }}
```

You can control the number of decimal places, decimal point, and thousands separator using the additional arguments:

Listing 29-2
```
{{ 9800.333|number_format(2, ',', '.') }}
```

If no formatting options are provided then Twig will use the default formatting options of:

- 0 decimal places.
- `.` as the decimal point.
- `,` as the thousands separator.

These defaults can be easily changed through the core extension:

Listing 29-3
```
$twig = new Twig_Environment($loader);
$twig->getExtension('core')->setNumberFormat(3, ',', '.');
```

The defaults set for `number_format` can be over-ridden upon each call using the additional parameters.

1. http://php.net/number_format

Chapter 30

url_encode

The `url_encode` filter URL encodes a given string:

```
{{ data|url_encode() }}
```

Listing 30-1

Internally, Twig uses the PHP *urlencode*[1] function.

1. `http://php.net/urlencode`

Chapter 31

json_encode

The `json_encode` filter returns the JSON representation of a string:

Listing 31-1

```
{{ data|json_encode() }}
```

Internally, Twig uses the PHP *json_encode*[1] function.

1. http://php.net/json_encode

Chapter 32
convert_encoding

New in version 1.4: The `convert_encoding` filter was added in Twig 1.4.

The `convert_encoding` filter converts a string from one encoding to another. The first argument is the expected output charset and the second one is the input charset:

```
{{ data|convert_encoding('UTF-8', 'iso-2022-jp') }}
```

Listing 32-1

This filter relies on the *iconv*[1] or *mbstring*[2] extension, so one of them must be installed. In case both are installed, *mbstring*[3] is used by default (Twig before 1.8.1 uses *iconv*[4] by default).

1. http://php.net/iconv
2. http://php.net/mbstring
3. http://php.net/mbstring
4. http://php.net/iconv

Chapter 33

`title`

The `title` filter returns a titlecased version of the value. Words will start with uppercase letters, all remaining characters are lowercase:

Listing 33-1

```
{{ 'my first car'|title }}

{# outputs 'My First Car' #}
```

Chapter 34
capitalize

The `capitalize` filter capitalizes a value. The first character will be uppercase, all others lowercase:

Listing 34-1

```
{{ 'my first car'|capitalize }}

{# outputs 'My first car' #}
```

Chapter 35

nl2br

New in version 1.5: The nl2br filter was added in Twig 1.5.

The `nl2br` filter inserts HTML line breaks before all newlines in a string:

Listing 35-1
```
{{ "I like Twig.\nYou will like it too."|nl2br }}
{# outputs

    I like Twig.<br />
    You will like it too.

#}
```

The `nl2br` filter pre-escapes the input before applying the transformation.

Chapter 36

upper

The `upper` filter converts a value to uppercase:

Listing 36-1

```
{{ 'welcome'|upper }}

{# outputs 'WELCOME' #}
```

Chapter 37

lower

The `lower` filter converts a value to lowercase:

Listing 37-1

```
{{ 'WELCOME'|lower }}

{# outputs 'welcome' #}
```

Chapter 38
striptags

The `striptags` filter strips SGML/XML tags and replace adjacent whitespace by one space:

```
{% some_html|striptags %}
```

Listing 38-1

Internally, Twig uses the PHP *strip_tags*[1] function.

1. `http://php.net/strip_tags`

Chapter 39

join

The `join` filter returns a string which is the concatenation of the items of a sequence:

Listing 39-1
```
{{ [1, 2, 3]|join }}
{# returns 123 #}
```

The separator between elements is an empty string per default, but you can define it with the optional first parameter:

Listing 39-2
```
{{ [1, 2, 3]|join('|') }}
{# returns 1|2|3 #}
```

Chapter 40

reverse

New in version 1.6: Support for strings has been added in Twig 1.6.

The `reverse` filter reverses a sequence, a mapping, or a string:

Listing 40-1

```
{% for use in users|reverse %}
    ...
{% endfor %}

{{ '1234'|reverse }}

{# outputs 4321 #}
```

It also works with objects implementing the *Traversable*[1] interface.

1. http://php.net/Traversable

Chapter 41

`length`

The `length` filters returns the number of items of a sequence or mapping, or the length of a string:

Listing 41-1

```
{% if users|length > 10 %}
    ...
{% endif %}
```

Chapter 42

sort

The `sort` filter sorts an array:

```
{% for use in users|sort %}
    ...
{% endfor %}
```

Listing 42-1

Internally, Twig uses the PHP *asort*[1] function to maintain index association.

1. `http://php.net/asort`

Chapter 43

`default`

The `default` filter returns the passed default value if the value is undefined or empty, otherwise the value of the variable:

Listing 43-1

```
{{ var|default('var is not defined') }}

{{ var.foo|default('foo item on var is not defined') }}

{{ var['foo']|default('foo item on var is not defined') }}

{{ ''|default('passed var is empty')  }}
```

When using the `default` filter on an expression that uses variables in some method calls, be sure to use the `default` filter whenever a variable can be undefined:

Listing 43-2

```
{{ var.method(foo|default('foo'))|default('foo') }}
```

Read the documentation for the *defined* and *empty* tests to learn more about their semantics.

Chapter 44

keys

The keys filter returns the keys of an array. It is useful when you want to iterate over the keys of an array:

```
{% for key in array|keys %}
    ...
{% endfor %}
```

Listing 44-1

Chapter 45

escape

The `escape` filter converts the characters `&`, `<`, `>`, `'`, and `"` in strings to HTML-safe sequences. Use this if you need to display text that might contain such characters in HTML:

Listing 45-1
```
{{ user.username|escape }}
```

For convenience, the `e` filter is defined as an alias:

Listing 45-2
```
{{ user.username|e }}
```

The `escape` filter can also be used in other contexts than HTML thanks to an optional argument which defines the escaping strategy to use:

Listing 45-3
```
{{ user.username|e }}
{# is equivalent to #}
{{ user.username|e('html') }}
```

And here is how to escape variables included in JavaScript code:

Listing 45-4
```
{{ user.username|escape('js') }}
{{ user.username|e('js') }}
```

Internally, `escape` uses the PHP native *htmlspecialchars*[1] function for the HTML escaping strategy.

1. `http://php.net/htmlspecialchars`

Chapter 46

raw

The `raw` filter marks the value as being "safe", which means that in an environment with automatic escaping enabled this variable will not be escaped if `raw` is the last filter applied to it:

Listing 46-1

```
{% autoescape true %}
    {{ var|raw }} {# var won't be escaped #}
{% endautoescape %}
```

Chapter 47

merge

The `merge` filter merges an array with the another array:

Listing 47-1
```
{% set values = [1, 2] %}

{% set values = values|merge(['apple', 'orange']) %}

{# values now contains [1, 2, 'apple', 'orange'] #}
```

New values are added at the end of the existing ones.

The `merge` filter also works on hashes:

Listing 47-2
```
{% set items = { 'apple': 'fruit', 'orange': 'fruit', 'peugeot': 'unknown' } %}

{% set items = items|merge({ 'peugeot': 'car', 'renault': 'car' }) %}

{# items now contains { 'apple': 'fruit', 'orange': 'fruit', 'peugeot': 'car', 'renault':
'car' } #}
```

For hashes, the merging process occurs on the keys: if the key does not already exist, it is added but if the key already exists, its value is overridden.

If you want to ensure that some values are defined in an array (by given default values), reverse the two elements in the call:

Listing 47-3
```
{% set items = { 'apple': 'fruit', 'orange': 'fruit' } %}

{% set items = { 'apple': 'unknown' }|merge(items) %}

{# items now contains { 'apple': 'fruit', 'orange': 'fruit' } #}
```

Chapter 48

slice

New in version 1.6: The slice filter was added in Twig 1.6.

The `slice` filter extracts a slice of a sequence, a mapping, or a string:

Listing 48-1

```
{% for i in [1, 2, 3, 4]|slice(1, 2) %}
    {# will iterate over 2 and 3 #}
{% endfor %}

{{ '1234'|slice(1, 2) }}

{# outputs 23 #}
```

You can use any valid expression for both the start and the length:

Listing 48-2

```
{% for i in [1, 2, 3, 4]|slice(start, length) %}
    {# ... #}
{% endfor %}
```

As syntactic sugar, you can also use the `[]` notation:

Listing 48-3

```
{% for i in [1, 2, 3, 4][start:length] %}
    {# ... #}
{% endfor %}

{{ '1234'[1:2] }}
```

The `slice` filter works as the *array_slice*[1] PHP function for arrays and *substr*[2] for strings.

If the start is non-negative, the sequence will start at that start in the variable. If start is negative, the sequence will start that far from the end of the variable.

1. http://php.net/array_slice
2. http://php.net/substr

If length is given and is positive, then the sequence will have up to that many elements in it. If the variable is shorter than the length, then only the available variable elements will be present. If length is given and is negative then the sequence will stop that many elements from the end of the variable. If it is omitted, then the sequence will have everything from offset up until the end of the variable.

It also works with objects implementing the *Traversable*[3] interface.

3. http://php.net/manual/en/class.traversable.php

Chapter 49

trim

New in version 1.6.2: The trim filter was added in Twig 1.6.2.

The `trim` filter strips whitespace (or other characters) from the beginning and end of a string:

Listing 49-1

```
{{ '  I like Twig.  '|trim }}

{# outputs 'I like Twig.' #}

{{ '  I like Twig.'|trim('.') }}

{# outputs '  I like Twig' #}
```

Internally, Twig uses the PHP *trim*[1] function.

1. http://php.net/trim

Chapter 50

range

Returns a list containing an arithmetic progression of integers:

Listing 50-1
```
{% for i in range(0, 3) %}
    {{ i }},
{% endfor %}

{# returns 0, 1, 2, 3 #}
```

When step is given (as the third parameter), it specifies the increment (or decrement):

Listing 50-2
```
{% for i in range(0, 6, 2) %}
    {{ i }},
{% endfor %}

{# returns 0, 2, 4, 6 #}
```

The Twig built-in `..` operator is just syntactic sugar for the `range` function (with a step of 1):

Listing 50-3
```
{% for i in 0..3 %}
    {{ i }},
{% endfor %}
```

The `range` function works as the native PHP *range*[1] function.

1. `http://php.net/range`

Chapter 51

cycle

The `cycle` function cycles on an array of values:

```
{% for i in 0..10 %}
    {{ cycle(['odd', 'even'], i) }}
{% endfor %}
```

Listing 51-1

The array can contain any number of values:

```
{% set fruits = ['apple', 'orange', 'citrus'] %}

{% for i in 0..10 %}
    {{ cycle(fruits, i) }}
{% endfor %}
```

Listing 51-2

Chapter 52

constant

`constant` returns the constant value for a given string:

Listing 52-1

```
{{ some_date|date(constant('DATE_W3C')) }}
{{ constant('Namespace\\Classname::CONSTANT_NAME') }}
```

Chapter 53

random

New in version 1.5: The random function was added in Twig 1.5.

New in version 1.6: String and integer handling was added in Twig 1.6.

The `random` function returns a random value depending on the supplied parameter type:

- a random item from a sequence;
- a random character from a string;
- a random integer between 0 and the integer parameter (inclusive).

Listing 53-1

```
{{ random(['apple', 'orange', 'citrus']) }} {# example output: orange #}
{{ random('ABC') }}                         {# example output: C #}
{{ random() }}                              {# example output: 15386094 (works as native PHP
`mt_rand`_ function) #}
{{ random(5) }}                             {# example output: 3 #}
```

Chapter 54

attribute

New in version 1.2: The `attribute` function was added in Twig 1.2.

`attribute` can be used to access a "dynamic" attribute of a variable:

Listing 54-1

```
{{ attribute(object, method) }}
{{ attribute(object, method, arguments) }}
{{ attribute(array, item) }}
```

The resolution algorithm is the same as the one used for the `.` notation, except that the item can be any valid expression.

Chapter 55

block

When a template uses inheritance and if you want to print a block multiple times, use the `block` function:

```
<title>{% block title %}{% endblock %}</title>

<h1>{{ block('title') }}</h1>

{% block body %}{% endblock %}
```

Listing 55-1

extends, parent

Chapter 56

parent

When a template uses inheritance, it's possible to render the contents of the parent block when overriding a block by using the `parent` function:

Listing 56-1

```
{% extends "base.html" %}

{% block sidebar %}
    <h3>Table Of Contents</h3>
    ...
    {{ parent() }}
{% endblock %}
```

The `parent()` call will return the content of the `sidebar` block as defined in the `base.html` template.

extends, block, block

Chapter 57

dump

New in version 1.5: The dump function was added in Twig 1.5.

The `dump` function dumps information about a template variable. This is mostly useful to debug a template that does not behave as expected by introspecting its variables:

```
{{ dump(user) }}
```

Listing 57-1

The `dump` function is not available by default. You must add the `Twig_Extension_Debug` extension explicitly when creating your Twig environment:

```
$twig = new Twig_Environment($loader, array(
    'debug' => true,
    // ...
));
$twig->addExtension(new Twig_Extension_Debug());
```

Listing 57-2

Even when enabled, the `dump` function won't display anything if the `debug` option on the environment is not enabled (to avoid leaking debug information on a production server).

In an HTML context, wrap the output with a `pre` tag to make it easier to read:

```
<pre>
    {{ dump(user) }}
</pre>
```

Listing 57-3

Using a `pre` tag is not needed when *XDebug*[1] is enabled and `html_errors` is `on`; as a bonus, the output is also nicer with XDebug enabled.

You can debug several variables by passing them as additional arguments:

Listing 57-4
```
{{ dump(user, categories) }}
```

If you don't pass any value, all variables from the current context are dumped:

Listing 57-5
```
{{ dump() }}
```

Internally, Twig uses the PHP *var_dump*[2] function.

1. http://xdebug.org/docs/display
2. http://php.net/var_dump

Chapter 58

date

New in version 1.6: The date function has been added in Twig 1.6.

New in version 1.6.1: The default timezone support has been added in Twig 1.6.1.

Converts an argument to a date to allow date comparison:

```
{% if date(user.created_at) < date('+2days') %}
    {# do something #}
{% endif %}
```

Listing 58-1

The argument must be in a format supported by the *date*[1] function.

You can pass a timezone as the second argument:

```
{% if date(user.created_at) < date('+2days', 'Europe/Paris') %}
    {# do something #}
{% endif %}
```

Listing 58-2

If no argument is passed, the function returns the current date:

```
{% if date(user.created_at) < date() %}
    {# always! #}
{% endif %}
```

Listing 58-3

1. `http://www.php.net/date`

You can set the default timezone globally by calling `setTimezone()` on the `core` extension instance:

Listing 58-4
```
$twig = new Twig_Environment($loader);
$twig->getExtension('core')->setTimezone('Europe/Paris');
```

Chapter 59

divisibleby

`divisibleby` checks if a variable is divisible by a number:

```
{% if loop.index is divisibleby(3) %}
    ...
{% endif %}
```

Listing 59-1

Chapter 60

`null`

`null` returns `true` if the variable is `null`:

Listing 60-1

```
{{ var is null }}
```

`none` is an alias for `null`.

Chapter 61

even

`even` returns `true` if the given number is even:

```
{{ var is even }}
```

Listing 61-1

odd

Chapter 62

odd

`odd` returns `true` if the given number is odd:

Listing 62-1

```
{{ var is odd }}
```

```
    even
```

Chapter 63

sameas

`sameas` checks if a variable points to the same memory address than another variable:

Listing 63-1

```
{% if foo.attribute is sameas(false) %}
    the foo attribute really is the ``false`` PHP value
{% endif %}
```

Chapter 64

constant

`constant` checks if a variable has the exact same value as a constant. You can use either global constants or class constants:

Listing 64-1

```
{% if post.status is constant('Post::PUBLISHED') %}
    the status attribute is exactly the same as Post::PUBLISHED
{% endif %}
```

Chapter 65

defined

`defined` checks if a variable is defined in the current context. This is very useful if you use the `strict_variables` option:

Listing 65-1

```
{# defined works with variable names #}
{% if foo is defined %}
    ...
{% endif %}

{# and attributes on variables names #}
{% if foo.bar is defined %}
    ...
{% endif %}

{% if foo['bar'] is defined %}
    ...
{% endif %}
```

When using the `defined` test on an expression that uses variables in some method calls, be sure that they are all defined first:

Listing 65-2

```
{% if var is defined and foo.method(var) is defined %}
    ...
{% endif %}
```

Chapter 66

empty

`empty` checks if a variable is empty:

Listing 66-1
```
{# evaluates to true if the foo variable is null, false, or the empty string #}
{% if foo is empty %}
    ...
{% endif %}
```

Chapter 67

iterable

New in version 1.7: The iterable test was added in Twig 1.7.

`iterable` checks if a variable is an array or a traversable object:

Listing 67-1

```
{# evaluates to true if the foo variable is iterable #}
{% if users is iterable %}
    {% for user in users %}
        Hello {{ user }}!
    {% endfor %}
{% else %}
    {# users is probably a string #}
    Hello {{ users }}!
{% endif %}
```