Cloud Mailing Documentation

Release 0.1

Cedric RICARD

May 06, 2016

Contents

Cloud Mailing is an e-mailing engine designed for simplicity and performance thanks to its cloud (= *distributed*) architecture.

Look how easy it is to use:

```
import xmlrpclib

config = {
    'ip': '192.168.1.150',
    'api_key': "xXXxxxXxxX",
}

cm_master = xmlrpclib.ServerProxy("https://admin:%(api_key)s@%(ip)s:33610/CloudMailing" % config)

mailing_id = cm_master.create_mailing(
    "my-mailing@example.org",          # Sender email
    "My Mailing",                      # Sender name
    "The great newsletter",            # Subject
    "<h1>Title</h1><p>Coucou</p>",     # HTML content
    "Title\nCoucou\n",                 # Plain text content
    "UTF-8"                            # Text encoding (for both HTML and plain text content)
)

cm_master.set_mailing_properties(mailing_id, {
    'scheduled_start': datetime.now() + timedelta(hours=3),
    'scheduled_duration': 1440,  # in minutes
    'click_tracking': True,
})

cm_master.add_recipients(mailing_id, [
    {'email': 'john.doe@example.org', 'firstname': 'John', 'lastname': 'DOE', 'another_custom_field'
    {'email': 'wilfred.smith@example.org'},
    [...]
])

cm_master.start_mailing(mailing_id)
```

CHAPTER 1

Table of content

1.1 User guide

The goal of this guide is to help you to use Cloud Mailing API to send e-mailings.

1.1.1 Get started with Cloud Mailing API

Overview

The CloudMailing XML-RPC server allows to directly manage MailFountain CloudMailing Engine.

Authentication

You should be authenticated to be able to use it. Authentication is done by simple HTTP authentication method. A special API key should be used as password (login is fixed to 'admin'). This key has to be generated from Web administration pages.

Example of authentication:

```
import xmlrpclib

config = {
    'ip': '192.168.1.150',
    'api_key': "xXXxxxXxxX",
}
cm_master = xmlrpclib.ServerProxy("https://admin:%(api_key)s@%(ip)s:33610/CloudMailing" % config)
```

Create a mailing

To create a mailing, simply call the *create_mailing()* RPC function:

```
mailing_id = cm_master.create_mailing(
    "my-mailing@%example.org",          # Sender email
    "My Mailing",                       # Sender name
    "The great newsletter",             # Subject
    "<h1>Title</h1><p>Coucou</p>",      # HTML content
    "Title\nCoucou\n",                  # Plain text content
    "UTF-8"                             # Text encoding (for both HTML and plain text content)
)
```

This function returns you the mailing ID. This ID is unique and allows you to manage the mailing.

Then you should want to change/add more properties to your mailing. The function *set_mailing_properties()* is made for this:

```
cm_master.set_mailing_properties(mailing_id, {
    'scheduled_start': datetime.now() + timedelta(hours=3),
    'scheduled_duration': 1440,  # in minutes
    'click_tracking': True,
})
```

Add some recipients

To add recipients into your mailing, you should use the *add_recipients()* function:

```
cm_master.add_recipients(mailing_id, [
    {'email': 'john.doe@example.org', 'firstname': 'John', 'lastname': 'DOE', 'another_custom_field'
    {'email': 'wilfred.smith@example.org'},
    [...]
])
```

The function will return you an array with exactly the same number of entries, in the same order as input. Each entry is also a dictionary containing 'email' field (the same as input), an 'id' which is unique for each recipient and, only in case an error occurs, a field 'error' containing the failure reason in plain text.

'id' and 'error' are mutually exclusive. In case of success, only 'id' is present; in case of failure, only 'error' can be found.

You can of course call this function as many time you want. There is no limit to the quantity of recipients a mailing can handle. **But** be careful **to not send too many recipients at once** (i.e. in one single call) because depending of the amount of customization data per recipient, you may reach the buffer limit of either the XMLRPC client or server.

Start a mailing

The start of a mailing is very simple:

```
cm_master.start_mailing(mailing_id)
```

After this call, the mailing will be immediately eligible for adding its recipients to a sending queue, on condition you didn't define a scheduled_start date in the future of course.

Retrieve recipients reports

Once the mailing started, you should want to retrieve sending status for each recipient. As it can have a hugh amount of recipients, it won't be efficient to request continuously for each ones.

So Cloud Mailing API provides a more sophisticated function which will only return recipients for which the sending status has changed since the last call. And for security, the maximum amount of results is limited.

```
mailing_is_running = True
cursor = ''
status_filter = []  # No filter = all status except 'READY' (no sens to get them)
while mailing_is_running:
    result = cm_master.get_recipients_status_updated_since(cursor, status_filter, 1000)
    cursor = result['cursor']
    recipients_status = result['recipients']
```

```
    # recipients_status is an array containing dictionaries
    [...]
```

To make it possible, the function returns you a private cursor object that you have to send it back on next call.

Retrieve mailing report

With mailing report, you will able to know how many recipients have been handled with success, are in error, will be tried again and left to be handled. This will allow you to know (and its probably the most important) if your mailing is finished or not (throw the mailing status):

```
filter = {'id': [mailing_id]}
mailing = cm_master.list_mailings(filter)[0]
while mailing['status'] != 'FINISHED':
    print("Total recipients: %d", mailing['total_recipient'])
    print("Recipients finished: %d", mailing['total_sent'])
    print("Recipients in error: %d", mailing['total_error'])
    [...]
    mailing = cm_master.list_mailings(filter)[0]
```

1.1.2 Create a mailing

Mailing type

Cloud Mailing is capable to mange two types of mailing:

- **Regular mailing**: simple mailing which will end as soon as its recipients list is empty, or its end date is reached, whatever occurs in first.
- **Opened mailing**: for this type of mailing, only the end date (if set) will close the mailing.

In fact, the real (and only) difference between 2 types is the mailing automatic ending.

Regular mailing

The *regular* type is for mailings for which the number of recipients is known (or nearly) from start. This is the *default* type.

The simplest workflow is:

- create mailing
- add recipients
- start mailing
- wait for the automatic end when all recipients have been handled

While this is very simple, it can be too long before the first recipient is addressed when the number of recipients is high. It is possible to start the mailing before adding all recipients, then adding them after, but there is a big risk that the recipients queue become empty before the end (because you didn't added recipients fast enough), and the mailing closes itself too early.

That why it is possible to set a mailing property (*dont_close_if_empty*) that explicitly tells Cloud Mailing to not close this mailing because we are still filling its recipients queue. So the workflow become:

- create mailing

- set *dont_close_if_empty* property to *True*
- start mailing
- add recipients
- set *dont_close_if_empty* property to *False*
- wait for the automatic end when all recipients have been handled

Opened mailing

The *opened* type is for *permanent* mailings. They are mailing that are always active, and recipients can be added at any time and handled immediately.

An example of opened mailing usage can be the confirmation email of an online store when a customer completes a purchase. The email is always the same, only content (purchased items) change and is handled by advanced customization.

1.1.3 Template Designer Documentation

This document describes the syntax and semantics of the template engine (called 'Jinja') and will be most useful as reference to those creating Mailing templates. As the template engine is very flexible the configuration from the application might be slightly different from here in terms of delimiters and behavior of undefined values.

Synopsis

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

A template contains **variables** or **expressions**, which get replaced with values when the template is evaluated, and tags, which control the logic of the template. The template syntax is heavily inspired by Django and Python.

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

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
    <title>My Webpage</title>
</head>
<body>
    <ul id="navigation">
    {% for item in navigation %}
        <li><a href="{{ item.href }}">{{ item.caption }}</a></li>
    {% endfor %}
    </ul>

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

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

Variables

You can mess around with the variables in templates provided they are passed in by the application. Variables may have attributes or elements on them you can access too. What attributes a variable has depends heavily on the application providing that variable.

You can use a dot (`.`) to access attributes of a variable, but alternatively the so-called "subscript" syntax (`[]`) can be used. The following lines do the same thing:

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

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

If a variable or attribute does not exist you will get back an undefined value. What you can do with that kind of value depends on the application configuration: the default behavior is that it evaluates to an empty string if printed and that you can iterate over it, but every other operation fails.

Implementation

For convenience sake `foo.bar` in Jinja2 does the following things on the Python layer:

- check if there is an attribute called *bar* on *foo*.
- if there is not, check if there is an item `'bar'` in *foo*.
- if there is not, return an undefined object.

`foo['bar']` on the other hand works mostly the same with the a small difference in the order:

- check if there is an item `'bar'` in *foo*.
- if there is not, check if there is an attribute called *bar* on *foo*.
- if there is not, return an undefined object.

This is important if an object has an item or attribute with the same name. Additionally there is the `attr()` filter that just looks up attributes.

Filters

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

`{{ name|striptags|title }}` for example will remove all HTML Tags from the *name* and title-cases it. Filters that accept arguments have parentheses around the arguments, like a function call. This example will join a list by commas: `{{ list|join(', ') }}`.

The *List of Builtin Filters* below describes all the builtin filters.

Tests

Beside filters there are also so called "tests" available. Tests can be used to test a variable against a common expression. To test a variable or expression you add *is* plus the name of the test after the variable. For example to find out if a variable is defined you can do `name is defined` which will then return true or false depending on if *name* is defined.

Tests can accept arguments too. If the test only takes one argument you can leave out the parentheses to group them. For example the following two expressions do the same:

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

The *List of Builtin Tests* below describes all the builtin tests.

Comments

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

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

Whitespace Control

In the default configuration, a single trailing newline is stripped if present, and whitespace is not further modified by the template engine. Each whitespace (spaces, tabs, newlines etc.) is returned unchanged. If the application configures Jinja to *trim_blocks* the first newline after a template tag is removed automatically (like in PHP). The *lstrip_blocks* option can also be set to strip tabs and spaces from the beginning of line to the start of a block. (Nothing will be stripped if there are other characters before the start of the block.)

With both *trim_blocks* and *lstrip_blocks* enabled you can put block tags on their own lines, and the entire block line will be removed when rendered, preserving the whitespace of the contents. For example, without the *trim_blocks* and *lstrip_blocks* options, this template:

```
<div>
    {% if True %}
        yay
    {% endif %}
</div>
```

gets rendered with blank lines inside the div:

```
<div>

        yay

</div>
```

But with both *trim_blocks* and *lstrip_blocks* enabled, the lines with the template blocks are removed while preserving the whitespace of the contents:

```
<div>
        yay
</div>
```

You can manually disable the *lstrip_blocks* behavior by putting a plus sign (+) at the start of a block:

```
<div>
        {%+ if something %}yay{% endif %}
</div>
```

You can also strip whitespace in templates by hand. If you put an minus sign (-) to the start or end of an block (for example a for tag), a comment or variable expression you can remove the whitespaces after or before that block:

```
{% for item in seq -%}
    {{ item }}
{%- endfor %}
```

This will yield all elements without whitespace between them. If *seq* was a list of numbers from `1` to `9` the output would be `123456789`.

If *Line Statements* are enabled they strip leading whitespace automatically up to the beginning of the line.

Jinja2 by default also removes trailing newlines. To keep the single trailing newline when it is present, configure Jinja to *keep_trailing_newline*.

Note

You must not use a whitespace between the tag and the minus sign.

valid:

```
{%- if foo -%}...{% endif %}
```

invalid:

```
{% - if foo - %}...{% endif %}
```

Escaping

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

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

```
{{ '{{' }}
```

For bigger sections it makes sense to mark a block *raw*. For example to put Jinja syntax as example into a template you can use this snippet:

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

Line Statements

If line statements are enabled by the application it's possible to mark a line as a statement. For example if the line statement prefix is configured to `#` the following two examples are equivalent:

```
<ul>
# for item in seq
    <li>{{ item }}</li>
# endfor
```

```
</ul>

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

The line statement prefix can appear anywhere on the line as long as no text precedes it. For better readability statements that start a block (such as *for*, *if*, *elif* etc.) may end with a colon:

```
# for item in seq:
    ...
# endfor
```

Note

Line statements can span multiple lines if there are open parentheses, braces or brackets:

```
<ul>
# for href, caption in [('index.html', 'Index'),
                        ('about.html', 'About')]:
    <li><a href="{{ href }}">{{ caption }}</a></li>
# endfor
</ul>
```

Since Jinja 2.2 line-based comments are available as well. For example if the line-comment prefix is configured to be `##` everything from `##` to the end of the line is ignored (excluding the newline sign):

```
# for item in seq:
    <li>{{ item }}</li>     ## this comment is ignored
# endfor
```

HTML Escaping

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

Jinja supports both, but what is used depends on the application configuration. The default configuration is no automatic escaping for various reasons:

- escaping everything except of safe values will also mean that Jinja is escaping variables known to not include HTML such as numbers which is a huge performance hit.
- The information about the safety of a variable is very fragile. It could happen that by coercing safe and unsafe values the return value is double escaped HTML.

Working with Manual Escaping

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

Working with Automatic Escaping

When automatic escaping is enabled everything is escaped by default except for values explicitly marked as safe. Those can either be marked by the application or in the template by using the *|safe* filter. The main problem with this approach is that Python itself doesn't have the concept of tainted values so the information if a value is safe or unsafe can get lost. If the information is lost escaping will take place which means that you could end up with double escaped contents.

Double escaping is easy to avoid however, just rely on the tools Jinja2 provides and don't use builtin Python constructs such as the string modulo operator.

Functions returning template data (macros, *super*, *self.BLOCKNAME*) return safe markup always.

String literals in templates with automatic escaping are considered unsafe too. The reason for this is that the safe string is an extension to Python and not every library will work properly with it.

List of Control Structures

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

For

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

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

As variables in templates retain their object properties, it is possible to iterate over containers like *dict*:

```
<dl>
{% for key, value in my_dict.iteritems() %}
    <dt>{{ key|e }}</dt>
    <dd>{{ value|e }}</dd>
{% endfor %}
</dl>
```

Note however that dictionaries usually are unordered so you might want to either pass it as a sorted list to the template or use the *dictsort* filter.

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

Variable	Description
loop.index	The current iteration of the loop. (1 indexed)
loop.index0	The current iteration of the loop. (0 indexed)
loop.revindex	The number of iterations from the end of the loop (1 indexed)
loop.revindex0	The number of iterations from the end of the loop (0 indexed)
loop.first	True if first iteration.
loop.last	True if last iteration.
loop.length	The number of items in the sequence.
loop.cycle	A helper function to cycle between a list of sequences. See the explanation below.
loop.depth	Indicates how deep in deep in a recursive loop the rendering currently is. Starts at level 1
loop.depth0	Indicates how deep in deep in a recursive loop the rendering currently is. Starts at level 0

Within a for-loop, it's possible to cycle among a list of strings/variables each time through the loop by using the special *loop.cycle* helper:

```
{% for row in rows %}
    <li class="{{ loop.cycle('odd', 'even') }}">{{ row }}</li>
{% endfor %}
```

Since Jinja 2.1 an extra *cycle* helper exists that allows loop-unbound cycling. For more information have a look at the *List of Global Functions*. Unlike in Python it's not possible to *break* or *continue* in a loop. You can however filter the sequence during iteration which allows you to skip items. The following example skips all the users which are hidden:

```
{% for user in users if not user.hidden %}
    <li>{{ user.username|e }}</li>
{% endfor %}
```

The advantage is that the special *loop* variable will count correctly thus not counting the users not iterated over.

If no iteration took place because the sequence was empty or the filtering removed all the items from the sequence you can render a replacement block by using *else*:

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

Note that in Python *else* blocks are executed whenever the corresponding loop did not *break*. Since in Jinja loops cannot *break* anyway, a slightly different behavior of the *else* keyword was chosen.

It is also possible to use loops recursively. This is useful if you are dealing with recursive data such as sitemaps. To use loops recursively you basically have to add the *recursive* modifier to the loop definition and call the *loop* variable with the new iterable where you want to recurse.

The following example implements a sitemap with recursive loops:

```
<ul class="sitemap">
{%- for item in sitemap recursive %}
    <li><a href="{{ item.href|e }}">{{ item.title }}</a>
    {%- if item.children -%}
        <ul class="submenu">{{ loop(item.children) }}</ul>
    {%- endif %}</li>
{%- endfor %}
</ul>
```

The *loop* variable always refers to the closest (innermost) loop. If we have more than one levels of loops, we can rebind the variable *loop* by writing *{% set outer_loop = loop %}* after the loop that we want to use recursively. Then, we can call it using *{{ outer_loop(...) }}*

If

The *if* statement in Jinja is comparable with the if statements of Python. In the simplest form you can use it to test if a variable is defined, not empty or not false:

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

```
{% endfor %}
</ul>
{% endif %}
```

For multiple branches *elif* and *else* can be used like in Python. You can use more complex *Expressions* there too:

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

If can also be used as *inline expression* and for *loop filtering*.

Macros

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

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

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

The macro can then be called like a function in the namespace:

```
<p>{{ input('username') }}</p>
<p>{{ input('password', type='password') }}</p>
```

If the macro was defined in a different template you have to import it first.

Inside macros you have access to three special variables:

varargs If more positional arguments are passed to the macro than accepted by the macro they end up in the special *varargs* variable as list of values.

kwargs Like *varargs* but for keyword arguments. All unconsumed keyword arguments are stored in this special variable.

caller If the macro was called from a *call* tag the caller is stored in this variable as macro which can be called.

Macros also expose some of their internal details. The following attributes are available on a macro object:

name The name of the macro. `{{ input.name }}` will print `input`.

arguments A tuple of the names of arguments the macro accepts.

defaults A tuple of default values.

catch_kwargs This is *true* if the macro accepts extra keyword arguments (ie: accesses the special *kwargs* variable).

catch_varargs This is *true* if the macro accepts extra positional arguments (ie: accesses the special *varargs* variable).

caller This is *true* if the macro accesses the special *caller* variable and may be called from a *call* tag.

If a macro name starts with an underscore it's not exported and can't be imported.

Call

In some cases it can be useful to pass a macro to another macro. For this purpose you can use the special *call* block. The following example shows a macro that takes advantage of the call functionality and how it can be used:

```
{% macro render_dialog(title, class='dialog') -%}
    <div class="{{ class }}">
        <h2>{{ title }}</h2>
        <div class="contents">
            {{ caller() }}
        </div>
    </div>
{%- endmacro %}

{% call render_dialog('Hello World') %}
    This is a simple dialog rendered by using a macro and
    a call block.
{% endcall %}
```

It's also possible to pass arguments back to the call block. This makes it useful as replacement for loops. Generally speaking a call block works exactly like an macro, just that it doesn't have a name.

Here an example of how a call block can be used with arguments:

```
{% macro dump_users(users) -%}
    <ul>
    {%- for user in users %}
        <li><p>{{ user.username|e }}</p>{{ caller(user) }}</li>
    {%- endfor %}
    </ul>
{%- endmacro %}

{% call(user) dump_users(list_of_user) %}
    <dl>
        <dl>Realname</dl>
        <dd>{{ user.realname|e }}</dd>
        <dl>Description</dl>
        <dd>{{ user.description }}</dd>
    </dl>
{% endcall %}
```

Filters

Filter sections allow you to apply regular Jinja2 filters on a block of template data. Just wrap the code in the special *filter* section:

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

Expressions

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

Literals

The simplest form of expressions are literals. Literals are representations for Python objects such as strings and numbers. The following literals exist:

"Hello World": Everything between two double or single quotes is a string. They are useful whenever you need a string in the template (for example as arguments to function calls, filters or just to extend or include a template).

42 / 42.23: Integers and floating point numbers are created by just writing the number down. If a dot is present the number is a float, otherwise an integer. Keep in mind that for Python `42` and `42.0` is something different.

['list', 'of', 'objects']: Everything between two brackets is a list. Lists are useful to store sequential data in or to iterate over them. For example you can easily create a list of links using lists and tuples with a for loop:

```
<ul>
{% for href, caption in [('index.html', 'Index'), ('about.html', 'About'),
                         ('downloads.html', 'Downloads')] %}
    <li><a href="{{ href }}">{{ caption }}</a></li>
{% endfor %}
</ul>
```

('tuple', 'of', 'values'): Tuples are like lists, just that you can't modify them. If the tuple only has one item you have to end it with a comma. Tuples are usually used to represent items of two or more elements. See the example above for more details.

{'dict': 'of', 'key': 'and', 'value': 'pairs'}: A dict in Python is a structure that combines keys and values. Keys must be unique and always have exactly one value. Dicts are rarely used in templates, they are useful in some rare cases such as the `xmlattr()` filter.

true / false: true is always true and false is always false.

Note

The special constants *true*, *false* and *none* are indeed lowercase. Because that caused confusion in the past, when writing *True* expands to an undefined variable that is considered false, all three of them can be written in title case too (*True*, *False*, and *None*). However for consistency (all Jinja identifiers are lowercase) you should use the lowercase versions.

Math

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

+ Adds two objects together. Usually the objects are numbers but if both are strings or lists you can concatenate them this way. This however is not the preferred way to concatenate strings! For string concatenation have a look at the ~ operator. `{{ 1 + 1 }}` is `2`.

- Subtract the second number from the first one. `{{ 3 - 2 }}` is `1`.

/ Divide two numbers. The return value will be a floating point number. `{{ 1 / 2 }}` is `{{ 0.5 }}`.

// Divide two numbers and return the truncated integer result. `{{ 20 // 7 }}` is `2`.

% Calculate the remainder of an integer division. `{{ 11 % 7 }}` is `4`.

***** Multiply the left operand with the right one. `{{ 2 * 2 }}` would return `4`. This can also be used to repeat a string multiple times. `{{ '=' * 80 }}` would print a bar of 80 equal signs.

****** Raise the left operand to the power of the right operand. `{{ 2**3 }}` would return `8`.

Comparisons

== Compares two objects for equality.

!= Compares two objects for inequality.

> *true* if the left hand side is greater than the right hand side.

>= *true* if the left hand side is greater or equal to the right hand side.

< *true* if the left hand side is lower than the right hand side.

<= *true* if the left hand side is lower or equal to the right hand side.

Logic

For *if* statements, *for* filtering or *if* expressions it can be useful to combine multiple expressions:

and Return true if the left and the right operand is true.

or Return true if the left or the right operand is true.

not negate a statement (see below).

(expr) group an expression.

Note

The `is` and `in` operators support negation using an infix notation too: `foo is not bar` and `foo not in bar` instead of `not foo is bar` and `not foo in bar`. All other expressions require a prefix notation: `not (foo and bar)`.

Other Operators

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

in Perform sequence / mapping containment test. Returns true if the left operand is contained in the right. `{{ 1 in [1, 2, 3] }}` would for example return true.

is Performs a *test*.

| Applies a *filter*.

~ Converts all operands into strings and concatenates them. `{{ "Hello " ~ name ~ "!" }}` would return (assuming *name* is `'John'`) `Hello John!`.

() Call a callable: `{{ post.render() }}`. Inside of the parentheses you can use positional arguments and keyword arguments like in python: `{{ post.render(user, full=true) }}`.

. / [] Get an attribute of an object. (See *Variables*)

If Expression

It is also possible to use inline *if* expressions. These are useful in some situations. For example you can use this to extend from one template if a variable is defined, otherwise from the default layout template:

```
{% extends layout_template if layout_template is defined else 'master.html' %}
```

The general syntax is `<do something> if <something is true> else <do something else>`.

The *else* part is optional. If not provided the else block implicitly evaluates into an undefined object:

```
{{ '[%s]' % page.title if page.title }}
```

List of Builtin Filters

List of Builtin Tests

List of Global Functions

The following functions are available in the global scope by default:

range(*[start]*, *stop[, step]*)

Return a list containing an arithmetic progression of integers. range(i, j) returns [i, i+1, i+2, ..., j-1]; start (!) defaults to 0. When step is given, it specifies the increment (or decrement). For example, range(4) returns [0, 1, 2, 3]. The end point is omitted! These are exactly the valid indices for a list of 4 elements.

This is useful to repeat a template block multiple times for example to fill a list. Imagine you have 7 users in the list but you want to render three empty items to enforce a height with CSS:

```
<ul>
{% for user in users %}
    <li>{{ user.username }}</li>
{% endfor %}
{% for number in range(10 - users|count) %}
    <li class="empty"><span>...</span></li>
{% endfor %}
</ul>
```

lipsum(*n=5*, *html=True*, *min=20*, *max=100*)

Generates some lorem ipsum for the template. Per default five paragraphs with HTML are generated each paragraph between 20 and 100 words. If html is disabled regular text is returned. This is useful to generate simple contents for layout testing.

dict(***items*)

A convenient alternative to dict literals. `{'foo': 'bar'}` is the same as `dict(foo='bar')`.

class cycler(**items*)

The cycler allows you to cycle among values similar to how *loop.cycle* works. Unlike *loop.cycle* however you can use this cycler outside of loops or over multiple loops.

This is for example very useful if you want to show a list of folders and files, with the folders on top, but both in the same list with alternating row colors.

The following example shows how *cycler* can be used:

```
{% set row_class = cycler('odd', 'even') %}
<ul class="browser">
{% for folder in folders %}
  <li class="folder {{ row_class.next() }}">{{ folder|e }}</li>
{% endfor %}
{% for filename in files %}
  <li class="file {{ row_class.next() }}">{{ filename|e }}</li>
{% endfor %}
</ul>
```

A cycler has the following attributes and methods:

reset()
Resets the cycle to the first item.

next()
Goes one item a head and returns the then current item.

current
Returns the current item.

new in Jinja 2.1

class joiner(*sep=', '*)
A tiny helper that can be use to "join" multiple sections. A joiner is passed a string and will return that string every time it's called, except the first time in which situation it returns an empty string. You can use this to join things:

```
{% set pipe = joiner("|") %}
{% if categories %} {{ pipe() }}
    Categories: {{ categories|join(", ") }}
{% endif %}
{% if author %} {{ pipe() }}
    Author: {{ author() }}
{% endif %}
{% if can_edit %} {{ pipe() }}
    <a href="?action=edit">Edit</a>
{% endif %}
```

new in Jinja 2.1

Extensions

The following sections cover the built-in Jinja2 extensions that may be enabled by the application. The application could also provide further extensions not covered by this documentation. In that case there should be a separate document explaining the extensions.

Expression Statement

If the expression-statement extension is loaded a tag called *do* is available that works exactly like the regular variable expression ({{ ... }}) just that it doesn't print anything. This can be used to modify lists:

```
{% do navigation.append('a string') %}
```

Loop Controls

If the application enables the loopcontrols-extension it's possible to use *break* and *continue* in loops. When *break* is reached, the loop is terminated; if *continue* is reached, the processing is stopped and continues with the next iteration.

Here a loop that skips every second item:

```
{% for user in users %}
    {%- if loop.index is even %}{% continue %}{% endif %}
    ...
{% endfor %}
```

Likewise a look that stops processing after the 10th iteration:

```
{% for user in users %}
    {%- if loop.index >= 10 %}{% break %}{% endif %}
{%- endfor %}
```

With Statement

New in version 2.3.

If the application enables the with-extension it is possible to use the *with* keyword in templates. This makes it possible to create a new inner scope. Variables set within this scope are not visible outside of the scope.

With in a nutshell:

```
{% with %}
    {% set foo = 42 %}
    {{ foo }}           foo is 42 here
{% endwith %}
foo is not visible here any longer
```

Because it is common to set variables at the beginning of the scope you can do that within the with statement. The following two examples are equivalent:

```
{% with foo = 42 %}
    {{ foo }}
{% endwith %}

{% with %}
    {% set foo = 42 %}
    {{ foo }}
{% endwith %}
```

Autoescape Extension

New in version 2.4.

If the application enables the autoescape-extension one can activate and deactivate the autoescaping from within the templates.

Example:

```
{% autoescape true %}
    Autoescaping is active within this block
{% endautoescape %}

{% autoescape false %}
    Autoescaping is inactive within this block
{% endautoescape %}
```

After the *endautoescape* the behavior is reverted to what it was before.

1.1.4 API reference

1.2 Developer guide

1.2.1 Using MongoDB

Install MongoDB

brew install mongodb

Start MongoDB

To have launchd start mongodb at login:

ln -sfv /usr/local/opt/mongodb/*.plist ~/Library/LaunchAgents

Then to load mongodb now:

launchctl load ~/Library/LaunchAgents/homebrew.mxcl.mongodb.plist

Or, if you don't want/need launchctl, you can just run:

mongod –config /usr/local/etc/mongod.conf

Settings

Settings from config file

The config file is located in *config/cloud-mailing.ini* and is in INI format. These settings are common to master and satellite.

Master Settings from DB

CHAPTER 2

Features

- Simple to use
- Scalable

CHAPTER 3

Installation

TODO

CHAPTER 4

Contribute

- Issue Tracker: github.com/ricard33/cloud-mailing/issues
- Source Code: github.com/ricard33/cloud-mailing

CHAPTER 5

Support

CHAPTER 6

License

The project is licensed under the GNU Affero General Public License v3.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

C

D

J

L

N

R