Table of Contents

Guides

Tags

Filters

- title
- trim
- upper
- urlencode

Functions

- block
- i18n
- max
- min
- parent
- range

Tests

- empty
- even
- map
- null
- odd
- iterable

Operators

- comparisons (`==`, `!=`, `<`, `>`, `<=`, `>=`, `equals`)
- contains (`contains`)
- is
- logic (`and`, `or`, `not`, `()`)
- math (`+`, `-`, `/`, `%`, `*`)
- others (`|`, `?:`)

Installation and Configuration

Installation & Configuration

Installation

Pebble is hosted in the Maven Central Repository. Simply add the following dependency into your `pom.xml` file:

```
<dependency>
        <groupId>io.pebbletemplates</groupId>
        <artifactId>pebble</artifactId>
        <version>3.0.10</version>
</dependency>
```

Also, snapshots of the master branch are deployed automatically with each successful commit. Instead of Maven Central, use the Sonatype snapshots repository at:

```
<url>https://oss.sonatype.org/content/repositories/snapshots</url>
```

You can add the repository in your pom.xml

```
<repositories>
  <repository>
    <id>sonatype-public</id>
    <name>Sonatype Public</name>
    <url>https://oss.sonatype.org/content/groups/public</url>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
  </repository>
</repositories>
```

Set Up

If you are integrating Pebble with Spring MVC, read this guide.

You will want to begin by creating a `PebbleEngine` which is responsible for coordinating the retrieval and compilation of your templates:

```
PebbleEngine engine = new PebbleEngine.Builder().build();
```

And now, with your new `PebbleEngine` instance you can start compiling templates:

```
PebbleTemplate compiledTemplate = engine.getTemplate("templateName");
```

Finally, simply provide your compiled template with a `Writer` object and a Map of variables to get your output!

```
Writer writer = new StringWriter();

Map<String, Object> context = new HashMap<>();
context.put("name", "Mitchell");

compiledTemplate.evaluate(writer, context);
```

```
String output = writer.toString();
```

Template Loader

The `PebbleEngineBuilder` will also accept a `Loader` implementation as an argument. A loader is responsible for finding your templates.

Pebble ships with the following loader implementations:

- `ClasspathLoader`: Uses a classloader to search the current classpath.
- `FileLoader`: Finds templates using a filesystem path.
- `ServletLoader`: Uses a servlet context to find the template. This is the recommended loader for use within an application server but is not enabled by default.
- `StringLoader`: Considers the name of the template to be the contents of the template.
- `DelegatingLoader`: Delegates responsibility to a collection of children loaders.

If you do not provide a custom Loader, Pebble will use an instance of the `DelegatingLoader` by default. This delegating loader will use a `ClasspathLoader` and a `FileLoader` behind the scenes to find your templates.

Pebble Engine Settings

All the settings are set during the construction of the `PebbleEngine` object.

Setting	Description	Default
`cacheActive`	Flag to activate/desactivate template caching	true
`templateCache`	An implementation of a ConcurrentMap cache that the Pebble engine will use to cache compiled templates.	Default implementation is `ConcurrentMapTemplateCache` and another implementation based on Caffeine is available (`CaffeineTemplateCache`)
`tagCache`	An implementation of a ConcurrentMap cache that the Pebble engine will use for cache tag.	Default implementation is `ConcurrentMapTagCache` and another implementation based on Caffeine is available (`CaffeineTagCache`)
`defaultLocale`	The default locale which will be passed to each compiled template. The templates then use this locale for functions such as i18n, etc. A template can also be given a unique locale during evaluation.	`Locale.getDefault()`
`executorService`	An `ExecutorService` that allows the usage of some advanced multithreading features, such as the `parallel` tag.	`null`
`loader`	An implementation of the `Loader` interface which is used to find templates.	An implementation of the `DelegatingLoader` which uses a `ClasspathLoader` and a `FileLoader` behind the scenes.
	If set to true, Pebble will throw an exception if you try to access a	

`strictVariables`	variable or attribute that does not exist (or an attribute of a null variable). If set to false, your template will treat non-existing variables/attributes as null without ever skipping a beat.	`false`
`allowGetClass`	If set to false, Pebble will throw an exception if you try to access the class/getClass attribute.	`true` in 2.x, 'false' in v3.x
`literalDecimalTreatedAsInteger`	option for toggling to enable /disable literal decimal treated as integer	`false`
`greedyMatchMethod`	option for toggling to enable /disable greedy matching mode for finding java method. Reduce the limit of the parameter type, try to find other method which has compatible parameter types.	`false`

Spring Integration

Integration with Spring

Example

There is the spring petclinic example which has been migrated to pebble

There is also a fully working example project located on github which can be used as a reference. It is a very simple and bare-bones project designed to only portray the basics. To build the project, simply run `mvn install` and then deploy the resulting war file to a an application container.

Setup

Pebble has integration for both versions 3.x, 4.x and 5.x of the Spring Framework, provided by three separate libraries called pebble-spring3, pebble-spring4 and pebble-spring5.

First of all, make sure your project includes the `pebble-spring3`, `pebble-spring4` or `pebble-spring5` dependency. This will provide the necessary `ViewResolver` and `View` classes.

```
<dependency>
        <groupId>io.pebbletemplates</groupId>
        <artifactId>pebble-spring{version}</artifactId>
        <version>3.0.10</version>
</dependency>
```

Secondly, make sure your templates are on the classpath (ex. `/WEB-INF/templates/`). Now you want to define a `PebbleEngine` bean and a `PebbleViewResolver` in your configuration.

```
@Configuration
@ComponentScan(basePackages = { "com.example.controller", "com.example.service" })
@EnableWebMvc
public class MvcConfig extends WebMvcConfigurerAdapter {

    @Autowired
    private ServletContext servletContext;

    @Bean
    public Loader templateLoader(){
        return new ServletLoader(servletContext);
    }

    @Bean
    public SpringExtension springExtension() {
        return new SpringExtension();
    }

    @Bean
    public PebbleEngine pebbleEngine() {
         return new PebbleEngine.Builder()
                .loader(this.templateLoader())
                .extension(springExtension())
                .build();
    }

    @Bean
    public ViewResolver viewResolver() {
        PebbleViewResolver viewResolver = new PebbleViewResolver();
```

```
        viewResolver.setPrefix("/WEB-INF/templates/");
        viewResolver.setSuffix(".html");
        viewResolver.setPebbleEngine(pebbleEngine());
        return viewResolver;
    }

}
```

Now the methods in your `@Controller` annotated classes can simply return the name of the template as you normally would if using JSPs:

```
@Controller
@RequestMapping(value = "/profile")
public class ProfileController {

        @Autowired
        private UserService userService;

        @RequestMapping
        public ModelAndView getUserProfile(@RequestParam("id") long id) {
                ModelAndView mav = new ModelAndView();
                mav.addObject("user", userService.getUser(id));
                mav.setViewName("profile");
                return mav;
        }

}
```

The above example will render `\WEB-INF\templates\profile.html` and the "user" object will be available in the evaluation context.

Features

Access to Spring beans

Spring beans are now available to the template.

```
{{ beans.beanName }}
```

Access to http request

HttpServletRequest object is available to the template.

```
{{ request.contextPath }}
```

Access to http response

HttpServletResponse is available to the template.

```
{{ response.contentType }}
```

Access to http session

HttpSession is available to the template.

```
{{ session.maxInactiveInterval }}
```

Spring extension

This extension has many functions for spring validation and the use of message bundle.

Href function

Function to automatically add the context path to a given url

```
<a href="{{ href('/foobar') }}">Example</a>
```

Message function

It achieves the same thing as the i18n function, but instead, it uses the configured spring messageSource, typically the ResourceBundleMessageSource.

```
Label = {{ message('label.test') }}
Label with params = {{ message('label.test.params', 'params1', 'params2') }}
```

Spring validations and error messages

6 validations methods and error messages are exposed using spring BindingResult. It needs as a parameter the form name and for a particular field, the field name.

To check if there's any error:

```
{{ hasErrors('formName' }}

{{ hasGlobalErrors('formName' }}

{{ hasFieldErrors('formName', 'fieldName' }}
```

To output any error:

```
{% for err in getAllErrors('formName') %}
    <p>{{ err }}</p>
{% endfor %}

{% for err in getGlobalErrors('formName') %}
    <p>{{ err }}</p>
{% endfor %}

{% for err in getFieldErrors('formName', 'fieldName') %}
    <p>{{ err }}</p>
{% endfor %}
```

Timer

A timer in PebbleView is available to output the time taken to process a template. Just add the following config to your log4j.xml

```
<Logger name="com.mitchellbosecke.pebble.spring.PebbleView.timer" level="DEBUG" additivit
      <AppenderRef ref="STDOUT" />
</Logger>
```

Spring Boot Integration

Pebble Spring Boot Starter

Spring Boot starter for autoconfiguring Pebble as an MVC ViewResolver.

Basic Usage

Add the starter dependency to your pom.xml:

spring-boot v1

```
<dependency>
        <groupId>io.pebbletemplates</groupId>
        <artifactId>pebble-spring-boot-starter</artifactId>
        <version>3.0.10</version>
</dependency>
```

Or build.gradle:

```
compile "io.pebbletemplates:pebble-spring-boot-starter:3.0.10"
```

spring-boot v2

```
<dependency>
        <groupId>io.pebbletemplates</groupId>
        <artifactId>pebble-spring-boot-2-starter</artifactId>
        <version>3.0.10</version>
</dependency>
```

Or build.gradle:

```
compile "io.pebbletemplates:pebble-spring-boot-2-starter:3.0.10"
```

This is enough for autoconfiguration to kick in. This includes:

- a Loader that will pick template files ending in `.pebble` from `/templates/` dir on the classpath
- a PebbleEngine with default settings, configured with the previous loader
- a ViewResolver that will output `text/html` in `UTF-8`

PLEASE NOTE: the starter depends on `spring-boot-starter-web` but is marked as optional, you'll need to add the dependency yourself or configure Spring MVC appropiately.

Compatibility matrix

Pebble vs tested Boot versions (may work on older Boot releases).

Pebble Boot Starter	Spring Boot
2.2.0+	1.2.1+
2.6.0+	2.0.1+

Boot externalized configuration

A number of properties can be defined in Spring Boot externalized configuration, eg. `application.properties`, starting with the prefix `pebble`. See the corresponding PebbleProperties.java for your starter version. Notable properties are:

- `pebble.prefix`: defines the prefix that will be prepended to the mvc view name. Defaults to `/templates/`
- `pebble.suffix`: defines the suffix that will be appended to the mvc view name. Defaults to `.pebble`
- `pebble.cache`: enables or disables PebbleEngine caches. Defaults to `true`
- `pebble.contentType`: defines the content type that will be used to configure the ViewResolver. Defaults to `text/html`
- `pebble.encoding`: defines the text encoding that will be used to configure the ViewResolver. Defaults to `UTF-8`
- `pebble.exposeRequestAttributes`: defines whether all request attributes should be added to the model prior to merging with the template for the ViewResolver. Defaults to `false`
- `pebble.exposeSessionAttributes`: defines whether all session attributes should be added to the model prior to merging with the template for the ViewResolver. Defaults to `false`
- `pebble.defaultLocale`: defines the default locale that will be used to configure the PebbleEngine. Defaults to `null`
- `pebble.strictVariables`: enable or disable the strict variable checking in the PebbleEngine. Defaults to `false`

Customizing Pebble

Pebble extensions

Extensions defined as beans will be picked up and added to the PebbleEngine automatically:

```
@Bean
public Extension myPebbleExtension1() {
   return new MyPebbleExtension1();
}

@Bean
public Extension myPebbleExtension2() {
   return new MyPebbleExtension2();
}
```

CAVEAT: Spring will not gather all the beans if they're scattered across multiple @Configuration classes. If you use this mechanism, bundle all Extension @Beans in a single @Configuration class.

Customizing the Loader

The autoconfigurer looks for a bean named `pebbleLoader` in the context. You can define a custom loader with that name and it will be used to configure the default PebbleEngine:

```
@Bean
public Loader<?> pebbleLoader() {
   return new MyCustomLoader();
}
```

PLEASE NOTE: this loader's prefix and suffix will be both overwritten when the ViewResolver is configured. You should use the externalized configuration for changing these properties.

Customizing the PebbleEngine

Likewise, you can build a custom engine and make it the default by using the bean name `pebbleEngine`:

```
@Bean
public PebbleEngine pebbleEngine() {
   return new PebbleEngine.Builder().build();
}
```

Customizing the ViewResolver

And the same goes for the ViewResolver, using the bean name `pebbleViewResolver`:

```
@Bean
public PebbleViewResolver pebbleViewResolver() {
   return new PebbleViewResolver();
}
```

PLEASE NOTE: you need to change the Loader's prefix and suffix to match the custom ViewResolver's values.

Using Pebble for other tasks

The main role of this starter is to configure Pebble for generating MVC View results (the typical HTML). You may define more PebbleEngine/Loader beans for other usage patterns (like generating email bodies). Bear in mind that you should not reuse the default Loader for other Engine instances.

Basic Usage

Basic Usage

Introduction

Pebble templates can be used to generate any sort of textual output. It is typically used to generate HTML but it can also be used to create CSS, XML, JS, etc. A template itself will contain whatever language you are attempting to output alongside Pebble-specific features and syntax. Here is a simple example that will generate a trivial HTML page:

```
<html>
        <head>
                <title>{{ websiteTitle }}</title>
        </head>
        <body>
                {{ content }}
        </body>
</html>
```

When you evaluate the template you will provide it with a "context" which is just a map of variables. This context should include the two variables above, `websiteTitle` and `content`.

Set Up

You will want to begin by creating a PebbleEngine object which is responsible for compiling your templates:

```
PebbleEngine engine = new PebbleEngine.Builder().build();
```

And now, with your new PebbleEngine instance you can start compiling templates:

```
PebbleTemplate compiledTemplate = engine.getTemplate("templates/home.html");
```

Finally, simply provide your compiled template with a java.io.Writer object and a Map of variables (the context) to get your output!

```
Writer writer = new StringWriter();

Map<String, Object> context = new HashMap<>();
context.put("websiteTitle", "My First Website");
context.put("content", "My Interesting Content");

compiledTemplate.evaluate(writer, context);

String output = writer.toString();
```

Syntax Reference

There are two primary delimiters used within a Pebble template: `{{ ... }}` and `{% ... %}`. The first set of delimiters will output the result of an expression. Expressions can be very simple (ex. a variable name) or much more complex. The second set of delimiters is used to change the control flow of the template; it can contain an if-statement, define a parent template, define a new block, etc.

Variables

You can print variables directly to the output; for example, if the context contains a variable called `foo` which is a String with the value "bar" you can do the following which will output "bar".

```
{{ foo }}
```

You can use the dot (.) notation to access attributes of variables. If the attribute contains any atypical characters, you can use the subscript notation ([]) instead.

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

Behind the scenes `foo.bar` will attempt the following techniques to to access the `bar` attribute of the `foo` variable:

- If `foo` is a map, `foo.get("bar")`
- `foo.getBar()`
- `foo.isBar()`
- `foo.hasBar()`
- `foo.bar()`
- `foo.bar`

If the value of variable (or attribute) is null it will output an empty string.

Type Safety

Pebble templates are dynamically typed and any possible type safety issues won't occur until the actual runtime evaluation of your templates. Pebble does however allow you to choose how to handle type safety issues with the use of it's `strictVariables` setting. By default, `strictVariables` is set to `false` which means that the following:

```
{{ foo.bar }}
```

will print an empty string even if the object `foo` does not actually have an attribute called `bar`. If `strictVariables` is set to true, the above expression would throw an exception.

When `strictVariables` is set to false your expressions are also null safe. The following expression will print an empty string even if foo and/or bar are null:

```
{{ foo.bar.baz }}
```

The default filter might come in handy for the above situations.

Filters

Output can be further modified with the use of filters. Filters are separated from the variable using a pipe symbol (`|`) and may have optional arguments in parentheses. Multiple filters can be chained and the output of one filter is applied to the next.

```
{{ "If life gives you lemons, eat lemons." | upper | abbreviate(13) }}
```

The above example will output the following:

```
IF LIFE GI...
```

Functions

Whereas filters are intended to modify existing content/variables, functions are intended to generate new content. Similar to other programming languages, functions are invoked via their name followed by parentheses (`()`).

```
{{ max(user.score, highscore) }}
```

Control Structure

Pebble provides several tags to control the flow of your template, two of the main ones being the for loop, and if statements.

```
{% for article in articles %}
    <h3>{{ article.title }}</h3>
    <p>{{ article.content }}</p>
{% else %}
    <p> There are no articles. </p>
{% endfor %}
```

```
{% if category == "news" %}
    {{ news }}
{% elseif category == "sports" %}
    {{ sports }}
{% else %}
    <p>Please select a category</p>
{% endif %}
```

Including other Templates

The include tag is used to include the rendered output of one template into another.

```
<div class="sidebar">
        {% include "advertisement.html" %}
</div>
```

Template Inheritance

Template inheritance is the most powerful feature of Pebble. It allows templates to override sections of their parent template. In your parent template you define "blocks" which are the sections that are allowed to be overriden.

First let us look at an example of a parent template:

```
<html>
<head>
        <title>{% block title %}My Website{% endblock %}</title>
</head>
<body>
        <div id="content">
                {% block content %}{% endblock %}
        </div>
        <div id="footer">
                {% block footer %}
                        Copyright 2013
                {% endblock %}
        </div>
</body>
</html>
```

In the above example, we have used the block tag to define several sections that child templates are allowed to override.

A child template might look like this:

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

{% block title %} Home {% endblock %}

{% block content %}
        <h1> Home </h1>
        <p> Welcome to my home page.</p>
{% endblock %}
```

The first line uses the extends tag to declare the parent template. The extends tag should be the first tag in the template and there can only be one.

Evaluating the child template will produce the following output:

```
<html>
<head>
        <title>Home</title>
</head>
<body>
        <div id="content">
                <h1> Home </h1>
                <p> Welcome to my home page.</p>
        </div>
        <div id="footer">
                Copyright 2013
        </div>
</body>
</html>
```

You may have noticed that in the above example, because the child template doesn't override the `footer` block, the value from the parent is used instead.

Dynamic inheritance is possible by using an expression with the `extends` tag:

```
{% extends ajax ? 'ajax.html' : 'base.html' %}
```

Macros

Macros are lightweight and reusable template fragments. A macro is defined via the macro tag:

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

And the macro will be invoked just like a function:

```
{{ input("text", "name", "Mitchell") }}
```

Child templates will have access to macros defined in a parent template. To use macros located in a completely different template, you can use the import) tag. A macro does not have access to the main context; the only variables it can access are it's local arguments.

Named Arguments

Using named arguments allows you to be more explicit with the values you are passing to a filter, function, test or macro. They also allow you to avoid specifying arguments for which you don't want to change the default value.

```
{{ stringDate | date(existingFormat="yyyy-MMMM-d", format="yyyy/MMMM/d") }}
```

Positional arguments can be used in conjunction with named arguments but all positional arguments must come before any named arguments:

```
{{ stringDate | date("yyyy/MMMM/d", existingFormat="yyyy-MMMM-d") }}
```

Macros are a great use case for named arguments because they also allow you to define default values for unused arguments:

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

{{ input(name="country") }}

{# will output: <input type="text" name="country" value="" /> #}
```

Escaping

XSS vulnerabilites are the most common types of security vulnerabilities in web applications and in order to avoid them you must escape potentially unsafe data before presenting it to the end user. Pebble provides autoescaping of all such data which is enabled by default. Autoescaping can be turned off, in which case Pebble provides an escape filter for more fine-grained manual escaping.

The following is an example of how autoescaping will escape your context variables:

```
{% set danger = "<br>" %}
{{ danger }}

{# will output: &lt;br&gt; #}
```

If autoescaping is disabled you can still use the escape filter to aid with manual escaping:

```
{% set danger = "<br>" %}
{{ danger | escape }}

{# will output: &lt;br&gt; #}
```

By default, the autoescaping mechanism and the escape filter assume that it is escaping within an HTML context. You may want to use an alternate escaping strategy depending on the context:

```
{% set danger = "alert(...)" %}
<script>var username="{{ danger | escape(strategy="js") }}"</script>
```

See the escaping guide for more information on how autoescaping works, how to disable it, and the various escaping strategies that are available.

Whitespace

The first newline after a pebble tag is automatically ignored; all other whitespace is ignored by Pebble and will be included in the rendered output.

Pebble provides a whitespace control modifier to trim leading or trailing whitespace adjacent to any pebble tag.

```
<p>                 {{- "no whitespace" -}}                </p>
{# output: "<p>no whitespace</p>" #}
```

It is also possible to only use the modifier on one side of the tag:

```
<p>                 {{- "no leading whitespace" }}                 </p>
{# output: "<p>no whitespace                 </p>" #}
```

Comments

You can comment out any part of the template using the `` delimiters. These comments will not appear in the rendered output.

```
{# THIS IS A COMMENT #}
{% for article in articles %}
        <h3>{{ article.title }}</h3>
        <p>{{ article.content }}</p>
{% endfor %}
```

Expressions

Expressions in a Pebble template are very similar to expressions found in Java.

Literals

The simplest form of expressions are literals. Literals are representations for Java types such as strings and numbers.

- `"Hello World"`: Everything between two double or single quotes is a string. You can use a backslash to escape quotation marks within the string.
- `"Hello #{who}"`: String interpolation is also possible using `#{}` inside quotes. In this example, if the value of the variable `who` is `"world"`, then the expression will be evaluated to `"Hello world"`.
- `100 + 10l * 2.5`: Integers, longs and floating point numbers are similar to their Java counterparts.
- `true` / `false`: Boolean values equivalent to their Java counterparts.
- `null`: Represents no specific value, similar to it's Java counterpart. `none` is an alias for null.

Collections

Both lists and maps can be created directly within the template.

- `["apple", "banana", "pear"]`: A list of strings
- `{"apple":"red", "banana":"yellow", "pear":"green"}`: A map of strings

The collections can contain expressions.

Math

Pebble allows you to calculate values using some basic mathematical operators. The following operators are supported:

- `+`: Addition
- `-`: Subtraction
- `/`: Division
- `%`: Modulus
- `*`: Multiplication

Logic

You can combine multiple expressions with the following operators:

- `and`: Returns true if both operands are true
- `or`: Returns true if either operand is true
- `not`: Negates an expression
- `(...)`: Groups expressions together

Comparisons

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

```
{% if user.age >= 18 %}
        ...
{% endif %}
```

Tests

The `is` operator performs tests. Tests can be used to test an expression for certain qualities. The right operand is the name of the test:

```
{% if 3 is odd %}
        ...
{% endif %}
```

Tests can be negated by using the is not operator:

```
{% if name is not null %}
        ...
{% endif %}
```

Conditional (Ternary) Operator

The conditional operator is similar to it's Java counterpart:

```
{{ foo ? "yes" : "no" }}
```

Operator Precedence

In order from highest to lowest precedence:

- `.`
- `|`
- `%`, `/`, `*`
- `-`, `+`
- `==`, `!=`, `>`, `<`, `>=`, `<=`
- `is`, `is not`
- `and`
- `or`

IDE's plugin

If you want to add IDE's syntax highlighting, you can install this plugin for IntelliJ. Thank you to Bastien Jansen for his contribution.

Escaping

Escaping

Overview

XSS vulnerabilites are the most common types of security vulnerabilities in web applications and in order to avoid them you must escape potentially unsafe data before presenting it to the end user. Pebble provides autoescaping of all such data which is enabled by default. Autoescaping can be turned off, in which case Pebble provides an escape filter for more fine-grained manual escaping.

Autoescaping

Autoescaping, which is enabled by default, will automatically escape the outcome of expressions contained within print delimiters, i.e. `{{` and `}}`:

```
{% set danger = "<br>" %}
{{ danger }}

{# will output: &lt;br&gt; #}
```

The raw filter can be used to prevent the autoescaper from escaping a particular expression. It is important that the raw filter is the last operation performed in the expression.

```
{% set danger = "<br>" %}
{{ danger | raw }}

{# will output: <br> #}
```

If the raw filter is not the last operation performed within the expression, the expression will be deemed as possibly unsafe by the autoescaper and will be escaped. For example:

```
{% set danger = "<br>" %}
{{ danger | raw | uppercase }}

{# will output: &lt;BR&gt; #}
```

Exceptions

There are a few exceptions where expressions are **not** automatically escaped:

- If the expression only contains a string literal, it is assumed to be safe. For example:

```
{{ '<br>' }}

{# will output: <br> #}
```

- The last operation contained within that expression is a filter or function that explicitly returns safe output. Such a filter or function would return an instance of `SafeString` instead of a regular String. The built-in filters that return safe markup include: `date`, `escape`, and `raw`. These filters must be the last operation performed within the expression in order for their output to be ignored by the autoescaper. For example:

```
{% set danger = "<br>" %}
{{ danger | uppercase | raw }}
```

```
{# will output: <br> #}
```

Autoescape Tag

The autoescape tag can be used to temporarily disable/re-enable the autoescaper as well as change the escaping strategy for a portion of the template.

```
{{ danger }} {# will be escaped by default #}
{% autoescape false %}
        {{ danger }} {# will not be escaped #}
{% endautoescape %}

{{ danger }} {# will use the "html" escaping strategy #}
{% autoescape "js" %}
        {{ danger }} {# will use the "js" escaping strategy #}
{% endautoescape %}
```

Disabling Autoescaper

```
PebbleEngine engine = new PebbleEngine.Builder().autoEscaping(false).build();
```

Manual Escaping

If autoescaping is disabled you can still use the escape filter to aid with manual escaping:

```
{% set danger = "<br>" %}
{{ danger | escape }}

{# will output: &lt;br&gt; #}
```

Strategies

When escaping data it is crucial that you utilize the correct escaping strategy depending on the context of the data. By default, the autoescaper and the `escape` filter assume that you are escaping HTML data. I highly recommend reading the OWASP Cheat Sheet to understand the significance of escaping context.

Pebble provides the following escaping strategies:

- html
- js
- css
- url_param

You can use the autoescape tag to temporarily change the strategy used by the autoescaper otherwise you can change the globally used default strategy:

```
PebbleEngine engine = new PebbleEngine.Builder().defaultEscapingStrategy("js").build();
```

The escape filter will also accept a strategy as an argument:

```
var username ="{{ user.name | escape(strategy="js") }}";
```

Custom Strategy

You can add a custom escaping strategy by implementing `EscapingStrategy` and adding it to the `EscaperExtension`:

```
PebbleEngine engine = new PebbleEngine.Builder().addEscapingStrategy("custom", new Custor
```

Extending Pebble

Extending Pebble

Overview

Pebble was designed to be flexible and accomodate the requirements of any project. You can add your own tags, functions, operators, filters, tests, and global variables. The majority of these are quite trivial to implement.

Begin by creating a class that implements `Extension`. For your own convenience, I recommend extending `AbstractExtension` if you can. After implementing the required methods, register your extension with the `PebbleEngine` before compiling any templates:

```
PebbleEngine engine = new PebbleEngine.Builder().extension(new CustomExtension()).build(
```

Filters

To create custom filters, implement the `getFilters()` method of your extension which will return a map of filter names and their corresponding implementations. A filter implementation must implement the `Filter` interface. The `Filter` interface requires two methods to be implemented, `getArgumentNames()` and `apply()`. The `getArgumentNames()` method returns a list of Strings that define both the order and names of expected arguments.

The `apply` method is the actual filter implementation. Here's the parameters definition.

Parameter name	Description
input	the data to be filtered
args	the map of arguments the user may have provided
self	An instance of `PebbleTemplate` which can be used to retrieve the template name for example
context	An instance of `EvaluationContext` which can be used to retrieve the locale for example
lineNumber	Useful when throwing exception to provide line number

Because Pebble is dynamically typed, you will have to downcast the arguments to the expected type. Here is an example of how the upper filter might be implemented:

```
public UpperFilter implements Filter {

        @Override
        public List<String> getArgumentNames() {
                return null;
        }

        @Override
        public Object apply(Object input, Map<String, Object> args, PebbleTemplate self,
                if(input == null){
                        return null;
                }
                if (input instanceof String) {
                        return ((String) input).toUpperCase(context.getLocale());
```

```
                } else {
                        return input.toString().toUpperCase(context.getLocale());
                }
        }

}
```

Tests

Adding custom tests is very similar to custom filters. Implement the `getTests()` method within your extension which will return a map of test names and their corresponding implementations. A test implementation will implement the `Test` interface. The `Test` interface is exactly like the `Filter` interface except the apply method returns a boolean instead of an arbitrary object of any type.

Here is an example of how the even test might be implemented:

```
public EvenTest implements Test {

        @Override
        public List<String> getArgumentNames() {
                return null;
        }

        @Override
        public boolean apply(Object input, Map<String, Object> args, PebbleTemplate self,
                if (input == null) {
                        throw new PebbleException(null, "Can not pass null value to \"eve
                }

                if (input instanceof Integer) {
                        return ((Integer) input) % 2 == 0;
                } else {
                        return ((Long) input) % 2 == 0;
                }
        }

}
```

Functions

Adding functions is also very similar to custom filters. First and foremost, it's important to understand the different intentions behind a function and a filter because it can often be ambiguous which one should be implemented. A filter is intended to modify existing content where a function is moreso intended to produce new content.

To add functions, implement the `getFunctions()` method within your extension which will return a map of function names and their corresponding implementations. A function implementation will implement the `Function` interface. The `Function` interface is very similar to the `Filter` and `Test` interfaces.

Here is an example of how a fictional `fibonacciString` function might be implemented:

```
public FibonnaciStringFunction implements Function {

        @Override
        public List<String> getArgumentNames() {
                List<String> names = new ArrayList<>();
                names.add("length");
                return names;
        }

        @Override
        public Object execute(Map<String, Object> args, PebbleTemplate self, EvaluationCo
```

```
                Integer length = (Integer)args.get("length");
                Integer prev1 = 0;
                Integer prev2 = 1;

                StringBuilder result = new StringBuilder();

                result.append("01");

                for(int i = 2; i < length; i++){
                        Integer next = prev1 + prev2;
                        result.append(next);
                        prev1 = prev2;
                        prev2 = next;
                }
                return result.toString();

        }
}
```

Positional and Named Arguments

For filters, tests, and functions it is required that you implement the `getArgumentNames` method even if it returns null. Returning a list of strings will allow the end user to call your filter/test/function using named arguments. Using the above fictional fibonacci function as an example, a user can invoke it in two different ways:

```
{{ fibonacci(10) }}
{{ fibonacci(length=10) }}
```

If the end user excludes the names and only uses positional arguments, the argument values will still end up be mapped to the proper names when it's time to invoke the function's execute method. Your function implementation doesn't have to worry whether the user used positional or named arguments. It is important though that if the filter/function/test expects more than one argument, then the developer must communicate to the user the expected order of arguments in the chance that the user wants to invoke it without using names.

Some functions such as the built in `min` and `max` functions accept an unlimited amount of arguments. For this to happen, your function must not accept any named arguments (i.e. your `getArgumentNames` method will return null or empty) and your `execute`` method will simply iterate over the values of the user provided argument map while ignoring the keys of that map (Pebble will use arbitrary keys if there are no names to map to).

Global Variables

Adding global variables, which are variables that are accessbile to all templates, is very trivial. In your custom extension, implement the `getGlobalVariables()` method which returns a `Map<String,Object>`. The contents of this map will be merged into the context you provide to each template at the time of rendering.

Operators

Operators are more complex to implement than filters or tests. To add custom operators, implement the `getBinaryOperators()` or the `getUnaryOperators()` method in your extension, or both. These methods return a list of `BinaryOperator` or `UnaryOperator` objects, respectively.

Binary operators require the following information:

- Precedence: an integer relative to other operators which defines the order of operations.

- Symbol: a String representing the actual operator. This is typically a single character but doesn't have to be.
- Expression Class: A class that extends `BinaryExpression`. This class will perform the actual operator implementation.
- Associativity: Either left or right depending on how the operator is used.

A unary operator is much the same except it's expression class must extend `UnaryExpression` and there is no associativity.

The precedence values for existing core operators are as followed:

- `or`: 10
- `and`: 15
- `is`: 20
- `is not`: 20
- `==`: 30
- `!=`: 30
- `>`: 30
- `<`: 30
- `>=`: 30
- `<=`: 30
- `+`: 40
- `-`: 40
- `not`: 50 (Unary)
- `*`: 60
- `/`: 60
- `%`: 60
- `|`: 100
- `+`: 500 (Unary)
- `-`: 500 (Unary)

The following is an example of how the addition operator (+) might have been implemented:

```
public AdditionOperator implements BinaryOperator {

        public int getPrecedence(){
                return 30;
        }

        public String getSymbol(){
                return "+";
        }

        public Class<? extends BinaryExpression<?>> getNodeClass(){
                return AdditionExpression.class;
        }

        public Associativity getAssociativity(){
                return Associativity.LEFT;
        }

}
```

Alongside each operator class you will also need to implement a corresponding `BinaryExpression` class which actually implements the operator. The above example references a fictional `AdditionExpression` class which might look like the following:

```
public AdditionExpression extends BinaryExpression<Object> {

        @Override
        public Object evaluate(PebbleTemplateImpl self, EvaluationContext context){
```

```
                Integer left = (Integer)getLeftExpression().evaluate(self, context);
                Integer right = (Integer)getRightExpression().evaluate(self, context);

                return left + right;
        }

}
```

In the above example you will notice that children of BinaryExpression have access to two other expressions, `leftExpression`, and `rightExpression`; these are the operands of your operator. Please note that in the above example both operands are casted to Integers but in reality you can't always make that assumption; the true addition expression is much more complex to handle different types of operands (Integers, Longs, Doubles, etc).

Tags

Creating new tags is one of the most powerful abilities of Pebble. Your extension should start by implementing the `getTokenParsers()` method. A `TokenParser` is responsible for converting all necessary tokens to appropriate `RenderableNodes`. A token is a significant and irreducible group of characters found in a template (such as an operator, whitespace, variable name, delimiter, etc) and a `RenderableNode` is a Pebble class that is responsible for generating output.

Let us look at an example of a `TokenParser`:

```
public SetTokenParser extends AbstractTokenParser {

        public String getTag(){
                return "set";
        }

        public RenderableNode parse(Token token) throws SyntaxException {
                TokenStream stream = this.parser.getStream();
                int lineNumber = token.getLineNumber();

                // skip the "set" token
                stream.next();

                // use the built in expression parser to parse the variable name
                NodeExpressionNewVariableName name = this.parser.getExpressionParser().pa

                stream.expect(Token.Type.PUNCTUATION, "=");

                // use the built in expression parser to parse the variable value
                Expression<?> value = this.parser.getExpressionParser().parseExpression(

                // expect to see "%}"
                stream.expect(Token.Type.EXECUTE_END);

                // NodeSet is composed of a name and a value
                return new SetNode(lineNumber, name, value);
        }

}
```

The `getTag()` method must return the name of the tag. Pebble's main parser will use this name to determine when to delegate responsibility to your custom `TokenParser`. This example is parsing the `set` tag.

The parse method is invoked whenever the primary parser encounters a set token. This method should return one `RenderableNode` instance which when rendered during the template evaluation, will write output to the provided Writer object. If the `RenderableNode` contains children nodes, it should invoke the render method of those nodes as well.

The best way to learn all the details of parsing is to look at some of the tools used, as well as some examples. Here is a list of classes I suggest reading:

- `TokenParser`
- `Parser`
- `SetTokenParser`
- `ForTokenParser`
- `IfNode`
- `SetNode`

Attribute resolver (v3 only)

To create a new attribute resolver, implement the `getAttributeResolver()` method of your extension which will return a list of attribute resolvers to run. A attribute resolver implementation must implement the `AttributeResolver` interface. The `AttributeResolver` interface requires one method to be implemented, `resolve()`.

The custom attribute resolver will be executed before all default pebble attribute resolvers. It replaces the `DynamicAttributeProvider` interface

```
public class DefaultAttributeResolver implements AttributeResolver {

  @Override
  public ResolvedAttribute resolve(Object instance,
                                   Object attributeNameValue,
                                   Object[] argumentValues,
                                   boolean isStrictVariables,
                                   String filename,
                                   int lineNumber) {
    if (instance instanceof CustomObject) {
      return "customValue"
    }
    return null;
  }
}
```

High Performance Techniques

High Performance

Concurrency

First and foremost, a `PebbleTemplate` object, once compiled, is completely thread safe. As long as the data backing the template is also thread safe, you can render that single template instance using multiple threads at once.

The actual rendering of a template will typically occur in a sequential manner, from top to bottom. If, however, you provide an `ExecutorService` to the `PebbleEngine` and make use of the parallel tag, you can have multiple threads render different sections of your template at one time. This is especially useful if one section of your template is costly and will otherwise block the rendering of the rest of the template.

Streaming

The use of the flush tag can be used to stream the rendered output as it's being rendered. This can significantly improve latency.

Performance Pitfalls

- It is typically okay for a block to use the `flush` tag unless the contents of that block is being rendered using the block function. Typically the flush tag will flush to the `Writer` that you provided but the block function internally uses it's own `StringWriter` and therefore flushing will do no good.

autoescape

`autoescape`

The `autoescape` tag can be used to temporarily disable/re-enable the autoescaper as well as change the escaping strategy for a portion of the template.

```
{{ danger }} {# will be escaped by default #}
{% autoescape false %}
        {{ danger }} {# will not be escaped #}
{% endautoescape %}

{{ danger }} {# will use the "html" escaping strategy #}
{% autoescape "js" %}
        {{ danger }} {# will use the "js" escaping strategy #}
{% endautoescape %}
```

Please read the escaping guide for more information about escaping.

block

`block`

The `block` tag performs two functions. If used in a parent template, it will designate a section as being allowed to be overriden by a child template. If used in a child template, it will override the content originally declared in the parent template. See the extends tag for a more detailed explanation on how to implement template inheritance.

The contents of a block will only be used if a child template does not override it. It is often useful to define empty blocks as placeholders for content to be provided by a child template.

The `block` tag is immediately followed by the name of the block. This name will be the same name the child template uses to override it. The `endblock` tag can optionally contain the block's name for readability.

In the following example we create a block with the name 'header':

```
{% block header %}
        <h1> Introduction </h1>
{% endblock header %}
```

A child template should not have any content outside of blocks. A child template is only used to override blocks of a parent template.

cache

cache

Cache the rendering portion of a page. Cache name can be an expression or a static string. It uses the cache name and the locale as a key in the cache.

In the following example we create a cache with the name 'menu':

```
{% cache 'menu' %}
    {% for item in items %}
        {{ item.text }}
        ....
    {% endfor %}
{% endcache %}
```

Cache implementation can be overriden with the PebbleEngine Builder.

```
 return new PebbleEngine.Builder()
                .loader(this.templateLoader())
                .tagCache(CacheBuilder.newBuilder().maximumSize(200).build())
                .build();
```

embed

embed

The `embed` tag allows you to insert the rendered output of another template directly into the current template, while overriding some of its blocks. It effectively combines the behavior of include with that of extends for creating reusable, yet flexible, template fragments, or for composing micro-layouts.

For example, imagine building a template `card.peb` as a reusable component in your layout. All cards should have the same markup, but the content can change drastically throughout your site. `card.peb` might then look like:

```
// card.peb
<div class="card">
    {% block cardContent %}
    {% endblock %}
</div>
```

Now, you can include that template elsewhere in your layout, and override the `cardContent` block to "inject" rich content into that template at the call-side. For example, you may want to display a grid of your store's most popular products as cards, with the last card linking to the full catalog. Embedding `card.peb` and overriding the `cardContent` block ensures that the markup for both types of cards are always the same, even though what's displayed on each card is quite different.

```
// layout.peb

{% for product in popularProducts %}
    {% embed 'card.peb' %}
        {% block cardContent %}
            <h1>{{ product.name }}</h1>
            <p>{{ product.description }}</p>
        {% endblock %}
    {% endembed %}
{% endfor %}

{% embed 'card.peb' %}
    {% block cardContent %}
        <a href="...">See all 100+ products</a>
    {% endblock %}
{% endembed %}
```

Embeds can be used multiple times in the same template, and may also be used in a template that itself extends another. Each template will then maintain its own block hierarchy. In other words, block overridden within the body of the `embed` tag will not accidentally override those defined in the main template, and likewise blocks defined in the main template or its parent templates will not get mixed with those in the embedded template or its parent templates.

```
// main.peb
{% extends 'base.peb' %}

{% block mainContent %}
    {{ parent() }} {# renders mainContent block from base.peb #}
    {{ block('footer') }} {# renders footer block from base.peb, the global page footer #

    {% embed 'card.peb' %}
        {% block mainContent %}
            {{ parent() }} {# renders mainContent block from card.peb #}
            {{ block('footer') }} {# renders footer block from card.peb, the card footer
        {% endblock %}
```

```
    {% endembed %}
{% endblock %}
```

Scope

Embedded templates will have access to the same variables that the current template does.

```
Top Content
{% embed "advertisement" %}{% endembed %}
Bottom Content
{% embed "footer" %}{% endembed %}
```

You can add additional variables to the context of the embedded template by passing a map after the `with` keyword. The embedded template will have access to the same variables that the current template does plus the additional ones defined in the map passed after the `with` keyword:

```
{% embed "advertisement" with {"foo":"bar"} %}
    {% block title %}
        Ad with title
    {% endblock %}
    {% block content %}
        Ad with title
    {% endblock %}
{% endembed %}
```

Dynamic embed

The `embed` tag will accept an expression to determine the template to embed at runtime. For example:

```
{% embed admin ? 'adminFooter' : 'defaultFooter' %}
{% endembed %}
```

extends

`extends`

The `extends` tag is used to declare a parent template. It should be the very first tag used in a child template and a child template can only extend up to one parent template.

The best way to understand template inheritance is to study an example. Let us look at a parent template called "base":

```
<html>
        <head>
                <title>{% block title %} {% endblock %}</title>
        </head>
        <body>
                <div id="content">
                        {% block content %}
                                Default content goes here.
                        {% endblock %}
                </div>

                <div id="footer">
                        {% block footer %}
                                Default footer content
                        {% endblock %}
                </div>
        </body>
</html>
```

And now let's look at a child template called "home" which extends "base":

```
{% extends "base" %}

{% block title %} Home {% endblock %}

{% block content %}
        Home page content.
{% endblock %}
```

And finally let's look at the resulting output after evaluating "home":

```
<html>
        <head>
                <title> Home </title>
        </head>
        <body>
                <div id="content">
                        Home page content will override the default content.
                </div>

                <div id="footer">
                        Default footer content
                </div>
        </body>
</html>
```

To summarize, parent templates define blocks and child templates will override the contents of those blocks. If a child template does not override the content of a particular block, the content provided by the parent template will be used.

There is no limit to how long of an inheritance chain that you can create; i.e. a child template can itself have a child template. A lot of potential comes from this fact because you can create a hierarchy of templates to minimize how much content you have to write on the lower levels.

Dynamic Inheritance

The `extends` tag will accept an expression to determine the parent template at runtime. For example:

```
{% extends ajax ? 'ajax' : 'base' %}
```

filter

`filter`

The `filter` tag allows you to apply a filter to a large chunk of template.

```
{% filter upper %}
        hello
{% endfilter %}}

{# output: 'HELLO' #}
```

Multiple filters can be chained together.

```
{% filter upper | escape %}
        hello<br>
{% endfilter %}}

{# output: 'HELLO&lt;br&gt;' #}
```

flush

`flush`

The `flush` tag allows you to flush all currently rendered output to the provided `Writer`.

```
{{ headerText }}
{% flush %}
{{ content }}
```

for

`for`

The `for` tag is used to iterate through primitive arrays or anything that implements the `java.lang.Iterable` interface, as well as maps.

```
{% for user in users %}
        {{ user.name }} lives in {{ user.city }}.
{% endfor %}
```

While inside of the loop, Pebble provides a couple of special variables to help you out:

- loop.index - a zero-based index that increments with every iteration.
- loop.length - the size of the object we are iterating over.
- loop.first - True if first iteration
- loop.last - True if last iteration
- loop.revindex - The number of iterations from the end of the loop

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

The `for` tag also provides a convenient way to check if the iterable object is empty with the included `else` tag.

```
{% for user in users %}
        {{ loop.index }} - {{ user.id }}
{% else %}
        There are no users to display.
{% endfor %}
```

Iterating over maps can be done like so:

```
{% for entry in map %}
    {{ entry.key }} - {{ entry.value }}
{% endfor %}
```

from

`from`

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

if

`if`

The `if` tag allows you to designate a chunk of content as conditional depending on the result of an expression

```
{% if users is empty %}
        There are no users.
{% elseif users.length == 1 %}
        There is only one user.
{% else %}
        There are many users.
{% endif %}
```

The expression used in the `if` statement often makes use of the is operator.

Supported conditions

`If` tag currently supports the following expression

Value	Boolean expression
boolean	boolean value
Empty string	false
Non empty string	true
numeric zero	false
numeric different than zero	true

import

`import`

The `import` tag allows you to use macros defined in another template.

Assuming that a macro named `input` exists in a template called `form_util` you can import it like so:

```
{% import "form_util" %}

{{ input("text", "name", "Mitchell") }}
```

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

```
{% 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:

```
{% 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>
```

Dynamic Import

The `import` tag will accept an expression to determine the template to import at runtime. For example:

```
{% import modern ? 'ajax_form_util' : 'simple_form_util' %}

{{ input("text", "name", "Mitchell") }}
```

include

`include`

The `include` tag allows you to insert the rendered output of another template directly into the current template. The included template will have access to the same variables that the current template does.

```
Top Content
{% include "advertisement" %}
Bottom Content
{% include "footer" %}
```

You can add additional variables to the context of the included template by passing a map after the `with` keyword. The included template will have access to the same variables that the current template does plus the additional ones defined in the map passed after the `with` keyword:

```
{% include "advertisement" with {"foo":"bar"} %}
```

Dynamic Include

The `include` tag will accept an expression to determine the template to include at runtime. For example:

```
{% include admin ? 'adminFooter' : 'defaultFooter' %}
```

macro

`macro`

The `macro` tag allows you to create a chunk of reusable and dynamic content. The macro can be called multiple times in the current template or even from another template with the help of the import tag.

It doesn't matter where in the current template you define a macro, i.e. whether it's before or after you call it. Here is an example of how to define a macro:

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

And now the macro can be called numerous times throughout the template, like so:

```
{{ input(name="country") }}
{# will output: <input type="text" name="country" value="" /> #}
```

If the macro resides in another template, use the import tag first.

```
{% import "form_util" %}
{{ input("text", "country", "Canada") }}
```

A macro does not have access to the same variables that the rest of the template has access to. A macro can only work with the variables provided as arguments.

Access to the global context

You can pass the whole context as an argument by using the special '_context' variable if you need to access variables outside of the macro scope

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

{{ test(_context) }}
{% macro test(_context) %}
        {{ foo }}
{% endmacro %}

{# will output: bar #}
```

parallel

`parallel`

The `parallel` tag allows you to designate a chunk of content to be rendered using a new thread. This tag is only available if you provide an `ExecutorService` to the main `PebbleEngine`.

```
{{ upperContent }}

{% parallel %}
    {{ calculation.slowCalculation }}
{% endparallel %}

{{ lowerContent }}
```

In the above example, the slow calculation will not block the `lowerContent` from being evaluated concurrently.

See the high performance guide for more tips on how to improve performance.

set

`set`

The `set` tag allows you to define a variable in the current context, whether it currently exists or not.

```
{% set header = "Test Page" %}

{{ header }}
```

verbatim

`verbatim`

The `verbatim` tag allows you to write Pebble syntax that won't be parsed.

```
{% verbatim %}
        {% for user in users %}
                {{ user.name }}
        {% endfor %}}
{% endverbatim %}
```

abbreviate

`abbreviate`

The `abbrevate` filter will abbreviate a string using an ellipsis. It takes one argument which is the max width of the desired output including the length of the ellipsis.

```
{{ "this is a long sentence." | abbreviate(7) }}
```

The above example will output the following:

```
this...
```

Arguments

- length

abs

abs

The `abs` filter is used to obtain the absolute value.

```
{{ -7 | abs }}

{# output: 7 #}
```

capitalize

`capitalize`

The `capitalize` filter will capitalize the first letter of the string.

```
{{ "article title" | capitalize }}
```

The above example will output the following:

```
Article title
```

See also: title

date

`date`

The `date` filter is used to format an existing `java.util.Date` object. The filter will construct a `java.text.SimpleDateFormat` using the provided pattern and then use this newly created `SimpleDateFormat` to format the provided `Date` or `java.lang.Number` object.

```
{{ user.birthday | date("yyyy-MM-dd") }}
```

The alternative way to use this filter is to use it on a string but then provide two arguments: first is the desired pattern for the output and the second is the existing format used to parse the input string into a `java.util.Date` object.

```
{{ "July 24, 2001" | date("yyyy-MM-dd", existingFormat="MMMM dd, yyyy") }}
```

The above example will output the following:

```
2001-07-24
```

Arguments

- format
- existingFormat

default

`default`

The `default` filter will render a default value if and only if the object being filtered is empty. A variable is empty if it is null, an empty string, an empty collection, or an empty map.

```
{{ user.phoneNumber | default("No phone number") }}
```

In the following example, if `foo`, `bar`, or `baz` are null the output will become an empty string which is a perfect use case for the default filter:

```
{{ foo.bar.baz | default("No baz") }}
```

Note that the default filter will suppress any `AttributeNotFoundException` exceptions that will usually be thrown when `strictVariables` is set to `true.

Arguments

- default

escape

`escape`

The `escape` filter will turn special characters into safe character references in order to avoid XSS vulnerabilities. This filter will typically only need to be used if you've turned off autoescaping.

```
{{ "<div>" | escape }}
{# output: &lt;div&gt; #}
```

Please read the escaping guide for more information about escaping.

Arguments

- strategy

first

`first`

The `first` filter will return the first item of a collection, or the first letter of a string.

```
{{ users | first }}
{# will output the first item in the collection named 'user' #}

{{ 'Mitch' | first }}
{# will output 'M' #}
```

join

`join`

The `join` filter will concatenate all items of a collection into a string. An optional argument can be given to be used as the separator between items.

```
{#
    List<String> names = new ArrayList<>();
    names.add("Alex");
    names.add("Joe");
    names.add("Bob");
#}
{{ names | join(',') }}
{# will output: Alex,Joe,Bob #}
```

Arguments

- separator

last

`last`

The `last` filter will return the last item of a collection, or the last letter of a string.

```
{{ users | last }}
{# will output the last item in the collection named 'user' #}

{{ 'Mitch' | last }}
{# will output 'h' #}
```

length

`length`

The `length` filter returns the number of items of collection, map or the length of a string:

```
{% if users|length > 10 %}
    ...
{% endif %}
```

lower

`lower`

The `lower` filter makes an entire string lower case.

```
{{ "THIS IS A LOUD SENTENCE" | lower }}
```

The above example will output the following:

```
this is a loud sentence
```

numberformat

`numberformat`

The `numberformat` filter is used to format a decimal number. Behind the scenes it uses `java.text.DecimalFormat`.

```
{{ 3.141592653 | numberformat("#.##") }}
```

The above example will output the following:

```
3.14
```

Argumnets

- format

raw

`raw`

The `raw` filter prevents the output of an expression from being escaped by the autoescaper. The `raw` filter must be the very last operation performed within the expression otherwise the autoescaper will deem the expression as potentially unsafe and escape it regardless.

```
{% set danger = "<div>" %}
{{ danger | upper | raw }}
{# ouptut: <DIV> #}
```

If the `raw` filter is not the last operation performed then the expression will be escaped:

```
{% set danger = "<div>" %}
{{ danger | raw | upper }}
{# output: &lt;DIV&gt; #}
```

Please read the escaping guide for more information about escaping.

replace

`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"}) }}
```

Arguments

- placeholders to replace

reverse

`reverse`

The 'reverse' filter reverses a List:

```
{% for user in users | reverse %} {{ user }} {% endfor %}
```

rsort

`rsort`

The `rsort` filter will sort a list in reversed order. The items of the list must implement `Comparable`.

```
{% for user in users | sort %}
        {{ user.name }}
{% endfor %}
```

slice

`slice`

The `slice` filter returns a portion of a list, array, or string.

```
{{ ['apple', 'peach', 'pear', 'banana'] | slice(1,3) }}
{# results in: [peach, pear] #}


{{ 'Mitchell' | slice(1,3) }}
{# results in: 'it' #}
```

Arguments

- `fromIndex`: 0-based and inclusive
- `toIndex`: 0-based and exclusive

sort

`sort`

The `sort` filter will sort a list. The items of the list must implement `Comparable`.

```
{% for user in users | sort %}
        {{ user.name }}
{% endfor %}
```

title

`title`

The `title` filter will capitalize the first letter of each word.

```
{{ "article title" | title }}
```

The above example will output the following:

```
Article Title
```

See also: capitalize

trim

`trim`

The `trim` filter is used to trim whitespace off the beginning and end of a string.

```
{{ "    This text has too much whitespace.    " | trim }}
```

The above example will output the following:

```
This text has too much whitespace.
```

upper

`upper`

The `upper` filter makes an entire string upper case.

```
{{ "this is a quiet sentence." | upper }}
```

The above example will output the following:

```
THIS IS A QUIET SENTENCE.
```

urlencode

`urlencode`

The `urlencod` translates a string into `application/x-www-form-urlencoded` format using the "UTF-8" encoding scheme.

```
{{ "The string ü@foo-bar" | urlencode }}
```

The above example will output the following:

```
The+string+%C3%BC%40foo-bar
```

block

`block`

The `block` function is used to render the contents of a block more than once. It is not to be confused with the block *tag* which is used to declare blocks.

The following example will render the contents of the "post" block twice; once where it was declared and again using the `block` function:

```
{% block "post" %} content {% endblock %}

{{ block("post") }}
```

The above example will output the following:

```
content

content
```

Performance Warning

The `block` function will impair the use of the flush tag used within the block being rendered. It is typically okay for a block to use the `flush` tag which will flush the already-rendered content to the user-provided `Writer` but the block function will internally use it's own `StringWriter` and therefore flushing inside the block will no longer do any good (nor will it do harm).

i18n

`i18n`

The `i18n` function is used to retrieve messages from a locale-specific `ResourceBundle`. Every `PebbleTemplate` is assigned a default locale from the `PebbleEngine`. At the point of evaluation, this locale can be changed with an argument to the `evaluate(...)` method of the individual template.

The `i18n` function wraps around `ResourceBundle.getBundle(name, locale).getObject(key)`. The first argument to the `i18n` function is the name of the bundle and the second argument is the key within the bundle.

```
{{ i18n("messages","greeting") }}
```

The above example assumes you have `messages.properties` on your classpath and that that file contains a key by the name of `greeting`. If the locale of that template was `es_US` for example, it would look for a `message_es_US.properties` file instead.

Going a little further, you can use variables within your message and pass a list of params to this function which will replace your variables using `MessageFormat`:

```
{# greeting.someone=Hello, {0} #}
{{ i18n("messages","greeting", "Jacob") }}

{# output: Hello, Jacob #}
```

Arguments

- bundle
- key
- params

max

`max`

The `max` function will return the largest of it's numerical arguments.

```
{{ max(user.age, 80) }}
```

min

`min`

The `min` function will return the smallest of it's numerical arguments.

```
{{ min(user.age, 80) }}
```

parent

`parent`

The `parent` function is used inside of a block to render the content that the parent template would have rendered inside of the block had the current template not overriden it. It is similar to Java's `super` keyword.

Let's assume you have a template, "parent.peb" that looks something like this:

```
{% block "content" %}
        parent contents
{% endblock %}
```

And then you have another template, "child.peb" that extends "parent.peb":

```
{% extends "parent.peb" %}

{% block "content" %}
        child contents
        {{ parent() }}
{% endblock %}
```

The output will look something like the following:

```
parent contents
child contents
```

range

`range`

The `range` function will return a list containing an arithmetic progression of numbers:

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

{# outputs 0, 1, 2, 3, #}
```

When step is given (as the third parameter), it specifies the increment (or decrement):

```
{% for i in range(0, 6, 2) %}
    {{ i }},
{% endfor %}

{# outputs 0, 2, 4, 6, #}
```

Pebble built-in .. operator is just a shortcut for the range function with a step of 1+

```
{% for i in 0..3 %}
    {{ i }},
{% endfor %}

{# outputs 0, 1, 2, 3, #}
```

empty

`empty`

The `empty` test checks if a variable is empty. A variable is empty if it is null, an empty string, an empty collection, or an empty map.

```
{% if user.email is empty %}
        ...
{% endif %}
```

even

even

The `even` test checks if an integer is even.

```
{% if 2 is even %}
        ...
{% endif %}
```

map

`map`

The `map` test checks if a variable is an instance of a map.

```
{% if {"apple":"red", "banana":"yellow"} is map %}
        ...
{% endif %}
```

null

`null`

The `null` test checks if a variable is null.

```
{% if user.email is null %}
        ...
{% endif %}
```

odd

odd

The `odd` test checks if an integer is odd.

```
{% if 3 is odd %}
        ...
{% endif %}
```

iterable

`iterable`

The `iterable` test checks if a variable implements `java.lang.Iterable`.

```
{% if users is iterable %}
        {% for user in users %}
                ...
        {% endfor %}
{% endif %}
```

comparisons

Comparisons

Pebble provides the following comparison operators: `==`, `!=`, `<`, `>`, `<=`, `>=`. All of them except for `==` are equivalent to their Java counterparts. The `==` operator uses `ava.util.Objects.equals(a, b)` behind the scenes to perform null safe value comparisons.

`equals` is an alias for `==`

```
{% if user.name equals "Mitchell" %}
        ...
{% endif %}
```

contains

`contains`

The `contains` operator can be used to determine if a collection, map, or array contains a particular item.

```
{% if ["apple", "pear", "banana"] contains "apple" %}
        ...
{% endif %}
```

When using maps, the contains operator checks for an existing key.

```
{% if {"apple":"red", "banana":"yellow"} contains "banana" %}
        ...
{% endif %}
```

The operator can be used to look for multiple items at once:

```
{% if ["apple", "pear", "banana", "peach"] contains ["apple", "peach"] %}
        ...
{% endif %}
```

is

`is`

The `is` operator will apply a test to a variable which will return a boolean.

```
{% if 2 is even %}
        ...
{% endif %}
```

The result can be negated using the not operator.

logic

Logic

The `and` operator and the `or` operator are available to join boolean expressions.

```
{% if 2 is even and 3 is odd %}
        ...
{% endif %}
```

The `not` operator is available to negate a boolean expression.

```
{% if 3 is not even %}
        ...
{% endif %}
```

Parenthesis can be used to group expressions to ensure a desired precedence.

```
{% if (3 is not even) and (2 is odd or 3 is even) %}
        ...
{% endif %}
```

math

Math

All the regular math operators are available for use. Order of operations applies.

```
{{ 2 + 2 / ( 10 % 3 ) * (8 - 1) }}
```

The result can be negated using the not operator.

others

Other Operators

The `|` operator is used to apply a filter to a variable.

```
{{ user.name | capitalize }}
```

Pebble supports the use of the conditional operator (often named the ternary operator).

```
{{ foo == null ? bar : baz }}
```