Jtwig Reference Manual

Jtwig

Published with GitBook

Table of Contents

Glossary

Preface

Welcome to the Jtwig Reference Manual. This book was created to serve as documentation support for Jtwig users. Although Jtwig is based on Twig, an already well established technology and heavily used by thousands of developers, studying the differences between Jtwig and Twig, also as, the Jtwig differences against other templating engines will allow one to learn about the advantages and disadvantages of using Jtwig, specially, how can such choice affect a project.

Audience

This book can be read by anyone knowledgeable about web development. Java language is also a requirement, but only for advanced chapters of this book.

Template Engines

To better understand the foundations of template engines it is best to start by clarifying what they are and what they do. What do projects benefit from using them and how do they affect projects, from the system design to its maintainability.

What is a template engine?

A template engine is a piece of software designed to combine a template with data in order to produce content. A template is an intermediate representation of the content. It specifies rules defining on how the output will be generated.

Let's take a look to one of the simplest template engines - the Java native `String.format`.

```
String.format("Hello, I am %d years old", 30);
```

Yes, String.format can be seen as a template engine, a really simple one. In the previous example, the data is the integer `30` which, when combined with the template `Hello, I am %d years old`, generates the output `Hello, I am 30 years old`.

Why template engines?

Because they optimize the process of producing content by, mainly, avoiding repetition. In essence, a template engine allows one to reuse the same template with different data generating therefore different content. It splits the data from its presentation, allowing to develop the presentation and logic layer almost independently, where the only dependency is the data (also named data model).

Splitting this layers eases maintenance by increasing the application modularity and changeability. For example, during the initial stages of a project, both layers might have similar change rates, however, when the project enters the maintenance mode, the presentation layer tends to be more subject to change. Without a template engine, it would be required logic layer changes in order to affect the presentation.

Another reason to adopt template engines, as tools written specifically for presentation logic development, it is because they simplify the development by providing several useful functionalities.

Why Jtwig?

Jtwig appears from the desire of a better template engine in the Java world. After investigating all technologies available in the market, one decided to, instead of creating a brand new technology, port one good template engine to the Java world, mainly because one could only benefit from an already mature technology, heavily used and proven in the wild. Twig which is based on Django templates itself, was the choice. It was seen as the best alternative given the following arguments:

- Code island based syntax, making it easier to read when mixed with presentation content;
- Small learning curve, easy and common syntactic constructs, in fact, for Java developers, Jtwig represents a small deviation from Java syntax;
- Simple yet powerful modularization mechanism;
- Great expressive power with the capability to create logical complex templates;

Syntax

Jtwig is a Twig port to Java, however, it does not support all Twig features and also includes functionality Twig does not offer. Syntax wise, they do also differ. This difference is due to Java environment specifics. To allow the engine to take full advantage of its mother language, some fundamental changes were introduced. Within this section one will detail Jtwig syntax, making a comparison, whenever sensible, with Twig.

Code Islands

As a template language, Jtwig allows one to have formatting logic around content. In order to do so, Jtwig uses the so called Code Islands. This is similar to many template languages and identical to Twig's syntax.

```
{% ... Jtwig code here ... %}
```

Jtwig code islands begin with `{%` and ends with `%}`, just as simple as that.

White space control

Tied up with the code island syntax is another syntactical enabled feature, 100% compatible with Twig, the white space control functionality. This allows one to remove white spaces before or after a code island. In order to accomplish that, append/prepend the symbol `-` to the beginning/ending of the Jtwig code island. Let's take for example:

```
{% ... jtwig code ... -%} text {%- ... jtwig code ... %}
```

Such code will produce `text` without white spaces.

Comments

In Jtwig there are only multiline comments.

```
{# This is the content of the comment.
    And it can be a multi-line content.  #}
```

Note that comments also support the white space control feature, using the same approach as per code islands.

```
{#- Comment... -#}
```

Output

In Jtwig there is only one way to write the value of an expression (the definition of expression will be detailed afterwards) to the output. That is accomplished with the print operation as shown below.

```
{{ expression... }}
```

Output constructs also support white space control feature in the same way as code islands.

```
{{- expression... -}}
```

Expressions

Expressions are the logic building blocks, they allow you to express a value. From the basic constants to binary and ternary operations, expressions give developers the power to specify a value.

Literals

The most basic expressions, check the table below.

Name	Regular Expression
Float	`-?[0-9]+'.'[0-9]+`
Integer	`-?[0-9]+`
Null Reference	`null`
Boolean	`true` or `false`
String	`"[^"]*"` or `'[^']*'`
Character	`'[a-zA-Z]'`

Identifier

An identifier is a name in Jtwig. It can contain alphanumeric characters and also underscores, but cannot start with digits. This is exactly the same regular expression as Java identifiers.

```
[a-zA-Z_$][a-zA-Z0-9_$]*
```

Of course such similarity happens on purpose, making it possible to specify any Java identifier in Jtwig templates. Such capability enables Jtwig to be fully compatible with Java identifiers, an important feature, specially, when it comes to the selection operator (as explained later on). This is one of the fundamental differences between Twig and Jtwig. Their regular expressions for identifiers are different and they match exactly the expression defined by their mother language. However different, they are quite similar. As per PHP official documentation PHP Manual, identifiers are defined according to the following regular expression:

```
[a-zA-Z_\x7f-\xff][a-zA-Z0-9_\x7f-\xff]*
```

PHP definition is very similar to Java's one, apart from the `$` character, however it also includes bytes from 127 to 225 (\x7f-\xff). Depending on the encoding, these extra bytes might be latin characters or even punctuation.

This allows one to create Jtwig and Twig compatible templates, nevertheless, it is important to be aware of such differences in order to accurately measure compatibility.

List

There are two ways to specify a list in Jtwig. It can be either by enumeration or by comprehension. To specify a list by enumeration, one basically need to provide every single element of the list, as shown below.

```
[1, 2, 3]
```

To specify a list by comprehension, however, one just need to specify both the beginning and the ending elements of the list, Jtwig engine, will then expand the definition. Currently, it only supports Integers or Characters.

```
1..3
```

Both lists, exemplified above, produce the same output.

Maps

It is also possible to represent collections of key and value pairs in Jtwig, so called maps, where keys can either be Identifiers or Strings and values can take form of any kind of expression.

```
{ key1: 'value1', 'key 2': 'value2' }
```

Keep in mind that identifiers used to represent the key elements are not used as variable placeholders, instead, they are converted to their String representation. For example, the identifier `key1` shown above will be converted to the String value `"key1"`.

List or Map Value Access

Whenever one need to access either a list element or map value, Jtwig, like Twig, comes with the value access expression. With such expression one can access the value given the key. Note that, for lists, the key is the position of the element in the list starting at zero.

```
list[1]
map[“hello”]
```

Keywords

Jtwig has some reserved identifiers, such identifiers are the building blocks of more complex Jtwig constructs. Below there is the list of the native reserved identifiers.

`include` `set` `block` `endblock` `if` `endif` `elseif` `else` `for` `endfor` `import` `macro` `endmacro` `extends` `embed` `endembed` `true` `false` `in` `as` `autoescape` `endautoescape` `do` `flush` `verbatim` `endverbatim` `spaceless` `endspaceless` `filter` `endfilter` `null` `is` `not` `with`

Note that Jtwig is extensible and the list of keywords might be affected by such extensions.

Unary Operators

Unary operators by definition only need one argument, Jtwig comes with only two built in unary operators, they are `not` and `-`.

Operator	Symbol	P*	Description	Example
Negative	`-`	5	Switches the signal.	`-(-1)` outputs `1`
Not	`not`	10	Negates the input.	`not false` outputs `true` </br> `not true` outputs `false`

* Precedence order, the lower the precedence, the higher the priority.

Binary Operators

Jtwig comes with several built in binary operators. Below, one will describe the entire list of built in binary operators.

Operator	Symbol	P	Description	Example
Selection	`.`	1	Access inner properties of objects.	`[1, 2].size` outputs `2`
Multiply	`*`	5	Multiplies two values.	`2.2 * 2.2` outputs `4.4`
Integer Multiply	`**`	5	Multiplies the integer part of two values.	`2.2 * 2.2` outputs `4`
Divide	`/`	5	Divides two values.	`2.2 * 2.2` outputs `1.1`
Integer			Divides the integer part	`2.2 * 2.2` outputs `1`

Divide			of two values.	`2.2 * 2.2` outputs `1`
Remainder	`%`	5	Gets the integer division remainder.	`5 % 2` outputs `1`
Sum	`+`	10	Sums two values.	`5 + 2` outputs `7`
Subtract	`-`	10	Subtracts two values.	`5 - 2` outputs `3`
Concat	`~`	10	Concatenates two strings.	`"5" ~ "2"` outputs `"52"`
Less	`<`	15	Compares two values, checking whether the first is lower than the second.	`1 < 2` outputs `true` `1 < 1` outputs `false`
Less or equal	`<=`	15	Compares two values, checking whether the first is lower or equal than the second.	`2 <= 2` outputs `true` `2 < 1` outputs `false`
Greater	`>`	15	Compares two values, checking whether the first is higher than the second.	`2 > 1` outputs `true` `2 > 2` outputs `false`
Greater or equal	`>=`	15	Compares two values, checking whether the first is higher or equal than the second.	`2 >= 2` outputs `true` `2 >= 3` outputs `false`
Contains	`in`	15	Checks whether the second value contains the first one.	`5 in [2]` outputs `false`
Equivalent	`==`	20	Compares two values, checking whether they are equal or not.	`true == false` outputs `false` `false == false` outputs `true`
Different	`!=`	20	Compares two values, checking whether they are different or not.	`true != false` outputs `true` `false != false` outputs `false`
And	`and`	25	Conjunction boolean operator.	`true and false` outputs `false` `true and true` outputs `true`
Or	`or`	25	Disjunction boolean operator.	`true or false` outputs `true` `false or false` outputs `false`
			Uses the first argument as parameter for the	

Compose	`|`	30	second argument. Note that, composition forces the second argument to be a function.	`-5 | abs` outputs `5`

Ternary Operator

Jtwig only contains one ternary operator. It allows to fork the behaviour based on a boolean expression, as exemplified below.

```
expr ? 1 : 2
```

Such expression will output `1` if the variable `expr` is true, or `2` if the variable is false.

Test

Test expressions are a complex predicate construct, they return a boolean value as result. Jtwig comes with some built in tests.

Name	Description	Example
Null	Checks whether a value is null or not.	`1 is null` outputs `false`
Divisible	Checks if a value is divisible by another.	`2 is divisible by 1` outputs `true`
Same As	Checks whether two objects are, actually, the same. Note that this uses the Java `==` operator between the two given operands.	`1 is same as 2` outputs `false`
Function based	This construct is based on the available list of functions defined in Jtwig. It even allows one to use user defined functions in a test expression.	`4 is defined` outputs `true` . Note that `defined` is a function from the built in list of functions.
Is Not	All test constructs listed before can be negated with the `is not` constructor.	`4 is not defined` outputs `false` </br> `1 is not null` outputs `true`

Selection Operator

The selection operator uses multiple strategies to extract properties or execute a method from a given identifier. An identifier in Jtwig can either specify a native Java object or a macro import. In this section one will only detail how the extracting of Java native object properties works. Lets start with two simple examples:

```
var.p1
var.method1(2)
```

The first expression above specifies a selection operation to extract property `p1` from `var` object. The second expression however is providing an argument, such feature allows Jtwig to execute Java methods. By default, there are several different strategies used to extract values from a Java object. All strategies are applied until one of them gets a value. It tries each strategy in the following order:

Method with the same name

In this strategy the given object meta information is searched, using reflection, for methods with the exact same name (case sensitive comparison) as the property name provided. Using the previous examples, it would search for a method with name `p1` without arguments given the first expression, where for the second expression it will search for a method, again, with the same exact name and the same number of arguments.

Method prefixed with `get`, `is` or `has`

Similar to the previous strategy but instead of searching for an exact match, it looks for methods with the `get`, `is` or `has` prefixes. As per previous first example, it would try to find in the following order `getP1`, `isP1` or `hasP1`. As exemplified, the first letter of the given property name is capitalized. Once again such comparisons are case sensitive. This strategy also works with arguments, exactly the same the previous strategy works.

Field with the same name

This strategy searches for fields with the exact same name as provided, the name comparison is case sensitive.

Map key with the same name

Is a strategy that only works against map objects and basically represents another way of accessing values in a map using the key as the property name.

Jtwig Tags

Jtwig tags provide developers a way to define rules over content. In this chapter one will describe the core tags available. Note that, this is also configurable, tags can be added as part of the Jtwig configuration, such will be subject to further explanation in later chapters.

Command constructs

Command constructs allows one to run specific commands against Jtwig. An important aspect common to all commands is that they never produce content, they only produce side effects.

`set` command

The set command allows to specify an assignment operation inside a Jtwig template, it will assign to the result of an expression to the specified variable.

```
{% set var = 2 + 3 %}
```

In the previous example one assigned the result of evaluating the expression `2 + 3` to the variable `var`. Note again that the output of the previous example will be empty because, as mentioned before, `set` as a command, does not produce content, it just affects the context defining, or redefining a variable.

`do` command

The `do` command just evaluates a given expression.

```
{% do something.run() %}
```

There is not a lot to say about this construct, it might be usefull to trigger behaviour from the template by running a Java method for example.

`flush` command

A common operation over streams is the `flush` operation. As a template engine Jtwig is very much associated with output buffers, which can be explicitly flushed. This forces the buffer to be flushed, for example, in a web application it will force the data to be sent over the wire.

```
{% flush %}
```

Control Flow constructs

Jtwig implements basic control flow constructs such as `if` conditions and `for` loops with the exact same syntax as Twig.

`if` conditions

If conditions are the simpliest control flow in Jtwig. It supports consecutive `elseif` conditions and the common `else` construct too.

```
{% if (expression) %}
  ... content if expression is evaluated to true ...
{% elseif (anotherExpression) %}
  ... content if anotherExpression is evaluated to true ...
{% else %}
  ... content if none of the previous conditions are met ...
{% endif %}
```

Note that if constructs will only output the content of the first block which condition is evaluated to true. The way Jtwig evaluates an expression to true is configurable and will be explained further on this manual.

In terms of variable scoping, if inner content shares the context with the parent context, this means if conditions can affect variables in the outer scope.

```
{% set variable = "a" %}
{% if (true) %}
  {% set variable = "b" %}
{% endif %}
{{ variable }}
```

The previous example will output `b` which illustrates how scoping works on if conditions by sharing the context with the parent construct.

`for` loops

Again based on Twig, Jtwig also implements, in a similar fashion `for` loops. It allows one to iterate over a list or a map. In this specific a list must be seens as a map where the key is the index.

```
{# Example with list #}
{% for item in list %}
  ... content using variable item ...
{% endfor %}

{# Example with map #}
{% for key, value in map %}
  ... content using variables key and value ...
{% endfor %}
```

Similar to the boolean expression evaluation, list or map evaluation in Jtwig is also configurable and will be discussed further on in this manual.

`loop` variable

For loops come with an extra variable defined in the context, the `loop` variable. The loop variable provides a set of useful properties when within a for loop, check below the properties exposed by this variable.

Property	Description
`length`	Size of the list or map being iterated
`index`	Current iteration count starting in 1
`index0`	Current iteration count starting in 0
`revindex`	Remaining number of iterations to reach the end of the list or map, ends in 1
`revindex0`	Remaining number of iterations to reach the end of the list or map, ends in 0
`first`	Boolean property only true in the first iteration
`last`	Boolean property only true in the last iteration
`parent`	Accessing the parent context

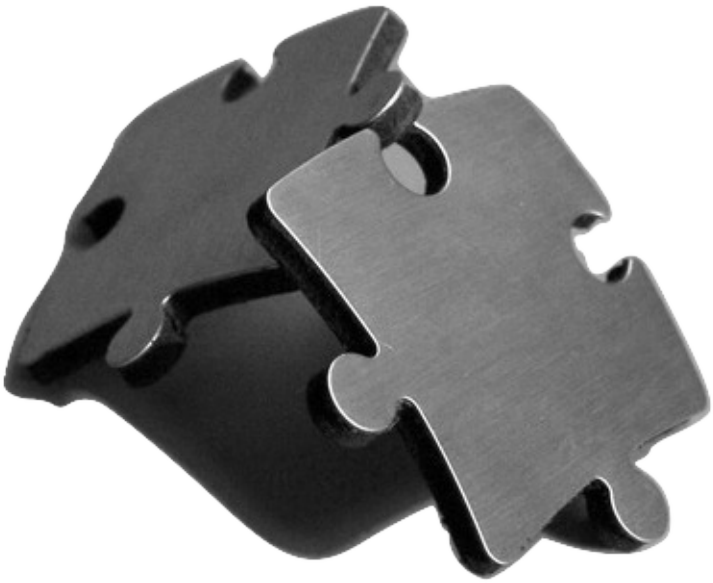
```
{% for item in [1, 2, 3] %}
  {% if (loop.first) %}
     Start
  {% endif %}
{% endfor %}
```

The previous example will output `Start` only once. This `loop` variable is only bound to the for context, it means this variable will not be visible outside of the for loop scope. If there is a `loop` variable in the context, it will not be overriden but to access it one need to get the parent context, for example:

```
{% set loop = 1 %}
{% for item in [1, 2, 3] %}
  {% if (loop.first) %}
     {{ loop.parent.loop }}
     {% set loop = 2 %}
  {% endif %}
{% endfor %}
{{ loop }}
```

The previous template will print `1 2`. Note that setting `loop` inside the for loop will write to the context, but won't override the for loop bound variable.

Modularization constructs



Jtwig comes with a powerfull modularization engine, such mechanism is key to improve template reusability and maintability. In this charpter one will talk about the available constructs in Jtwig and how to use them.

`include` construct

The simpliest modular construct in Jtwig is the `include` construct. By using this construct one can include another template.

```
{% include 'template.twig' ignore missing with {} only %}
```

The previous example shows all `include` possible arguments. In this section one will describe all arguments in detail.

Path (mandatory)

In it's simple form `include` only requires one argument, the template path. This is an expression which will be evaluated and used as the path for the Jtwig template to include.

```
{% include templatePath %}
```

The way Jtwig resolves the given path to a specific resource is configurable and will be detailed later.

`ignore missing` (optional)

If Jtwig resource resolution is not able to find the given resource, `include` will, by default, throw an error. When the argument `ignore missing` is set, Jtwig will ignore missing resources and proceed with the rendering.

```
{% include templatePath ignore missing %}
```

`with <expression>` (optional)

The `with` argument adds the capability to provide model variables to the included template. The expression is expected to be evaluated as a map.

```
{% include templatePath with { key: 'value' } %}
```

`only` (optional)

The `only` argument tells the engine to isolate the included template from the parent context. That way variables currently defined in the context won't be visible in the included template.

`extends` construct

The extends construct allows one to extend a given template redefining some of it's blocks. The concept of extending is very much associated with the concept of block detailed below. The extends construct expects one argument, a path for another Jtwig template.

```
{% extends pathExpression %}
```

Path

The path can be any expression that, once evaluated will then be used by the resource resolution mechanism in order to retrieve another Jtwig template. This resolution mechanism will be detailed futher on.

The extends template can then be followed by a sequence of `set`, `block` or `import` constructs only. For example:

```
{% extends pathExpression %}
{% import pathExpression as importAlias %}
{% set var = 1 %}
{% block ... %}{% endblock %}
```

`block` construct

A block construct plays two different roles depending on where it is placed. It can work as a placeholder with default content in parent templates or it can work as a content overrider in child templates.

Parent Template is an extendable template, meaning that other templates can extend it. This templates do not contain the `extends` construct (as shown previously).

Child Templates extend parent templates redefining its blocks. This templates contain the `extends` construct.

A block has an associated identifier, used as the way to reference it within child templates.

```
{% block blockId %}
  ... content here ...
{% endblock %}
```

From the previous example, the given block is identified with `blockId`, this mandatory identifier is also the only argument available for the block construct.

`block` as content placeholder

The `block` construct, as mentioned before, can be used as a content placeholder to define content that could be then replaced. A `block` is a content placeholder when used inside a parent template.

`block` as content overrider

The `block` construct, can also be used as a content overrider. This happens when the `block` is used underneath a child template.

How does it work

A block provides a way to change how a certain part of a template is rendered but it does not interfere in any way with the logic around it. Let’s take the following example to illustrate how a block works and more importantly, how it does not work:

```
{# block-base.twig #}

{% for post in posts %}
    {% block post %}
        <h1>{{ post.title }}</h1>
        <p>{{ post.body }}</p>
    {% endblock %}
{% endfor %}
```

If you render this template, the result would be exactly the same with or without the block tag. The block inside the for loop is just a way to make it overridable by a child template:

```
{# block-child.twig #}

{% extends "block-base.twig" %}

{% block post %}
    <article>
        <header>{{ post.title }}</header>
        <section>{{ post.text }}</section>
    </article>
{% endblock %}
```

Now, when rendering the child template, the loop is going to use the block defined in the child template instead of the one defined in the base one; the executed template is then equivalent to the following one:

```
{% for post in posts %}
    <article>
        <header>{{ post.title }}</header>
        <section>{{ post.text }}</section>
    </article>
{% endfor %}
```

`embed` construct

The embed construct works like a multiple inheritance mechanism, allowing developers to include and extend parent templates in one go.

```
{% embed resourceExpression ignore missing with mapExpression only %}
  {% block blockId %}
    ... redefine template ...
  {% endblock %}
{% endembed %}
```

Functionality wise, this construct is the result of merging the `include` feature with the `extends` one.

> Inside an `embed` tag one can only specify `block` tags which will override the parent template included.

Arguments

The `embed` arguments are exactly the same as the `include` construct and have identical behaviour.

Macro constructs

Macro constructs allows the user to reduce template code duplication by associating content to a specific name. In this chapter one will document how to define and use macros.

`macro` construct

The `macro` tag gives users a way to specify pieces of reusable templates by also allowing it to received arguments.

```
{% macro macroName (firstArgument, secondArgument, ...) %}
  ... reusable template content ....
{% endmacro %}
```

Macro Name

The macro name is an identifier used to reference the macro. It must be unique inside a Jtwig template file. There is no validation on macro name duplication, reusing a macro name within the same template file will only override the previous definition.

Macro Arguments

Macro arguments is a list of identifiers representing input variables which can then be used inside the macro body definition. All arguments are optional, is up to the caller to provide such arguments or not.

`import` construct

The `import` tag is the way to reuse defined macros, it affects the model, adding a variable containing all macros defined in a specific Jtwig template resource.

```
{% import resourcePath as macros %}
```

As shown in the previous example, all macro definitions inside `resourcePath` will be amalgamated in the `macros` variable, so called the **alias**.

How to use?

To call a macro, one just need to use the alias together with the desired macro name and its arguments.

```
{{ macros.macroName(argument1, argument2) }}
```

Let's then have a look to the example below.

File: `forms.twig`

```
{% macro text (name, defaultValue) %}
<input name="{{ name }}" type="text" value="{{ defaultValue }}" />
{% endmacro %}
```

File: `template.twig`

```
{% import 'forms.twig' as forms %}
{{ macros.text('username') }}
```

As one can see from the previous example, the template `template.twig` imports all macro definitions from `forms.twig` into the alias `forms`. It then renders the `text` macro providing only one argument `'username'`.

Argument resolution

As mentioned before, macro arguments are all optional. In the previous example, the macro defined two arguments `name` and `defaultValue`, however the call only provided one argument. The way arguments are feeded into the macro is by the order specified. In the previous example, the provided argument `'username'` will then be represented by the identifier `name` in macro specification.

Other Tags

`autoescape`

The `autoescape` construct only change the escape mode which will be used by the processing pipeline to generate the output. Escape modes are configurable in Jtwig, check Jtwig Core > Environment documentation for more information.

```
{% autoescape 'html' %}
<a href="#">Link</a>
{% endautoescape %}
```

The escaping functionality is attached to the very end of the Jtwig rendering pipeline.

`filter`

The filter construct is the way apply functions the a given content. It uses the provided body as the first argument in the specified function.

```
{% filter lower | capitalize %}
HELLO WORLD
{% endfilter %}
```

The previous example will produce `Hello world`. As we can see, the filter specified results in the composition of two distinct functions, it will first apply the `lower` function to the content `HELLO WORLD`, producing `hello world` and then apply the `capitalize` function producing `Hello world`.

`verbatim`

The `verbatim` tag is usefull avoiding the specifics around Jtwig syntax, as it will not try to parse the content.

```
{% verbatim %}
{{ test }}
{% endverbatim %}
```

The output of the previous template will be `{{ test }}`.

Built-in Functions

`abs`

Mathematical function allowing to get the absolute value of an expression. For example, `{{ abs(-1) }}` would produce `1`. It is expecting only one argument which must be converted to a number. Such conversion uses the `NumberConverter` configured. For more details check the Environment documentation.

`batch`

The batch function splits a given list in equally sized groups of items. It expects two or three arguments. A list as first argument, note that a list in Jtwig is a configurable concept as mentioned previously, whereby the ``CollectionConverter`` defined in the configuration will be used, it depends on the converter defined in the environment. The second argument is the group size. There is also an optional third argument used as padding, that is, if the last group is incomplete, the third argument will be used to fill it.

```
{{ batch([1,2,3], 2) }}
```

The previous example will output `[[1, 2], [3]]`.

```
{{ batch([1,2,3], 2, 0) }}
```

The previous example, now with the padding argument, will output `[[1, 2], [3, 0]]`.

`block`

Block function can be used to output the content of a defined block tag (check `{% block %}` tag definition).

```
{% block one %}Hello{% block %}{{ block('one') }}
```

The previous example will print `HelloHello`.

`capitalize`

The capitalize function is expecting one argument. It converts that argument to a String, using the `StringConverter` (check `Environment` documentation for more information).

```
{{ capitalize('hello world') }}
```

The previous template will render as `Hello world` which is the result of capilatizing the first word.

`concat` or `concatenate`

This function allows one to concatenate a set of strings. It is expecting an arbitrary number of arguments (varargs).

```
{{ concat('1', '+', '1', '=', '2') }}
```

Under the wood it uses the defined ``StringConverter`` to convert individual objects to a string representation and then concatenating them. The previous example will output ``1+1=2``.

`constant`

The constant function comes with two possible outputs depending on the number of arguments supplied. If one argument is supplied, it will return the value of the constant. If two arguments are provided, then it will compare the constant value against the second argument.

```
{{ constant("org.jtwig.example.TestClass.CONSTANT_NAME") }}
```

As shown in the above example, the expression will print the result of evaluating (using reflection) the constant with name `CONSTANT_NAME` defined in class `org.jtwig.example.TestClass`. Note that the evaluation will only work if the constant is public.

```
{{ constant("org.jtwig.example.TestClass.CONSTANT_NAME", "value") }}
```

The above example will return a Boolean expression. It will be `true` if the constant value is equal to `"value"`.

`default`

The default function is expecting two arguments. It returns the second argument if the first argument is either `null` or `Undefined`.

```
{{ default(null, 'Hello') }} {{ default(undefinedVariable, 'World') }}
```

if variable `undefinedVariable` is not defined in the provided `JtwigModel` then the previous example will produce `Hello World`.

defined

The defined function is useful to check if a given expression is defined or not. This logic is tied to the definition of an `Undefined` expression, explained later on (Jtwig core engine). It returns `true` if the given expression is defined, `false` otherwise.

```
{% if (defined([1, 2][5])) %}KO{% else %}OK{% endif %}
```

The previous example will output `OK` because index `5` of list `[1, 2]` is undefined.

empty

The empty function returns a boolean value. As input it is expecting a generic object, it returns true if the given input falls in one of the following scenarios:

- `null`
- `Undefined`
- non-empty list (implementing `Iterable` interface) of items
- non-empty array
- non-empty map
- non-zero number

Note that under the wood this function is using `NumberConverter` and `CollectionConveter`. Check the example below.

```
{% if (empty([1, 2])) %}A{% else %}B{% endif %}
```

The previous example produces `B`.

escape

The `escape` function allows to set the escape mode of the current context. A escape mode can be provided in order to choose which escaping strategy to use. If `false` is provided, it will set the escape mode to none.

```
escape(<Value> [, <Escape Mode>])
```

By default it uses HTML escape mode, meaning HTML special characters will be then escaped. For more information about available escaping strategies check `Environment` chapter.

```
{% autoescape 'html' %}
& {{ '&' | escape(false) }}
{%.endautoescape %}
```

The previous example, based on the Jtwig implementation of its processing pipeline and the escape mode functionality will produce `&&`.

even

This function is quite simple, it is expecting one number as argument and it returns true when that number is even, false otherwise.

```
{% if (even(2)) %}A{% else %}B{% endif %}
```

The previous example prints `A`. Note that the even function uses the `NumberConverter` concept to try to conver the given input to a number. If this conversion fails an exception will be thrown.

first

The first function returns the first element of a collection or String. If the argument provided is not a collection or a String, it will just return the input argument.

```
{{ first([1, 2]) }}
```

If the given argument is an empty list or String, then it returns `Undefined`. Note that this function uses the `CollectionConverter` mechanism. The previous template will output `1`.

format

The format function is just a Jtwig wrapper to call the `String.format` method available in Java. It receives an arbitrary number of arguments where the first is the template parameter (converted to a String) and the remaining arguments is provided as the model values for the `String.format` method.

```
{{ format('hello %s', 'world') }}
```

The previous example will print `hello world`. Note that this function uses the `StringConverter` (check `Environment` documentation for more information).

iterable

The iterable function allows one to check if a given argument is a collection for loops can iterate over, or index/map selections can be used.

```
{% if (iterable(2)) %}A{% else %}B{% endif %}
```

Iterable uses under the wood the `CollectionConverter`, which whenever an object can be converted to a Jtwig collection it will be iterable. The previous template will render `B` because `2` is not iterable as per default configuration.

join

The `join` function provides functionality somehow similar to the `concat` function, however they have some differences. To start with, `join` takes only one or two arguments, where the first argument is expected to be a list and the second, optional, argument a string to be used as the separator.

```
{{ join([1, null, 2], ', ') }}
```

The previous example will print `1, 2`. Note that, `join` function ignores `null` values.

keys

Keys function can be used to expose the collection of keys for a given collection. It uses the configured `CollectionConverter` (for more details check the Environment documentation).

```
{{ keys(['A', 'B']) }}
```

The previous example, as per the default configuration, will print `[1, 2]`.

last

The last function returns the last element of a collection or String. If the argument provided is not a collection or a String, it will just return the input argument.

```
{{ last([1, 2]) }}
```

If the given argument is an empty list or String, then it returns `Undefined`. Note that this function uses the `CollectionConverter` mechanism (check `Environment` documentation for more information). The previous template will output `2`.

length

The length function returns the length of a given collection or String. If neither a collection or String is provided then it returns `0` for both `null` and `Undefined`, otherwise `1` will be the result.

```
{{ length([1, 2]) }}
```

```
{{ length(null) }}
```

```
{{ length(9) }}
```

The previous examples will print respectively `2`, `0` and `1`. Note that this function uses the defined `CollectionConverter` configured, for more information visit the `Environment` documentation.

lower

Lowers the case of the String provided. Note that this function uses the `StringConverter` (check `Environment` documentation for more information). The following example will produce `jtwig`.

```
{{ lower('JTWIG') }}
```

merge

The `merge` function allows one to merge an arbitrary number of lists together by the given order. This function requires at least two arguments to run. It also supports singular elements as arguments.

```
{{ merge(1, 2, 3) }}
```

```
{{ merge([1, 2], 3) }}
```

The previous two examples return the same output, which is, `[1, 2, 3]`. Note that, this function uses the `CollectionConverter` defined by the Jtwig configuration.

nl2br

This function allows one to convert new line characters into it's HTML sibling `<br />`, just as simple as that. It uses `StringConverter` (check `Environment` documentation for more information) to convert the given argument to a String.

number_format

The `number_format` allows one to format a given number with specific symbols for the grouping and decimal separators, also the number of fractional digits. This function expects at least one argument and can receive up to four arguments. The list of arguments is presented below by the order they are expected by Jtwig.

1. The number to be formatted
2. The number of fractional digits (optional)
3. The decimal separator (optional)
4. The grouping separator (optional)

```
{{ number_format(11000.136, 2, '.', ' ') }}
```

The previous example will produce `11 000.14`. The rounding applied in here is the same strategy as `BigDecimal.ROUND_HALF_DOWN` (check Java documentation for more information). This function uses `StringConverter` to convert the third and fourth arguments to Strings, it also uses the `NumberConverter` to convert the first and second arguments to numbers, check `Environment` documentation for more information about this converters.

odd

This is the oposite of the `even` function, it is expecting one number as argument and it returns true when that number is odd, false otherwise.

```
{% if (odd(2)) %}A{% else %}B{% endif %}
```

The previous example prints `B`. Note that the even function uses the `NumberConverter` concept to try to conver the given input to a number. If this conversion fails an exception will be thrown.

raw

This function clears the current escape mode of the context. It is equivalent to `escape(false)`. This function does not take arguments.

```
{% autoescape 'HTML' %}
{{ '&' | raw }}
{% endautoescape %}
```

The previous example will produce `&` as output.

replace

The replace function allows one to specify a String and a map of replacements replacing all the ocurrences of the keys in the provided map by their respective value converted to a String. It expects two arguments, where it uses `StringConverter` to convert the first argument to a String and `CollectionConverter` to convert the second argument to a list of key value pairs.

```
{{ replace('Hello %name%', { '%name%': 'world' }) }}
```

The previous example will produce `Hello world` as output.

reverse

Reverse function reverses the order of the elements in a given collection or String. If no collection neither String is provided then it returns the given argument.

```
{{ reverse([1, 2]) }}
```

The previous example will print `[2, 1]`. Note that this function uses the defined `CollectionConverter` configured, for more information visit the `Environment` documentation.

round

The round function allows one to round a given number to integer. An optional strategy can be specified as second argument:

- `'CEIL'`
- `'FLOOR'`

```
{{ round(1.3, 'CEIL') }}
```

The previous example will produce `2`. Note that the strategy selection is case insensitive, so one can either specify `'CEIL'` or `'ceil'`. If the second argument is not specified `BigDecimal.ROUND_HALF_DOWN` will be applied. For further information check the official Java documentation.

slice

This function is expecting three arguments, where the first argument can either be a String or a collection (it uses `CollectionConverter` under the wood), and an integer as second and third arguments.

```
{{ slice("123", 1, 1) }}
```

```
{{ slice([1, 2, 3], 0, 2) }}
```

The second argument is the index position the slice will start from (inclusive), where the third argument is the length of the slice. As shown in the previous two examples, the result will be `"2"` and `[1, 2]` respectively. Note that, slice is smart enough to handle boudary cases, for exaple:

```
{{ slice("123", 2, 3) }}
```

```
{{ slice("123", 5, 1) }}
```

The previous examples will still return a slice, depending on the number of characters or items provided in the first argument, the previous examples would then resolve the slice to `"3"` and `""`

sort

Sort can be used to sort elements of a given collection in ascending order. It is based on the underlying core Java `java.lang.Comparable` interface, which elements should implement.

```
{{ sort([1, 3, 2]) }}
```

The previous example will produce `[1, 2, 3]`.

split

This function expects two arguments, using the second argument provided to split the first one into a collection. Such arguments are converted to String using the `StringConverter` configured in the `Environment` as detailed previously.

```
{{ split('jtwig-2', '-') }}
```

The previous example will return `[jtwig, 2]`.

striptags

The `striptags` function is expecting at least one argument or two at most. This function emulates the PHP `strip_tags` function behaviour in Java. The first argument is the String which HTML elements will be stripped. Where the second optional argument (a String aswell) enables users to specify a list of allowed tags (tags that won't be stripped).

```
{{ striptags('<a>jtwig</a><button>Submit</button>', '<a>') }}
```

The previous example will produce `<a>jtwig</a>Submit`. Note that this function uses `StringConverter` to convert the arguments to a String, check `Environment` documentation for further information.

title

The title function is expecting one String argument, capitalizing the first letter of all words present in it.

```
{{ title('hello world') }}
```

The previous example will produce `Hello World`. Note that this function uses `StringConverter` to convert the arguments to a String, check `Environment` documentation for further information.

trim

Similar to the Java sibling `String::trim`, the trim function in Jtwig removes any whitespace characters from the beggining and/or ending of the given argument.

```
{{ trim('  Hello World  ') }}
```

The previous example will produce `Hello World`. Note that this function uses `StringConverter` to convert the arguments to a String, check `Environment` documentation for further information.

upper

Changes the case of the String provided by turning all into capitals. Note that this function uses the `StringConverter` (check `Environment` documentation for more information). The following example will produce `JTWIG`.

```
{{ upper('jtwig') }}
```

url_encode

Encoding values to a url is the role of this function. It is expecting either a String or a map.

```
{{ url_encode({id: 1, special: '&'}) }}
```

The previous example produces `id=1&special=%26`.

```
{{ url_encode('one&two') }}
```

The previous example produces `one%26two`.

Jtwig Core

Within this section one will detail how to work with Jtwig at it's core API. As a template engine, Jtwig has three main concepts, they are **Environment**, **Resource** and **Model**.

```
Output = (Environment, ResourceReference, Model)
```

The **Environment** contains all Jtwig configurations and predefined behaviour, this includes possible extensions that might be added. The **ResourceReference** contains the intermediate Jtwig representation, also known as Template and the **Model** is the container of key and value pairs which combined with the Template generates the output. We can break it down in the following way.

```
Template = (Environment, ResourceReference)
```

Where, basically, the `Template` is the combination of the **Environment** with the **ResourceReference**. This has special meaning when extensions are added and means some intermediate representations don't mean anything without the proper extension added to the **Environment**. Note that, one will detail about Extensions later on.

```
Output = (Template, Model)
```

Hello World Example

Let's now have a look at the famous Hello World program in Jtwig using the core API.

```
// Environment
EnvironmentConfiguration configuration = new DefaultEnvironmentConfiguration();
EnvironmentFactory environmentFactory = new EnvironmentFactory();
Environment environment = environmentFactory.create(configuration);

// Resource
ResourceReference resource = new ResourceReference(
    ResourceReference.STRING,
    "Hello {{ token }}!"
);

// Template
JtwigTemplate jtwigTemplate = new JtwigTemplate(environment, resource);

// Model
JtwigModel model = JtwigModel.newModel().with("token", "World");

// Output
String output = jtwigTemplate.render(model);
```

As one can see, the way Jtwig core API is built follows the same concepts mentioned before, where the **Enviornment** and **ResourceReference** are first instantiated in order to create the `JtwigTemplate`, which when combined with the `JtwigModel` generates the output.

`JtwigTemplate` API

The `JtwigTemplate` implementation comes with API to simplify some application code. Namely the static methods `inlineTemplate`, `fileTemplate` and `classpathTemplate` allows one to, respectively, load a Jtwig template directly from a String, file or classpath. Using the previous example, one can simplify the code using the `inlineTemplate` method, as the below example shows.

```
// Environment
EnvironmentConfiguration configuration = new DefaultEnvironmentConfiguration();

// Template
JtwigTemplate jtwigTemplate = JtwigTemplate
    .inlineTemplate("Hello {{ token }}!", configuration);

// Model
JtwigModel model = JtwigModel.newModel().with("token", "World");

// Output
String output = jtwigTemplate.render(model);
```

`JtwigModel` API

Jtwig Model can be seen as a map of properties, which will then be used to render the template. The keys can only be valid Java identifiers as mentioned before.

```
JtwigModel model = JtwigModel.newModel()
     .with("variable", "Jtwig")
     .with("secondVariable", 1);
```

Processing Pipeline

Parsing Stage

The way Jtwig produces the final result is through a well defined processing pipeline. At start of this whole processing is the parsing stage. The parsing stage is where Jtwig parses a file producing a tree of nodes based of the Jtwig Abstract Syntax Tree. This tree is composed of content nodes and expressions. As mentioned before expressions are evaluated to values, and where content nodes are evaluated to streamable content. By content nodes we mean all the Jtwig tags and raw text defined in the template.

Lazy loading nested resources

In Jtwig nested resources, which can be referred using `include`, `extends`, `embed` and `import` constructs, are loaded in rendering time (lazy loaded) as the path expression needs to be evaluated first.

Caching resources

Another concept included in the parsing stage is the resource caching mechanism. By default Jtwig will cache the parsed resources in memory in a persistent fashion, that means, for the lifetime of the JVM, resources are only parsed once.

Rendering Stage

The next stage is the rendering stage. Each content node in the parsed tree is processed into streamable content. Before each content node being rendered the associated escape mode is initialised. This will then be used to serialize the output. The Jtwig serializer is capable of escaping content based on a specified strategy. Supported escape modes were already specified in the `autoescape` tag definition.

Expression Evaluation

During the rendering stage, expressions are evaluated, for example, to perform the Jtwig control flow. An important aspect of this evaluation mechanism is a special type introduced by Jtwig, that is, the `Undefined` value. It's a singleton used to specify when the result of evaluating an expression is undefined. For example, accessing an undefined array index or evaluating an undefined variable.

Environment

In this section one will define the `Environment` concept, how it can be configured, also describing the default configuration.

As mentioned before, the Environment contains all the configured properties and behaviour while rendering Jtwig Templates, it might include extensions as well. The simpliest way to instantiate an `Environment` object is with the following code.

```
EnvironmentConfiguration configuration = new DefaultEnvironmentConfiguration();
EnvironmentFactory environmentFactory = new EnvironmentFactory();
Environment environment = environmentFactory.create(configuration);
```

In order to create an `Environment` instance one need an `EnvironmentConfiguration` to be provided to the `EnvironmentFactory`.

`EnvironmentConfigurationBuilder` API

However, as shown in the previous example, the `DefaultEnvironmentConfiguration` creates an immutable `EnvironmentConfiguration` instance. To overcome that limitation and customize Jtwig configuration the `EnvironmentConfigurationBuilder` API was created. It comes with a set of methods to specify all possible Jtwig behaviour. As Jtwig is highly configurable, this API offers a tree of builders to organise and ease the specification of customised behaviour. Note that all the builders follow a common convention:

- All the builder methods starting with `with` will set the underlying configuration field.
- All methods starting with `without` will unset the underlying configuration field.
- `add` methods will append elements to the current list or map of values. For example, Jtwig allows the user to specify multiple extensions, in this case, `add` can be used to add another extension on top already existing ones.
- `set` methods will override the currently defined list or maps of values.
- `filter` methods allows the user to specify a filtering predicate which will be used to filter the existing list or map of items, such methods are useful to modify pre-defined behaviour.
- `and` methods enables developers to return the parent builder.

Extending the default configuration

An useful constructor of this builder is the prototype constructor which allows one to initialize the builder given an instance of an EnvironmentConfiguration. Specially useful when extending the default configuration (the most common scenario), instead of creating an entire new one.

```
new EnvironmentConfigurationBuilder(new DefaultEnvironmentConfiguration())
                .build()
```

The example above will create an instance of EnvironmentConfiguration copying all the definitions from the default configuration. One can then use the builder to modify the default configuration. Another way to achieve the same is by using the static method `configuration` of `EnvironmentConfigurationBuilder`, which makes the previous code snippet equivalent to the following one.

```
EnvironmentConfigurationBuilder.configuration().build()
```

`parser()`

The `parser()` method returns an instance of `AndJtwigParserConfigurationBuilder`, such class was built for the purpose of configuring the Jtwig parser. Let's see an example of it.

```
EnvironmentConfiguration configuration = EnvironmentConfigurationBuilder
                .configuration()
                    .parser()
                        .syntax()
                            .withStartCode("{ %").withEndCode("%}")
                            .withStartOutput("{{").withEndOutput("}}")
                            .withStartComment("{#").withEndComment("#}")
                        .and()
                        .addonParserProviders().add(customAddonParser()).and()
                        .binaryOperators().add(customBinaryOperator()).and()
                        .unaryOperators().add(customUnaryOperator()).and()
                        .withoutTemplateCache()
                    .and()
                .build();
```

With this one can specify:

- `StartCode`, `EndCode`, `StartOutput`, `EndOutput`, `StartComment` and `EndComment` allows one customize the syntactic symbols used by Jtwig code islands. By default, Jtwig sets `"{ %"`, `"%}"`, `" "`, `" "`, `"{#"` and `"#}"` respectively.
- `AddonParserProviders`, `BinaryOperators` and `UnaryOperators` provides API to enhance the parser with extra addons, this will be detailed further on. All the mentioned methods

are used to build lists of objects which means one can specify as many as we want. Jtwig by default doesn't include addons, however, in terms of `BinaryOperators` and `UnaryOperators` you can get the ones already specified as part of the Jtwig expression syntax definition.

- `TemplateCache` setting gives the user the possibility to configure a cache for compiled Jtwig templates. Such mechanism speeds up the parsing operation. It uses **Resource** as key and returns, as mentioned, the Jtwig compiled templates. By caching it, operations like reading files and flatening the template structure can get a significant performance boost. Jtwig core comes with one implementation used by default: `InMemoryConcurrentPersistentTemplateCache` which was built with high performance standards.

`functions()`

The `functions()` method returns a configurable list builder of Jtwig functions. Such functions will be provided to the Jtwig function repository mechanism, a fundamental piece of the function resolution system.

```
EnvironmentConfiguration configuration = EnvironmentConfigurationBuilder
                .configuration()
                    .functions()
                        .add(jtwigFunction)
                    .and()
                .build();
```

The list of functions available by default in Jtwig was already described in a previous chapter.

`resources()`

The `resources()` method returns an instance of `AndResourceResolverConfigurationBuilder` used to build the resource resolver.

```
EnvironmentConfiguration configuration = EnvironmentConfigurationBuilder
    .configuration()
        .resources()
            .resourceLoaders().add(typedResourceLoader).and()
            .absoluteResourceTypes().add("string").and()
            .relativeResourceResolvers().add(relativeResourceResolver).and()
            .withResourceReferenceExtractor(extractor)
            .withDefaultInputCharset(Charset.forName("UTF-8"))
        .and()
    .build();
```

The first configuration refered in the previous example is the `resourceLoaders` list builder, where multiple `TypedResourceLoader` can be specified. A `TypedResourceLoader` is a pair of reference type and the associated `ResourceLoader`. If multiple resource loaders are specified for the same type, Jtwig combines them together. The `absoluteResourceTypes` is a list builder where reference types can be marked as absolute. It's also possible to specify relative resources resolvers using `relativeResourceResolvers` list builder.

The resource reference extractor, is unlikely to be customized, but it extracts a given string representation of a resource reference into a pair of `(type, path)`. The default implementation is expecting references to be like `type:path` using `:` as separator. A default input charset can also be provided via `withDefaultInputCharset`, it will be used as the default input encoding for loaded templates.

`render()`

The `render()` method returns an instance of `AndRenderConfigurationBuilder` which provides a set of useful builder methods to configure the rendering, let's check the following example.

```
EnvironmentConfiguration result = EnvironmentConfigurationBuilder
                .configuration()
                    .render()
                        .withStrictMode(strictMode)
                        .withInitialEscapeMode(initialEscapeMode)
                        .withOutputCharset(outputCharset)
                        .nodeRenders().add(CustomNode.class, nodeRender).and()
                        .expressionCalculators()
                            .add(CustomExpression.class, expCalculator).and()
                        .binaryExpressionCalculators()
                            .add(CustomBinaryOperator.class, binOpCalc).and()
                        .unaryExpressionCalculators()
                            .add(CustomUnaryOperator.class, unaryOpCalc).and()
                        .testExpressionCalculators()
                            .add(CustomTestExpression.class, testCalc).and()
                    .and()
                .build();
```

Here you can find the following properties:

- `StrictMode` sets the way to resolve variables in Jtwig, if strict mode is active, undefined variables will throw an exception when evaluated. However, if strict mode is disabled, it will be evaluated to `Undefined.UNDEFINED`. Strict mode is disabled by default.
- `OutputChatset` defines the default output charset for Jtwig, this is used at the Jtwig rendering stage. By default it uses `Charset.defaultCharset()`, check Java documentation for more information.

- `NodeRenders` is a map of Content Node type to an implementation of the RenderNode interface. Such interface tell Jtwig how to render such type of element once they appear on the Jtwig rendering tree.
- `ExpressionCalculators` holds the mapping from Expression to it's calculator, allowing Jtwig to evaluate the given expression value.
- `BinaryExpressionCalculators` , `UnaryExpressionCalculators` and `TestExpressionCalculators` again, allows the user to specify implementations of calculators so that Jtwig can use to evaluate such expressions value.

The ``NodeRenders``, ``ExpressionCalculators``, ``BinaryExpressionCalculators``, ``UnaryExpressionCalculators`` and ``TestExpressionCalculators`` defined by default were already described in the Jtwig syntax definition.

`escape()`

The escape configuration allows one to configure Jtwig special characters escaping capability.

```
EnvironmentConfiguration configuration = EnvironmentConfigurationBuilder
                .configuration()
                    .escape()
                        .withInitialEngine("none")
                        .withDefaultEngine("custom")
                        .engines()
                            .add("custom", customEscapeEngine)
                        .and()
                    .and()
                .build();
```

- `Engines` is a map of String to escape engine, which will be used to resolve escape mode identifiers to escape engines. By default, the following are provided. `false` or `'none'` , which will perform no escaping at all. `'js'` or `'javascript'` , for Javascript special characters escaping strategy and, finally, `'html'` to escape HTML special characters.
- `InitialEngine` sets the initial escape mode, which by default is set to `'none'` , which means, strings wont be escaped when rendering the template.
- `DefaultEngine` property sets the default escape mode, which by default is set to `HTML` , which means, when not specified by `autoescape` tag or `escape` function `HTML` escaping engine will be used.

`propertyResolvers()`

This method allows one to setup multiple property resolvers. The default property resolution mechanism was already detailed when describing the selection operator behaviour.

```
EnvironmentConfiguration configuration = EnvironmentConfigurationBuilder
                .configuration()
                    .propertyResolver()
                        .add(propertyResolver)
                    .and()
                .build();
```

`enumerationStrategies()`

The `enumerationStrategies()` method allows to configure the list enumeration stretagies. This strategies are used when resolving lists by comprehension in Jtwig. By default, and as mentioned before, Jtwig comes with four different strategies:

- `CharDescendingOrderEnumerationListStrategy`
- `CharAscendingOrderEnumerationListStrategy`
- `IntegerAscendingOrderEnumerationListStrategy`
- `IntegerDescendingOrderEnumerationListStrategy`

```
EnvironmentConfiguration configuration = EnvironmentConfigurationBuilder
                .configuration()
                    .enumerationStrategies()
                        .add(enumerationListStrategy)
                    .and()
                .build();
```

`value()`

```
EnvironmentConfiguration configuration = EnvironmentConfigurationBuilder
                .configuration()
                    .value()
                        .withMathContext(mathContext)
                        .withRoundingMode(roundingMode)
                        .withValueComparator(valueComparator)
                        .withStringConverter(stringConverter)
                        .numberConverters().add(numberConverter).and()
                        .booleanConverters().add(booleanConverter).and()
                        .charConverters().add(charConverter).and()
                        .collectionConverters().add(collectionConverter).and()
                    .and()
                .build();
```

The `value()` method returns an instance of `AndValueConfigurationBuilder` allowing to configure Jtwig value handling.

- `MathContext` sets the java BigDecimal Math Context, note that Jtwig math operations are performed using Java's BigDecimal API, for that, whenever needed, the context specified in here will be used. By default Jtwig sets `MathContext.DECIMAL32`. For more information, check Java documentation.
- `RoundingMode` is used by mathematical operations like divide and multiply as rounding might be applied. By default Jtwig specifies `RoundingMode.HALF_UP`, check Java documentation of `RoundingMode` for more information.
- `ValueComparator` is used for all comparisons in Jtwig. By default, the value comparator tries to convert the operands to a number (then comparing using the BigDecimal equals method) or it converts the operands to a string (using the String equals method).
- `StringConverter` allows to specify the logic to use when converting objects to a String value, note this is used to serialize any object in Jtwig, the default implementation is null safe (returning empty string if null) and only returns the result of the Java native Object `toString` method.
- `NumberConverters` configuration field which is a list of converters from any possible object to Number. Jtwig environment will then chain this list of converters together in a composite converter. Such composite converter will call the specified converters where the first returning a value will be used. The default implementation can be defined by the following table:

Java Object	Result
null	0
Undefined	0
Boolean	1 if true, 0 if false
String	Will try to parse the string

- `CollectionConverters` provides a similar api as `NumberConverters`, users can appen as many collection converters as they want, Jtwig will chain them applying the same logic used for composing multiple `NumberConverters`. The default configuration for this will cover the following scenarios:

Java Object	Result
null	null
Array	keys as the index and values as the items in the array
Iterable	keys as the index and values as the items in the iterable
Map	all key and values in the map

- `CharConverters` is another set of converters allowing Jtwig to convert a given generic Java object to a `Character`. The default behaviour you can expect will be the following:

Java Object	Result
null	null
Object	Object.toString only char, if there is one

- `BooleanConverters` similar to the previous converters mentioned here, the following behaviour can be expected as default:

Java Object	Result
null	false
Undefined	false
String	`"true"` and `"false"` get converted to true and false
Array	false if empty, true otherwise
Iterable	false if empty, true otherwise
Map	false if empty, true otherwise
Number	false if 0, true otherwise

`extensions()`

In order to add extensions to the Jtwig core behaviour one can use the `withExtension` method. Note that by default Jtwig Core does have any extension, however as part of the Jtwig community there is a list of already created extensions.

```
EnvironmentConfiguration configuration = EnvironmentConfigurationBuilder
                .configuration()
                    .extensions()
                        .add(customExtension1)
                        .add(customExtension2)
                    .and()
                .build();
```

`parameters()`

In a more advanced context, Jtwig allows developers to add custom configuration parameters. This can be used by extensions further on, during rendering time from the ``Environment::parameter(String)`` method.

```
EnvironmentConfiguration configuration = EnvironmentConfigurationBuilder
                .configuration()
                    .parameters()
                        .add(parameter1, value1)
                        .add(parameter2, value2)
                    .and()
                .build();
```

Resources

Resources in Jtwig are modeled as references. It uses the `ResourceReference` class to do so. A resource reference is a pair of type and path. For example:

- `file:/tmp/file.twig` represents a reference with type `file` and path `/tmp/file.twig`
- `classpath:template.twig` refers to type `classpath` and path `template.twig`.

Reference Type

To represent the type one use a raw string, allowing for custom types to be introduced at configuration time. Each type must have an associated `ResourceLoader`, allowing one to, given a reference, perform different operations, like:

- **Load** reading the reference as an `InputStream`.
- **Exists** check if the given reference exists.
- **URL** return the URL representation of the reference, if it exists.
- **Charset** return the charset of such reference, if possible.

Types defined in Core

The Jtwig Core default configuration comes with three reference types, namely:

- `file`
- `classpath`
- `string`

Absolute Type vs Relative Type

In Jtwig a reference type can either be relative or absolute, meaning that, relative path calculation is possible or not. For example, `file` and `classpath` types are relative types, while `string` is absolute.

Relative Resource Resolver

For relative reference types only the concept of relative resource resolver exists so that, using a given reference as base path, calculate the path to another reference.

The `string` reference

The `string` reference type is a special kind of reference where as path, one can provide, actually, a template definition. For example, the resource reference `string:{{ "hello world!" }}` defines a resource with content `{{ "hello world!" }}` which when rendered will produce `hello world!`.

The `any` reference type

The resource resolution in Jtwig also allows one to refer to a given resource without specifying the type, as such, the defined type will be `any`, meaning a best effort approach will be used to load/resolve the given resource. For example to load the reference `/tmp/template.twig` (without the type) Jtwig will iterate over the list of `ResourceLoader` and the first one finding the resource will be used. For this is important the ordering defined during configuration. As default the ordering is as such:

- `file` with base directory as current working directory
- `classpath` using jtwig-core `ClassLoader`

Note that, the `string` type is ignored for the purpose of resource lookup.

Integration

Integrating Jtwig Core on your project is quite simple. It will mainly depend on the dependency management mechanism you use. You will need to make sure `jcenter` is part of your repository list. Such will allow you to get access to the `jtwig-core` dependency. To check the most recent version, go to bintray.

For some integration examples check jtwig-examples.

Gradle

```
repositories {
    jcenter()
}
dependencies {
    compile 'org.jtwig:jtwig-core:5.X'
}
```

Maven

```
<repositories>
    <repository>
        <id>bintray</id>
        <url>https://jcenter.bintray.com/</url>
    </repository>
</repositories>

<dependencies>
    <dependency>
        <groupId>org.jtwig</groupId>
        <artifactId>jtwig-core</artifactId>
        <version>5.X</version>
    </dependency>
</dependencies>
```

SBT

```
libraryDependencies += "org.jtwig" % "jtwig-core" % "5.X"
```

Jtwig Web

Jtwig Web project extends Jtwig Core with and eases the integration with Java Servlet API (version 3.1). It also adds the function `path`, a new resource `WebResource` and resource resolver `WebResourceResolver`.

The `path` function

The path function can be used to output the Servlet Context Path, for more information about this value, check the Servlet API documentation. It is expecting either none of one argument. If no argument is provided, as mentioned, it will only print the servlet context path, on the other hand, if the an argument is provided (expected to be a String), then it appends such value to the servlet context path.

```
{{ path('/index') }}
```

Assuming the servlet context path is `/application`, then the previous example will output `/application/index`.

The `WebResource` and `WebResourceResolver`

This combo of `WebResource` and `WebResourceResolver` allows Jtwig to support web (`WEB-INF/` rooted) resources. Users can then reference such resources from their app. Note that, if a given path is prefixed with `web:`, it will be interpreted by this resource resolver, such can be used to disambiguate the source of those resources.

Combining with the Servlet API

To integrate Jtwig Web with the Servlet API the `JtwigRenderer` concept was introduced.

```
public static class HelloWorldServlet extends HttpServlet {
    private final JtwigRenderer renderer = JtwigRenderer.defaultRenderer();

    @Override
    protected void service(
        HttpServletRequest request,
        HttpServletResponse response
        ) throws ServletException, IOException {
        request.setAttribute("variable", "Hello");

        renderer.dispatcherFor("/WEB-INF/templates/index.twig.html")
                .with("name", "Jtwig")
                .render(request, response);
    }
}
```

The example above can be seen in jtwig-examples. It will print `Hello World!`, as the `/WEB-INF/templates/index.twig.html` template is defined as `{{ variable }} {{ name }}!`. As one can see in the previous example, `JtwigRenderer` exposes all request variable as part of the `JtwigModel` passed to the render stage.

Note that, this `JtwigRenderer` also exposes an API call to specify the Jtwig template inline, as shown in the example below. Such example will produce the same output as the example above.

```
public class HelloWorldServlet extends HttpServlet {
    private final JtwigRenderer renderer = JtwigRenderer.defaultRenderer();

    @Override
    protected void service(
            HttpServletRequest request,
            HttpServletResponse response
    ) throws ServletException, IOException {
        request.setAttribute("variable", "Hello");

        renderer.inlineDispatcherFor("{{ variable }} {{ name }}!")
                .with("name", "Jtwig")
                .render(request, response);
    }
}
```

The `app` variable

`JtwigRenderer` injects into the `JtwigModel` the `app` variable which exposes information extracted from the `HttpServletRequest` provided.

```
public class Application {
    private HttpRequest request;
}
```

```
public class HttpRequest {
    private Map<String, Object> parameter;
    private Map<String, Object> query;
    private Map<String, Object> session;
    private Map<String, Object> cookies;
}
```

This allows one to access:

- GET query parameters (using `app.request.query.parameterName`)
- POST parameters (using `app.request.parameter.parameterName`)
- Session parameters (using `app.request.session.parameterName`)
- Cookie parameters (using `app.request.cookies.parameterName`)

Integration

Integration of Jtwig Web in your project will depend on the dependency management mechanism being used. Also, you will need to make sure `jcenter` is part of your repository list. To check the most recent version, go to bintray.

Gradle

```
repositories {
    jcenter()
}
dependencies {
    compile 'org.jtwig:jtwig-web:1.X'
}
```

Maven

```
<repositories>
    <repository>
        <id>bintray</id>
        <url>https://jcenter.bintray.com/</url>
    </repository>
</repositories>

<dependencies>
    <dependency>
        <groupId>org.jtwig</groupId>
        <artifactId>jtwig-web</artifactId>
        <version>1.X</version>
    </dependency>
</dependencies>
```

Jtwig Spring

Project `jtwig-spring`

The `jtwig-spring` project includes implementations of Spring `View` and `ViewResolver`, allowing one to integrate Jtwig with Spring MVC.

```
@Configuration
@EnableWebMvc
public class WebConfig {
    @Bean
    public ViewResolver viewResolver () {
        JtwigViewResolver viewResolver = new JtwigViewResolver();
        viewResolver.setPrefix("web:/WEB-INF/templates/");
        viewResolver.setSuffix(".twig.html");
        return viewResolver;
    }
}
```

The example above defines the `ViewResolver` bean used by Spring MVC to render a given view. The example shown here can be found in jtwig-examples.

Integrating

Integrating Jtwig Spring on your project is quite simple with the help of dependency managers. To check the most recent version, go to bintray.

Gradle

```
repositories {
    jcenter()
}
dependencies {
    compile 'org.jtwig:jtwig-spring:5.X'
}
```

Maven

```
<repositories>
    <repository>
        <id>bintray</id>
        <url>https://jcenter.bintray.com/</url>
    </repository>
</repositories>

<dependencies>
    <dependency>
        <groupId>org.jtwig</groupId>
        <artifactId>jtwig-spring</artifactId>
        <version>5.X</version>
    </dependency>
</dependencies>
```

Jtwig Spring Boot

The `jtwig-spring-boot-starter` project allows one to easily integrate Jtwig with Spring Boot. Just by adding the dependency to your project, spring-boot will then load `JtwigViewResolver`.

Integrate

Check the most recent version, go to bintray.

Gradle

```
repositories {
    jcenter()
}
dependencies {
    compile 'org.jtwig:jtwig-spring-boot-starter:5.X'
}
```

Maven

```
<repositories>
    <repository>
        <id>bintray</id>
        <url>https://jcenter.bintray.com/</url>
    </repository>
</repositories>

<dependencies>
    <dependency>
        <groupId>org.jtwig</groupId>
        <artifactId>jtwig-spring-boot-starter</artifactId>
        <version>5.X</version>
    </dependency>
</dependencies>
```

Default Configuration

If you include jtwig-spring-boot-starter in your project, by default it will set the view resolver with prefix `classpath:/templates/`, suffix will be set as `.twig` and the Jtwig default configuration will be used. Note that, jtwig-spring-boot-starter uses jtwig-web which extends the default Jtwig Core configuration, as already mentioned.

Costumize Configuration

It is still possible to customize `JtwigViewResolver` to define prefix, suffix and also Jtwig Environment. For that `JtwigViewResolverConfigurer` interface can be extended by `@Configuration` annotated class. Note that, such class needs to be injected by spring-boot to the application context.

```
@Configuration
public class JtwigConfig implements JtwigViewResolverConfigurer {
    @Override
    public void configure(JtwigViewResolver viewResolver) {
        viewResolver.setRenderer(new JtwigRenderer(EnvironmentConfigurationBuilder
                .configuration()
                .extensions().add(new MyExtension()).and()
                .build()));
    }

    private static class MyExtension implements Extension {
        @Override
        public void configure(EnvironmentConfigurationBuilder configurationBuilder) {
            System.out.println("Hi");
        }
    }
}
```

The previous example sets the `JtwigViewResolver` renderer with an extended version of the `EnvironmentConfiguration` including a dummy `Extension`. This working example can be found in jtwig-examples.

Extensions

In this chapter one will mention the currently available extensions for Jtwig. This includes:

- Translate Extension
- Render Extension
- Json Extension
- Spaceless Extension

Translate Extension

The translate extension adds internationalization capabilities to Jtwig. It heavily relies on the concept of `MessageSource`, which will be detailed futher on. This extension is highly configurable and can be optimized for each specific use case.

Translation

The translation engine in Jtwig relies in two distinct concepts:

- Message Source
- Message Decoration

Message Source

Message source mechanism acts as a storage of translations, it allows one to get translations of text to another language, more specifically, to a Locale (check the official Java documentation `java.util.Locale`). In Jtwig, if such message source engine is unable to find a requested translation, then it returns the given text.

> **Key** and **Free Text**
> There are two typical approaches used as input for translations. It is either a **Key** or **Free Text**. The difference between this two approaches is that keys aren't readable, while the provided free text can be. Because keys are not shown to the end-user (at least they shouldn't), they can incorporate contextual information, like the purpose of the text, for example, `registration.title` or `registration.subtitle`. Free text approaches tend to define the wording directly, for example, `Register` or `Please, fill the form below and submit`. Jtwig allows for both approaches, where, for example, properties files can be used with the **Key** approach and Jtwig XLIFF can be used for the **Free Text** one.
>
> We believe translations should be context independent, that is, locating a given text or identifying the purpose of a given text should not be the purpose of a translation mechanism. However, given the size of some platforms, decouple this two concepts can cause more harm than good.

Message Decoration

Message decoration is the engine which allows one to modify the output of the message source. Jtwig translate extension makes use of a replacement strategy as decorator, allowing users to specify a map of replacements to modify the output, such will be detailed further on.

Configuration

```
TranslateConfiguration configuration = TranslateConfigurationBuilder
                .translateConfiguration()
                    .withCurrentLocaleSupplier(currentLocaleSupplier)
                    .withStringLocaleResolver(localeResolver)
                    .withMessageSourceFactory(messageSourceFactory)
                .build();
```

As mentioned this extension is highly configurable, it allows, as seen in the example above, to specify a `LocaleResolver`, a locale supplier and a `MessageSourceFactory`.

Locale Supplier

A locale supplier gives Jtwig the capability to retrieve a locale from the context when no locale is specified. By default Jtwig returns a static supplier which returns `Locale.ENGLISH` as result.

LocaleResolver

The locale resolver is used to convert a raw String to a `java.util.Locale`, by default Jtwig will use the `Locale::forLanguageTag` method provided by the Java API.

MessageSourceFactory

The message source factory gets called when Jtwig is initializing the environment, it can be used to preload resources and provide and instance of the `MessageSource` interface.

```
public interface MessageSourceFactory {
    MessageSource create (Environment environment);
}
```

Jtwig provides several implementations of such factory. A singleton factory `SingletonMessageSourceFactory`, which only returns the provided `MessageSource` instance. A cached factory `CachedMessageSourceFactory` allowing to, given a specific cache implementation, put it in front of the generated `MessageSource`. Shipped within this extension it's also the `PropertiesMessageSourceFactoryBuilder` which give the developer a nice API to create a `MessageSourceFactory` to load messages from properties files.

> **Jtwig XLIFF**
> Jtwig XLIFF was developed as a way to support XLIFF defined translation files. It comes with a `XliffMessageSourceFactoryBuilder` quite similar to the properties message source factory.

Function `translate`

This function has two other alias, they are `trans` and `message`. It expects one argument at least, with the possibility of receiving two extra, optional, arguments. The first mandatory argument is the text to be translated.

```
{{ translate('Hello World') }}
```

If two arguments are provided it can either be a Locale, represented as a string or a map of replacements.

```
{{ translate('Hello World', 'pt') }}
```

```
{{ translate('Hello %name%', {'%name%': 'Jtwig'}) }}
```

If a map is provided Jtwig will use it as replacements applying to the message returned by the message source. Otherwise, if a string is provided, Jtwig will ask the message source mechanism for a given message with the locale provided.

```
{{ translate('Hello World', {'%name%': 'Jtwig'}, 'pt') }}
```

If three arguments are provided, then the map is expected as second argument and a string, representing a locale, the third.

The `trans` tag

The `trans` tag has the same capabilities as the `translate` function allowing to specify the text to translate as body.

```
{% trans %}Hello world{% endtrans %}
```

```
{% trans into 'pt' %}Hello world{% endtrans %}
```

```
{% trans with {'%name%': 'Jtwig'} into 'pt' %}Hello %name%{% endtrans %}
```

```
{% trans with {'%name%': 'Jtwig'} %}Hello %name%{% endtrans %}
```

Note that, the text is trimmed before querying the message source for the translation.

Pluralization

Plural handling in Jtwig it's an easy to use and powerfull engine. Project jtwig-pluralization implements the underlying functionality.

The counter (single)

This whole functionality derives from a key piece of information, the counter. Such counter is used to derive the plural form to be used, as so, this pluralization engine only supports one subject. For example, `1 apple and 2 oranges` contains two subjects, therefore, outside of this engine capabilities, such can be address, for example, by splitting the sentence into two.

```
{0} No apples | {1} One apple | ]1, Inf[ Multiple apples
```

As per the previous example, the plural form contains three definitions (splitted by the `|` character). Definitions start with a range selector, which can either be a single value `{<value>}` or an interval of values, note that intervals can be inclusive or exclusive depending on the parentsis used. For example, `[1, 2]` , `]0, 3[` , `[1, 3[` , `]0, 2]` are all equivalent intervals.

The pluralization engine also trims the selected value. It uses the `String:trim` method underneath.

Function `translateChoice`

This function allows for the same capabilities as the `translate` plus choosing the plural form to use after processing the translation.

```
{{ '{0} No apples | {1} One apple | ]1, Inf[ Multiple apples'
    | translateChoice(numberOfApples, "pt") }}
```

The previous example will look for a `pt` translation of the sentence `{0} No apples | {1} One apple | ]1, Inf[ Multiple apples` and, only after retrieving the translation, uses the pluralization engine to select, based on the counter (here defined as the variable `numberOfApples`), the plural form.

Integration

Integrating Jtwig Translation Extension on your project is quite simple with the help of dependency managers. To check the most recent version, go to bintray.

Gradle

```
repositories {
    jcenter()
}
dependencies {
    compile 'org.jtwig:jtwig-translation-extension:1.X'
}
```

Maven

```
<repositories>
    <repository>
        <id>bintray</id>
        <url>https://jcenter.bintray.com/</url>
    </repository>
</repositories>

<dependencies>
    <dependency>
        <groupId>org.jtwig</groupId>
        <artifactId>jtwig-translation-extension</artifactId>
        <version>1.X</version>
    </dependency>
</dependencies>
```

Examples

Check the jtwig-examples project on github for examples using this translation engine.

Render Extension

The render extension, only available for web apps, enables users to embed the result of rendering a given path. It can be seen as another way to include other resources available throught HTTP GET endpoints. This extension exposes a new tag `render`.

`render` tag

The render tag is expecting a path to an internal resource and, optionally, a map of parameters. It will then create a new `HttpServletRequest` with the parameters provided in memory and ask the `RequestDispatcher` to include the resource with the given path.

```
{% render '/hello' with {name: 'Jtwig'} %}
```

The example above will make a call to render the servlet listening on the path `/hello` with the GET parameters `name=Jtwig`.

Configuration

As shown below, the configuration of this extension is none. One just need to pass it on to the Environment configuration so it will be used.

```
EnvironmentConfigurationBuilder.configuration()
                .extensions().add(new RenderExtension()).and()
                .build()
```

Integration

Integrating Jtwig Render Extension on your project is quite simple with the help of dependency managers. To check the most recent version, go to bintray.

Gradle

```
repositories {
    jcenter()
}
dependencies {
    compile 'org.jtwig:jtwig-render-extension:1.X'
}
```

Maven

```
<repositories>
    <repository>
        <id>bintray</id>
        <url>https://jcenter.bintray.com/</url>
    </repository>
</repositories>

<dependencies>
    <dependency>
        <groupId>org.jtwig</groupId>
        <artifactId>jtwig-render-extension</artifactId>
        <version>1.X</version>
    </dependency>
</dependencies>
```

Examples

Check the jtwig-examples project on github for examples using this render engine.

Json Extension

The JSON extension adds Jtwig the capability of converting Java objects to a String JSON representation. It includes a function `json_encode` to perform such operation.

```
{{ json_encode(object) }}
```

Under the wood, jtwig json extension will, using the configured mapper, convert the given object into a string.

Configuration

```
EnvironmentConfigurationBuilder
    .configuration()
        .extensions()
            .add(new JsonExtension(JsonMapperProviderConfigurationBuilder
                .jsonConfiguration()
                .build())
            )
        .and()
    .build()
```

The configuration allows one to define multiple `JsonMapperProvider`, which supplies an instance of a mapper. The first use of the function `json_encode` will cause the mapper to be resolved and stored as singleton. Jtwig iterates over the list of `JsonMapperProvider` where the first one resolving will be used. By default, Jtwig defines two `JsonMapperProvider`, the `Jackson2JsonMapperProvider` and `JacksonJsonMapperProvider` which use, respectively, Jackson version 2 and 1.

Integration

Integrating Jtwig JSON Extension on your project is quite simple with the help of dependency managers. To check the most recent version, go to bintray.

Gradle

```
repositories {
    jcenter()
}
dependencies {
    compile 'org.jtwig:jtwig-json-extension:1.X'
}
```

Maven

```
<repositories>
    <repository>
        <id>bintray</id>
        <url>https://jcenter.bintray.com/</url>
    </repository>
</repositories>

<dependencies>
    <dependency>
        <groupId>org.jtwig</groupId>
        <artifactId>jtwig-json-extension</artifactId>
        <version>1.X</version>
    </dependency>
</dependencies>
```

Examples

Check the jtwig-examples project on github for examples using this render engine.

Spaceless Extension

The spaceless extension is a tiny extension which add the `spaceless` tag construct to Jtwig. It works with HTML by default, allowing one to remove unnecessary whitespaces.

```
{% spaceless %}
<div>
    <label>Example</label>
</div>
{% endspaceless %}
```

The previous example will produce `<div><label>Example</label>`

Configuration

```
EnvironmentConfigurationBuilder.configuration()
    .extensions()
        .add(SpacelessExtension.defaultSpacelessExtension())
    .and()
.build()
```

The previous example will plugin the Spaceless extension with the default configuration. The only parameter to be configured is an instance of `SpaceRemover` which is used to remove the whitespaces from the content. By default this extension is configured to use the `HtmlSpaceRemover` implementation.

Integration

Integrating Jtwig Spaceless Extension on your project is quite simple with the help of dependency managers. To check the most recent version, go to bintray.

Gradle

```
repositories {
    jcenter()
}
dependencies {
    compile 'org.jtwig:jtwig-spaceless-extension:1.X'
}
```

Maven

```
<repositories>
    <repository>
        <id>bintray</id>
        <url>https://jcenter.bintray.com/</url>
    </repository>
</repositories>

<dependencies>
    <dependency>
        <groupId>org.jtwig</groupId>
        <artifactId>jtwig-spaceless-extension</artifactId>
        <version>1.X</version>
    </dependency>
</dependencies>
```

Examples

Check the jtwig-examples project on github for examples using this spaceless engine.

Glossary

Regular Expression

A regular expression (sometimes abbreviated to "regex") is a way for a computer user or programmer to express how a computer program should look for a specified pattern in text and then what the program is to do when each pattern match is found.

2.2. Expressions

Resource

The concept of resource is very much associated with a file containing a Jtwig template. However it can also be a String representing the template itself. Essencially, a resource is a template container and can take various forms.

7.2. Render 5. Jtwig Core 5.1. Environment 5.2. Resource 3.4.2. import 3.3.2. extends 3.3.1. include 6. Jtwig Web