Bottle Utils Documentation

Release 2.0.dev1

Outernet Inc <hello@outernet.is>

September 08, 2016

Contents

Bottle Utils (package name `bottle-utils`) is a collection of decorators, functions and classes that address typical problems developing web sites and applications using Bottle framework. This package is created based on code we use at Outernet for various user-facing interfaces as well as our own sites.

Bottle Utils are compatible with Python 2.7, 3.3, and 3.4. Compatibility with other versions of Python is possible, but not tested. It targets latest stable release of Bottle.

CHAPTER 1

Installation

Use one of the following commands to install Bottle Utils:

```
pip install bottle-utils

easy_install bottle-utils
```

Note: Between versions 0.3 and 0.5, bottle-utils package was split into multiple packages. Packages were structured in a way that allowed the API prior to version 0.3 to work without issues. However, this has caused various problems with deployment and development, and the approach was subsequently abandoned. Starting with version 0.5, Bottle Utils is again a monolithic package.

CHAPTER 2

Source code

The complete source code is licensed under BSD license (see `LICENSE` file in the source package), and available on GitHub.

CHAPTER 3

Package contents

The following functionality is available:

3.1 Common utilities (`bottle_utils.common`)

The `bottle_utils.common` module contains functions and constants that are used in other modules.

This module also contains a few constants and variables that improve code that targets both Python 2.x and Python 3.x. Note, though, that this mostly works in the context of Bottle Utils and hasn't been tested in too many different versions of Python. If you want a more comprehensive solution, you should look at six.

This module also contains names `unicode` and `basestring`, which work as expected in both Python 2.x and Python 3.x.

3.1.1 Module contents

`bottle_utils.common.`**`PY3 = False`**
Whether Python version is 3.x

`bottle_utils.common.`**`PY2 = True`**
Whether Python version is 2.x

`bottle_utils.common.`**`to_unicode`**(*v*, *encoding=u'utf8'*)
Convert a value to Unicode string (or just string in Py3). This function can be used to ensure string is a unicode string. This may be useful when input can be of different types (but meant to be used when input can be either bytestring or Unicode string), and desired output is always Unicode string.

The `encoding` argument is used to specify the encoding for bytestrings.

`bottle_utils.common.`**`to_bytes`**(*v*, *encoding=u'utf8'*)
Convert a value to bytestring (or just string in Py2). This function is useful when desired output is always a bytestring, and input can be any type (although it is intended to be used with strings and bytestrings).

The `encoding` argument is used to specify the encoding of the resulting bytestring.

`bottle_utils.common.`**`attr_escape`**(*attr*)
Escape `attr` string containing HTML attribute value. This function escapes certain characters that are undesirable in HTML attribute values. Functions that construct attribute values using user-supplied data should escape the values using this function.

`bottle_utils.common.`**`html_escape`**(*html*)
Escape `html` strning containing HTML. This function escapes characters that are not desirable in HTML

markup, when the source string should represent text content only. User-supplied data that should appar in markup should be escaped using this function.

bottle_utils.common.**full_url**(*path=u'/'*)
Convert a specified path to full URL based on request data. This function uses the current request context information about the request URL to construct a full URL using specified path. In particular it uses `bottle.request.urlparts` to obtain information about scheme, hostname, and port (if any).

Because it uses the request context, it cannot be called outside a request.

bottle_utils.common.**urlquote**(*s*)
Quote (URL-encode) a string with Unicode support. This is a simple wrapper for `urllib.quote` (or `urllib.parse.quote`) that converts the input to UTF-8-encoded bytestring before quoting.

3.2 AJAX (bottle_utils.ajax)

The `bottle_utils.ajax` module provides decorators for working with AJAX requests.

3.2.1 Decorators

bottle_utils.ajax.**ajax_only**(*func*)
Return HTTP 400 response for all non-XHR requests.

> **Warning:** AJAX header ('X-Requested-With') can be faked, so don't use this decorator as a security measure of any kind.

Example:

```
@ajax_only
def hidden_from_non_xhr():
    return "Foo!"
```

bottle_utils.ajax.**roca_view**(*full*, *partial*, ***defaults*)
Render `partal` for XHR requests and `full` template otherwise. If `template_func` keyword argument is found, it is assumed to be a function that renders the template, and is used instead of the default one, which is `bottle.template()`.

Note: To work around issues with Chrome browser (all platforms) when using this decorator in conjunction with HTML5 pushState, the decorator always adds a `Cache-Control: no-store` header to partial responses.

Example:

```
@roca_view('page.html', 'fragment.html')
def my_roca_handler():
    return dict()
```

3.3 CSRF protection (bottle_utils.csrf)

This module contains decorators and functions for facilitating CSRF protection.

3.3.1 App configuration

Functions in this module require the Bottle application to be configured with CSRF-specific options.

Here is an example of file-based configuration:

```
[csrf]
secret = SomeSecretValue
token_name = _csrf_token
path = /
expires = 600
```

`secret` setting is the only setting you really must override. Not having this setting set will result in `KeyError` exception.

When using dict-based configuration, prefix each key with `csrf.`.

The keys have following meaning:

- `csrf.secret` setting is a secret key used for setting cookies; it should be fairly random and difficult to guess
- `csrf.token_name` setting is the name of the cookie and form field that contain the token
- `csrf.path` setting is the path of the cookie
- `csrf.expires` setting is in seconds and sets the cookie's max-age

3.3.2 Caveat

As with most common CSRF protection schemes, decorators in this module will prevent the user from opening two forms and submitting them one after the other. This also applies to cases where forms are fetched from server side using XHR.

Every form must have a token, and a token must match the one in the cookie. However, there is only one cookie for the whole site. When you submit a form, the token in the cookie is replaced with a new one, making tokens in any of the previously opened forms invalid. The result is that form submission results in a HTTP 403 response (not authorized).

You need to decide whether you can live with this behavior before using this module. In case of XHR, consolidating different forms into a single form or fetching tokens separately may be viable solutions.

Note: A possible workaround for applications loading forms (and tokens) using XHR would be to reload all tokens on the page whenever one of the forms is submitted.

3.3.3 Functions and decorators

`bottle_utils.csrf.`**`csrf_protect`**(*func*)

Perform CSRF protection checks. Performs checks to determine if submitted form data matches the token in the cookie. It is assumed that the GET request handler successfully set the token for the request and that the form was instrumented with a CSRF token field. Use the *`csrf_token()`* decorator to do this.

If the handler function returns (i.e., it is not interrupted with `bottle.abort()`, `bottle.redirect()`, and similar functions that throw an exception, a new token is set and response is returned to the requester. It is therefore recommended to perform a reidrect on successful POST.

Generally, the handler does not need to do anything CSRF-protection-specific. All it needs is the decorator:

```
    @app.post('/')
    @bottle.view('myform')
    @csrf_protect
    def protected_post_handler():
        if successful:
            redirect('/someplace')
        return dict(errors="There were some errors")
```

bottle_utils.csrf.**csrf_tag**()

Generte HTML for hidden form field. This is a convenience function to generate a simple hidden input field. It does not accept any arguments since it uses the `bottle.request` object to obtain the token.

If the handler in which this function is invoked is not decorated with *csrf_token()*, an `AttributeError` will be raised.

Returns HTML markup for hidden CSRF token field

bottle_utils.csrf.**csrf_token**(*func*)

Create and set CSRF token in preparation for subsequent POST request. This decorator is used to set the token. It also sets the `'Cache-Control'` header in order to prevent caching of the page on which the token appears.

When an existing token cookie is found, it is reused. The existing token is reset so that the expiration time is extended each time it is reused.

The POST handler must use the *csrf_protect()* decotrator for the token to be used in any way.

The token is available in the `bottle.request` object as `csrf_token` attribute:

```
    @app.get('/')
    @bottle.view('myform')
    @csrf_token
    def put_token_in_form():
        return dict(token=request.csrf_token)
```

In a view, you can render this token as a hidden field inside the form. The hidden field must have a name `_csrf_token`:

```
    <form method="POST">
        <input type="hidden" name="_csrf_token" value="{{ token }}">
        ....
    </form>
```

bottle_utils.csrf.**generate_csrf_token**()

Generate and set new CSRF token in cookie. The generated token is set to `request.csrf_token` attribute for easier access by other functions.

It is generally not necessary to use this function directly.

> **Warning:** This function uses `os.urandom()` call to obtain 8 random bytes when generating the token. It is possible to deplete the randomness pool and make the random token predictable.

3.4 Flash messages (bottle_utils.flash)

Flash messages are messages that are generated in one handler and displayed to the user in another. Commonly, this is done when you want to redirect to another path and still show the results of an operation performed in another. One example would be "You have logged out" message after logging the user out.

There are several ways to do this, but storing the message in a cookie is the most straightforward. This module provides methods for doing just that.

Unicode strings are fully supported.

3.4.1 Basic usage

In order to make flash messaging available to your app, install the *message_plugin()* plugin.

```
bottle.install(message_plugin)
```

This makes `bottle.request.message` object and `bottle.response.flash()` method available to all request handlers. To set a message, use `response.flash()`:

```
response.flash('This is my message')
```

To show the message in the interface, make the `request.message` object available to your template's context and simply output it in your template:

```
<p class="flash">{{ message }}</p>
```

3.4.2 How it works

When a message is set, it is stored in a cookie in the user's browser. The `bottle.request.message` is a lazy object (see *Lazy*), and **does not do anything until you actually use the message**. When you access the message object, it retrieves the text from the cookie and clears the cookie.

Warning: There is no mechanism for automatically clearing messages if they are not consumed. Therefore, it is important to consume it at the very next step user takes in your app. Otherwise, the message may appear on an unpexpected page at unexpected time, taken out of context, and confuse the user.

3.4.3 Functions and plugins

`bottle_utils.flash.`**`get_message`**`()`
Return currently set message and delete the cookie. This function is lazily evaluated so it's side effect of removing the cookie will only become effective when you actually use the message it returns.

`bottle_utils.flash.`**`message_plugin`**(*func*)
Manages flash messages. This is a Bottle plugin that adds attributes to `bottle.request` and `bottle.response` objects for setting and consuming the flash messages.

See *Basic usage*.

Example:

```
bottle.install(message_plugin)
```

`bottle_utils.flash.`**`set_message`**(*msg*)
Sets a message and makes it available via `request` object. This function sets the message cookie and assigns the message to the `bottle.request._message` attribute.

In Python 2.x, the message is UTF-8 encoded.

3.5 Template helpers (bottle_utils.html)

Contents of this module are mostly meant to be used inside the view templates, but their usage is not limited to templates by any means. bottle_utils.html contains a few data-formatting functions as well as shortcuts for generating HTML snippets and binding data to form fields.

3.5.1 Basic usage

One way to make the module contents available to templates is to add the module itself as a default template variable.:

```
import bottle
from bottle_utils import html
bottle.BaseTemplate.defaults['h'] = html
```

This allows you to use the module members by access the h variable in templates:

```
<html {{! h.attr('lang', request.locale }}>
```

Note: If your tempate engine auto-escapes HTML, you need to instruct it to unescape strings gneerated by some of the helper functions. For instance, in Bottle's SimpleTemplate engine, you need to enclose the strings in {{! }}.

3.5.2 Data formatting

bottle_utils.html.**hsize**(*size*, *unit=u'B'*, *step=1024*, *rounding=2*, *sep=u' '*)

Given size in unit produce size with human-friendly units. This is a simple formatting function that takes a value, a unit in which the value is expressed, and the size of multiple (kilo, mega, giga, etc).

This function rounds values to 2 decimal places and does not handle fractions. It also uses metric prefixes (K, M, G, etc) and only goes up to Peta (P, quadrillion) prefix. The number of decimal places can be customized using the rounding argument.

The size multiple (step parameter) is 1024 by default, suitable for expressing values related to size of data on disk.

The sep argument represents a separator between values and units.

Example:

```
>>> hsize(12)
'12.00 B'
>>> hsize(1030)
'1.01 KB'
>>> hsize(1536)
'1.50 KB'
>>> hsize(2097152)
'2.00 MB'
>>> hsize(12, sep='')
'12.00B'
```

bottle_utils.html.**plur**(*word*, *n*, *plural=<function <lambda>>*, *convert=<function <lambda>>*)

Pluralize word based on number of items. This function provides rudimentary pluralization support. It is quite flexible, but not a replacement for functions like ngettext.

This function takes two optional arguments, plural() and convert(), which can be customized to change the way plural form is derived from the original string. The default implementation is a naive version of English

language plural, which uses plural form if number is not 1, and derives the plural form by simply adding 's' to the word. While this works in most cases, it doesn't always work even for English.

The `plural(n)` function takes the value of the `n` argument and its return value is fed into the `convert()` function. The latter takes the source word as first argument, and return value of `plural()` call as second argument, and returns a string representing the pluralized word. Return value of the `convert(w, p)` call is returned from this function.

Here are some simple examples:

```
>>> plur('book', 1)
'book'
>>> plur('book', 2)
'books'

# But it's a bit naive
>>> plur('box', 2)
'boxs'
```

The latter can be fixed like this:

```
>>> exceptions = ['box']
>>> def pluralize(word, is_plural):
...     if not is_plural:
...         return word
...     if word in exceptions:
...         return word + 'es'
...     return word + 's'
>>> plur('book', 2)
'books'
>>> plur('box', 2, convert=pluralize)
'boxes'
```

bottle_utils.html.**strft**(*ts*, *fmt*)

Reformat string datestamp/timestamp. This function parses a string representation of a date and/or time and reformats it using specified format.

The format is standard strftime format used in Python's `datetime.datetime.strftime()` call.

Actual parsing of the input is delegated to python-dateutil library.

bottle_utils.html.**trunc**(*s*, *chars*)

Trucante string at `n` characters. This function hard-trucates a string at specified number of characters and appends an elipsis to the end.

The truncating does not take into account words or markup. Elipsis is not appended if the string is shorter than the specified number of characters.

```
>>> trunc('foobarbaz', 6)
'foobar...'
```

Note: Keep in mind that the trucated string is always 3 characters longer than `n` because of the appended elipsis.

---

bottle_utils.html.**yesno**(*val*, *yes=u'yes'*, *no=u'no'*)

Return `yes` or `no` depending on value. This function takes the value and returns either yes or no depending on whether the value evaluates to `True`.

Examples:

```
>>> yesno(True)
'yes'
>>> yesno(False)
'no'
>>> yesno(True, 'available', 'not available')
'available'
```

3.5.3 HTML rendering

bottle_utils.html.**tag**(*name*, *content=u''*, *nonclosing=False*, ***attrs*)

Wraps content in a HTML tag with optional attributes. This function provides a Pythonic interface for writing HTML tags with a few bells and whistles.

The basic usage looks like this:

```
>>> tag('p', 'content', _class="note", _id="note1")
'<p class="note" id="note1">content</p>'
```

Any attribute names with any number of leading underscores (e.g., '_class') will have the underscores strpped away.

If content is an iterable, the tag will be generated once per each member.

```
>>> tag('span', ['a', 'b', 'c'])
'<span>a</span><span>b</span><span>c</span>'
```

It does not sanitize the tag names, though, so it is possible to specify invalid tag names:

```
>>> tag('not valid')
'<not valid></not valid>
```

Warning: Please ensure that `name` argument does not come from user-specified data, or, if it does, that it is properly sanitized (best way is to use a whitelist of allowed names).

Because attributes are specified using keyword arguments, which are then treated as a dictionary, there is no guarantee of attribute order. If attribute order is important, don't use this function.

This module contains a few partially applied aliases for this function. These mostly have hard-wired first argument (tag name), and are all uppercase:

- `A` - alias for `<a>` tag
- `BUTTON` - alias for `<button>` tag
- `HIDDEN` - alias for `<input>` tag with `type="hidden"` attribute
- `INPUT` - alias for `<input>` tag with `nonclosing` set to `True`
- `LI` - alias for `<li>` tag
- `OPTION` - alias for `<option>` tag
- `P` - alias for `<p>` tag
- `SELECT` - alias for `<select>` tag
- `SPAN` - alias for `<span>` tag
- `SUBMIT` - alias for `<button>` tag with `type="submit"` attribute
- `TEXTAREA` - alias for `<textarea>` tag

- `UL` - alias for `<ul>` tag

`bottle_utils.html.`**`link_other`**(*label*, *url*, *path*, *wrapper=lambda l*, **kw: l*, ***kwargs*)

Only wrap label in anchor if given target URL, `url`, does not match the `path`. Given a label, this function will match the page URL against the path to which the anchor should point, and generate the anchor element markup as necessary. If the paths, match, `wrapper` will be used to generate the markup around the label.

Any additional keyword arguments are passed to the function that generates the anchor markup, which is `A()` alias of the *`tag()`* function.

If the URLs match (meaning the page URL matches the target path), the label will be passed to the wrapper function. The default wrapper function is `SPAN()`, so the label is wrapped in SPAN tag when the URLs matches.:

```
>>> link_other('foo', '/here', '/there')
'<a href="/target">foo</a>'
>>> link_other('foo', '/there', '/there')
'<span>foo</span>'
```

You can customize the appearance of the label in the case URLs match by customizing the wrapper:

```
>>> link_other('foo', '/there', '/there',
...            wrapper=lambda l, **kw: l + 'bar')
'foobar'
```

Note that the wrapper lambda function has wild-card keyword arguments. The wrapper function accepts the same extra keyword arguments that the anchor function does, so if you have common classes and similar attributes, you can specify them as extra keyword arguments and use any of the helper functions in this module.:

```
>>> link_other('foo', '/here', '/there', wrapper=BUTTON, _class='cls')
'<a class="cls" href="/target">foo</a>'
>>> link_other('foo', '/there', '/there', wrapper=BUTTON, _class='cls')
'<button class="cls">foo</button>'
```

`bottle_utils.html.`**`vinput`**(*name*, *values*, ***attrs*)

Render input with bound value. This function can be used to bind values to form inputs. By default it will result in HTML markup for a generic input. The generated input has a `name` attribute set to specified name, and an `id` attribute that has the same value.

```
>>> vinput('foo', {})
'<input name="foo" id="foo">'
```

If the supplied dictionary of field values contains a key that matches the specified name (case-sensitive), the value of that key will be used as the value of the input:

```
>>> vinput('foo', {'foo': 'bar'})
'<input name="foo" id="foo" value="bar">'
```

All values are properly sanitized before they are added to the markup.

Any additional keyword arguments that are passed to this function are passed on the *`tag()`* function. Since the generated input markup is for generic text input, some of the other usual input types can be specified using `_type` parameter:

```
>>> input('foo', {}, _type='email')
'<input name="foo" id="foo" type="email">'
```

`bottle_utils.html.`**`varea`**(*name*, *values*, ***attrs*)

Render textarea with bound value. Textareas use a somewhat different markup to that of regular inputs, so a separate function is used for binding values to this form control.:

```
>>> varea('foo', {'foo': 'bar'})
'<textarea name="foo" id="foo">bar</textarea>'
```

This function works the same way as *vinput()* function, so please look at it for more information. The primary difference is in the generated markup.

bottle_utils.html.**vcheckbox**(*name*, *value*, *values*, *default=False*, ***attrs*)

Render checkbox with bound value. This function renders a checkbox which is checked or unchecked depending on whether its own name-value combination appears in the provided form values dictionary.

Because there are many ways to think about checkboxes in general, this particular function may or may not work for you. It treats checkboxes as a list of alues which are all named the same.

Let's say we have markup that looks like this:

```
<input type="checkbox" name="foo" value="1">
<input type="checkbox" name="foo" value="2">
<input type="checkbox" name="foo" value="3">
```

If user checks all of them, we consider it a list `foo=['1', '2', '3']`. If user checks only the first and last, we have `foo=['1', '3']`. And so on.

This function assumes that you are using this pattern.

The `values` map can either map the checkbox name to a single value, or a list of multiple values. In the former case, if the single value matches the value of the checkbox, the checkbox is checked. In the latter case, if value of the checkbox is found in the list of values, the checkbox is checked.:

```
>>> vcheckbox('foo', 'bar', {'foo': 'bar'})
'<input type="checkbox" name="foo" id="foo" value="bar" checked>'
>>> vcheckbox('foo', 'bar', {'foo': ['foo', 'bar', 'baz']})
'<input type="checkbox" name="foo" id="foo" value="bar" checked>'
>>> vcheckbox('foo', 'bar', {'foo': ['foo', 'baz']})
'<input type="checkbox" name="foo" id="foo" value="bar">'
```

When the field values dictionary doesn't contain a key that matches the checkbox name, the value of `default` keyword argument determines whether the checkbox should be checked:

```
>>> vcheckbox('foo', 'bar', {}, default=True)
'<input type="checkbox" name="foo" id="foo" value="bar" checked>'
```

bottle_utils.html.**vselect**(*name*, *choices*, *values*, *empty=None*, ***attrs*)

Render select list with bound value. This function renders the select list with option elements with appropriate element selected based on field values that are passed.

The values and labels for option elemnets are specified using an iterable of two-tuples:

```
>>> vselect('foo', ((1, 'one'), (2, 'two'),), {})
'<select name="foo" id="foo"><option value="1">one</option><option...'
```

There is no mechanism for default value past what browsers support, so you should generally assume that most browsers will render the select with frist value preselected. Using an empty string or `None` as option value will render an option element without value:

```
>>> vselect('foo', ((None, '---'), (1, 'one'),), {})
'<select name="foo" id="foo"><option value>---</option><option val...'
>>> vselect('foo', (('', '---'), (1, 'one'),), {})
'<select name="foo" id="foo"><option value="">---</option><option ...'
```

When specifying values, keep in mind that only `None` is special, in that it will crete a `value` attribute without any value. All other Python types become strings in the HTML markup, and are submitted as such. You will need to convert the values back to their appropriate Python type manually.

If the choices iterable does not contain an element representing the empty value (`None`), you can specify it using the `empty` parameter. The argument for `empty` should be a label, and the matching value is `None`. The emtpy value is always inseted at the beginning of the list.

`bottle_utils.html.`**`form`**(*method=None*, *action=None*, *csrf=False*, *multipart=False*, ***attrs*)

Render open form tag. This function renders the open form tag with additional features, such as faux HTTP methods, CSRF token, and multipart support.

All parameters are optional. Using this function without any argument has the same effect as naked form tag without any attributes.

Method names can be either lowercase or uppercase.

The methods other than GET and POST are faked using a hidden input with `_method` name and uppercase name of the HTTP method. The form will use POST method in this case. Server-side support is required for this feature to work.

Any additional keyword arguments will be used as attributes for the form tag.

3.5.4 URL handling

class `bottle_utils.html.`**`QueryDict`**(*qs=u''*)

Represents a query string in `bottle.MultiDict` format.

This class differs from the base `bottle.MultiDict` class in two ways. First, it is instantiated with raw query string, rather than a list of two-tuples:

```
>>> q = QueryDict('a=1&b=2')
```

The query string is parsed and converted to `MultiDict` format. This works exactly the same way as `request.query`.

Second difference is the way string coercion is handled. `QueryDict` instances can be converted back into a query string by coercing them into string or bytestring:

```
>>> str(q)
'a=1&b=2'
```

The coercion also happens when using the + operator to concatenate with other strings:

```
>>> 'foo' + q
'foo?a=1&b=2'
```

Notice that the '?' character is inserted when using the + operator.

Note: When converting back to string, the order of parameters in the resulting query string may differ from the original.

Furthermore, additional methods have been added to provide chaining capability in conjunction with `*_qparam()` functions in this module.

For instance:

```
>>> q = QueryDict('a=1&b=2')
>>> q.del_qparam('a').set_qparam(b=3).add_qparam(d=2, k=12)
>>> str(s)
'b=3&d=2&k=12'
```

When used with functions like *add_qparam()*, this provides a more intuitive API:

```
>>> qs = 'a=1&b=2'
>>> q = add_qparam(qs, c=2).set_qparam(a=2)
>>> str(q)
'a=2&b=2&c=2'
```

Since this class is a `bottle.MultiDict` subclass, you can expect it to behave the same way as a regular `MultiDict` object. You can assign values to keys, get values by key, get all items as a list of key-value tuples, and so on. Please consult the Bottle documentation for more information on how `MultiDict` objects work.

add_qparam(***params*)
Add query parameter. Any keyword arguments passed to this function will be converted to query parameters.

Returns the instance for further chaining.

del_qparam(**params*)
Remove a query parameter. Takes any number of parameter names to be removed.

Returns the instance for further chaining.

set_qparam(***params*)
Replace or add parameter. Any keyword arguments passed to this function will be converted to query parameters.

Returns the instance for further chaining.

to_qs()
Return the string representation of the query string with prepended '?' character.

bottle_utils.html.**add_qparam**(*qs=None*, ***params*)
Add parameter to query string

If query string is omitted `request.query_string` is used.

Any keyword arguments passed to this function will be converted to query parameters.

The returned object is a *QueryDict* instance, which is a `bottle.MultiDict` subclass.

Example:

```
>>> q = add_qparam('a=1', b=2)
>>> str(q)
'a=1&b=2'
>> q = add_qparam('a=1', a=2)
>>> str(q)
'a=1&a=2'
```

bottle_utils.html.**set_qparam**(*qs=None*, ***params*)
Replace or add parameters to query string

If query string is omitted `request.query_string` is used.

Any keyword arguments passed to this function will be converted to query parameters.

The returned object is a *QueryDict* instance, which is a `bottle.MultiDict` subclass.

bottle_utils.html.**del_qparam**(*qs=None*, **params*)
Remove query string parameters

If query string is None or empty, request.query_string is used.

Second and subsequent positional arguments are query parameter names to be removed from the query string.

The returned object is a *QueryDict* instance, which is a bottle.MultiDict subclass.

bottle_utils.html.**urlquote**(*value*)

bottle_utils.html.**urlunquote**(*value*)

bottle_utils.html.**quote_dict**(*mapping*)
URL quote keys and values of the passed in dict-like object.

Parameters **mapping** – bottle.MultiDict or dict-like object

Returns dict with url quoted values

bottle_utils.html.**quoted_url**(*route*, ***params*)
Return matching URL with it's query parameters quoted.

3.6 Form handling and validation (bottle_utils.form)

3.6.1 Validation

class bottle_utils.form.validators.**Validator**(*messages={}*)
Base validator class. This calss does not do much on its own. It is used primarily to build other validators.

The validate() method in the subclass performs the validation and raises a *ValidationError* exception when the data is invalid.

The messages argument is used to override the error messages for the validators. This argument should be a dictionary that maps the error names used by individual validators to the new messages. The error names used by validators is documented for each class. You can also dynamically obtain the names by inspecting the messages property on each of the validator classes.

messages = {}
Mapping between errors and their human-readabile messages

validate(*data*)
Perform actual validation over data. Should raise *ValidationError* if data does not pass the validation.

Two arguments are passed to the validation error, the error name and a dictionary with extra parameters (usually with value key that points to the value).

Error message is constructed based on the arguments, passed to the exception by looking up the key in the messages property to obtain the message, and then interpolating any extra parameters into the message.

This method does not need to return anything.

class bottle_utils.form.validators.**DateValidator**(*messages={}*)
Validates date fields. This validator attempts to parse the data as date (or date/time) data. When the data cannot be parsed, it fails.

Error name(s) date

class bottle_utils.form.validators.**InRangeValidator**(*min_value=None*, *max_value=None*, ***kwargs*)

Validates that value is within a range between two values. This validator works with any objects that support > and < operators.

The `min_value` and `max_value` arguments are used to set the lower and upper bounds respectively. The check for those bounds are only done when the arguments are supplied so, when both arguments are omitted, this validator is effectively pass-through.

Error name(s) min_val, max_val

class bottle_utils.form.validators.**LengthValidator**(*min_len=None*, *max_len=None*, ***kwargs*)

Validates that value's length is within specified range. This validator works with any objects that support the `len()` function.

The `min_len` and `max_len` arguments are used to set the lower and upper bounds respectively. The check for those bounds are only done when the arguments are supplied so, when both arguments are omitted, this validator is effectively pass-through.

Error name(s) min_len, max_len

class bottle_utils.form.validators.**Required**(*messages={}*)

Validates the presence of data. Technically, this validator fails for any data that is an empty string, a string that only contains whitespace, or data that is not a string and evaluates to `False` when coerced into boolean (e.g., 0, empty arrays and dicts, etc).

Error name(s) required

class bottle_utils.form.exceptions.**ValidationError**(*message*, *params=None*, *is_form=False*)

Error raised during field and form validation. The erorr object can be initialized using a message string, and two optional parameters, `params` and `is_form`.

The `params` is a dictionary of key-value pairs that are used to fill the message in with values that are only known at runtime. For example, if the message is `'{value} is invalid for this field'`, we can pass a `params` argument that looks like `{'value': foo}`. The `format()` method is called on the message.

Note that `message` is a key pointing to a value in the `messages` dictionary on the form and field objects, not the actual message.

3.6.2 Fields

class bottle_utils.form.fields.**DormantField**(*field_cls*, *args*, *kwargs*)

Proxy for unbound fields. This class holds the the field constructor arguments until the data can be bound to it.

You never need to use this class directly.

class bottle_utils.form.fields.**Field**(*label=None*, *validators=None*, *value=None*, *name=None*, *messages={}*, ***options*)

Form field base class. This class provides the base functionality for all form fields.

The `label` argument is used to specify the field's label.

The `validators` argument is used to specify the validators that will be used on the field data.

If any data should be bound to a field, the `value` argument can be used to specify it. Value can be a callable, in which case it is called and its return value used as `value`.

The `name` argument is used to specify the field name.

The `messages` argument is used to customize the validation error messages. These override any messages found in the `messages` attribute.

Any extra keyword attributes passed to the constructor are stored as `options` property on the instance.

exception `ValidationError`(*message*, *params=None*, *is_form=False*)
Error raised during field and form validation. The erorr object can be initialized using a message string, and two optional parameters, `params` and `is_form`.

The `params` is a dictionary of key-value pairs that are used to fill the message in with values that are only known at runtime. For example, if the message is `'{value} is invalid for this field'`, we can pass a `params` argument that looks like `{'value': foo}`. The `format()` method is called on the message.

Note that `message` is a key pointing to a value in the `messages` dictionary on the form and field objects, not the actual message.

`Field.`**`bind_value`**(*value*)
Binds a value. This method also sets the `is_value_bound` property to `True`.

`Field.`**`generic_error`**
Generic error message to be used when no messages match the validation error.

`Field.`**`is_valid`**`()`
Validate form field and return `True` is data is valid. If there is an error during Validation, the error object is stored in the `_error` property. Before validation, the raw value is processed using the `parse()` method, and stored in `processed_value` attribute.

When parsing fails with `ValueError` exception, a 'generic' error is stored. The default message for this error is stored in the `generic_error` property, and can be customized by passing a `'generic'` message as part of the `messages` constructor argument.

`Field.`**`is_value_bound`** **= None**
Whether value is bound to this field

`Field.`**`label`** **= None**
Field label

`Field.`**`messages`** **= {}**
Validation error messages.

`Field.`**`name`** **= None**
Field name

`Field.`**`options`** **= None**
Extra keyword argument passed to the constructor

`Field.`**`parse`**(*value*)
Parse the raw value and convert to Python object. Subclasses should return the value in it's correct type. In case the passed in value cannot be cast into it's correct type, the method should raise a `ValueError` exception with an appropriate error message.

`Field.`**`processed_value`** **= None**
Processed value of the field

`Field.`**`type`** **= 'text'**
Field markup type. This is arbitrary and normally used in the templates to differentiate between field types. It is up to the template author to decide how this should be treated.

`Field.`**`validators`** **= None**
Field validators

Field.**value = None**
Raw value of the field

class bottle_utils.form.fields.**BooleanField**(*label=None*, *validators=None*, *value=None*, *default=False*, ***options*)
Field for working with boolean values.

Two additional constructor arguments are added. The `default` argument is used to specify the default state of the field, which can be used in the template to, for instance, check or uncheck a checkbox or radio button. The `expected_value` is the base value of the field against which the bound value is checked: if they match, the Python value of the field is `True`.

Python type bool

Type checkbox

default = None
Default state of the field

expected_value = None
Base value of the field against which bound value is checked

class bottle_utils.form.fields.**DateField**(*label=None*, *validators=None*, *value=None*, ***options*)
Field for working with dates. This field overloads the base class' constructor to add a `DateValidator`.

Python type str (unicode in Python 2.x)

Type text

class bottle_utils.form.fields.**EmailField**(*label=None*, *validators=None*, *value=None*, *name=None*, *messages={}*, ***options*)
Field for working with emails.

Python type str (unicode in Python 2.x)

Type text

class bottle_utils.form.fields.**ErrorMixin**
Mixin class used to provide error rendering functionality.

error
Human readable error message. This property evaluates to empty string if there are no errors.

class bottle_utils.form.fields.**FileField**(*label=None*, *validators=None*, *value=None*, *name=None*, *messages={}*, ***options*)
Field for working with file uploads.

Python type raw value

Type file

class bottle_utils.form.fields.**FloatField**(*label=None*, *validators=None*, *value=None*, *name=None*, *messages={}*, ***options*)
Field for working with floating-point numbers.

Python type float

Type text

class bottle_utils.form.fields.**HiddenField**(*label=None*, *validators=None*, *value=None*, *name=None*, *messages={}*, ***options*)
Field for working with hidden inputs.

Python type str (unicode in Python 2.x)

Type hidden

class bottle_utils.form.fields.**IntegerField**(*label=None*, *validators=None*, *value=None*, *name=None*, *messages={}*, ***options*)

Field for working with integers.

Python type int

Type text

class bottle_utils.form.fields.**PasswordField**(*label=None*, *validators=None*, *value=None*, *name=None*, *messages={}*, ***options*)

Field for working with passwords.

Python type str (unicode in Python 2.x)

Type password

class bottle_utils.form.fields.**SelectField**(*label=None*, *validators=None*, *value=None*, *choices=None*, ***options*)

Field for dealing with select lists.

The `choices` argument is used to specify the list of value-label pairs that are used to present the valid choices in the interface. The field value is then checked against the list of value and the parser validates whether the supplied value is among the valid ones.

Python type str (unicode in Python 2.x)

Type select

choices = None
Iterable of value-label pairs of valid choices

class bottle_utils.form.fields.**StringField**(*label=None*, *validators=None*, *value=None*, *name=None*, *messages={}*, ***options*)

Field for working with string values.

Python type str (unicode in Python 2.x)

Type text

class bottle_utils.form.fields.**TextAreaField**(*label=None*, *validators=None*, *value=None*, *name=None*, *messages={}*, ***options*)

Field for working with textareas.

Python type str (unicode in Python 2.x)

Type textarea

3.6.3 Forms

class bottle_utils.form.forms.**Form**(*data=None*, *messages={}*)
Base form class to be subclassed. To define a new form subclass this class:

```
class NewForm(Form):
    field1 = Field('Field 1')
    field2 = Field('Field 2', [Required])
```

Forms support field pre- and post-procesors. These methods are named after the field names by prepending `preprocess_` and `postprocess_` respectively. For example:

```
class NewForm(Form):
    field1 = Field('Field 1')
    field2 = Field('Field 2', [Required])
```

```
        def preprocess_field1(self, value):
            return value.replace('this', 'that')

        def postprocess_field1(self, value):
            return value + 'done'
```

Preprocessors can be defined for individual fields, and are ran before any validation happens over the field's data. Preprocessors are also allowed to raise `ValidationError`, though their actual purpose is to perform some manipulation over the incoming data, before it is passed over to the validators. The return value of the preprocessor is the value that is going to be validated further.

Postprocessors perform a similar purpose as preprocessors, except that they are invoked after field-level validation passes. Their return value is the value that is going to be the stored as cleaned / validated data.

exception `ValidationError`(*message*, *params=None*, *is_form=False*)
Error raised during field and form validation. The erorr object can be initialized using a message string, and two optional parameters, `params` and `is_form`.

The `params` is a dictionary of key-value pairs that are used to fill the message in with values that are only known at runtime. For example, if the message is `'{value} is invalid for this field'`, we can pass a `params` argument that looks like `{'value': foo}`. The `format()` method is called on the message.

Note that `message` is a key pointing to a value in the `messages` dictionary on the form and field objects, not the actual message.

`Form.`**`field_errors`**
Dictionary of all field error messages. This property maps the field names to error message maps. Field names are mapped to fields' messages property, which maps error type to actual message. This dictionary can also be used to modify the messages because message mappings are not copied.

`Form.`**`field_messages`**
Alias for *`field_errors`* retained for

`Form.`**`fields`**
Dictionary of all the fields found on the form instance. The return value is never cached so dynamically adding new fields to the form is allowed.

`Form.`**`generic_error`**

`Form.`**`is_valid`**`()`
Perform full form validation over the initialized form. The method has the following side-effects:

- in case errors are found, the form's *errors* container is going to be populated accordingly.
- validated and processed values are going to be put into the *processed_data* dictionary.

Return value is a boolean, and is `True` is form data is valid.

`Form.`**`validate`**`()`
Perform form-level validation, which can check fields dependent on each other. The function is expected to be overridden by implementors in case form-level validation is needed, but it's optional. In case an error is found, a *ValidationError* exception should be raised by the function.

3.7 HTTP helpers (`bottle_utils.http`)

This module provides decorators for working with HTTP headers and other aspects of HTTP.

3.7.1 Module contents

bottle_utils.http.**format_ts**(*seconds=None*)
Given a timestamp in seconds since UNIX epoch, return a string representation suitable for use in HTTP headers according to RFC.

If `seconds` is omitted, the time is asumed to be current time.

bottle_utils.http.**get_mimetype**(*filename*)
Guess mime-type based on file's extension.

bottle_utils.http.**iter_read_range**(*fd*, *offset*, *length*, *chunksize=1048576*)
Return an iterator that allows reading files in chunks. The `fd` should be a file-like object that has a `read()` method. The `offset` value sets the start offset of the read. If the `fd` object does not support `seek()`, the file will be simply read up until offset, and the read data discarded.

`length` argument specifies the amount of data to read. The read is not done in one go, but in chunks. The size of a chunk is specified using `chunksize`.

This function is similar to `bottle._file_iter_range` but does not fail on missing `seek()` attribute (e.g., `StringIO` objects).

bottle_utils.http.**no_cache**(*func*)
Disable caching on a handler. The decorated handler will have `Cache-Control` header set to `private, no-cache`.

This is useful for responses that contain data that cannot be reused.

Simply deocrate a handler with it:

```
@app.get('/foo')
@no_cache
def not_cached():
    return 'sensitive data'
```

bottle_utils.http.**send_file**(*content*, *filename*, *size=None*, *timestamp=None*)
Convert file data into an HTTP response.

This method is used when the file data does not exist on disk, such as when it is dynamically generated.

Because the file does not exist on disk, the basic metadata which is usually read from the file itself must be supplied as arguments. The `filename` argument is the supposed filename of the file data. It is only used to set the Content-Type header, and you may safely pass in just the extension with leading period.

The `size` argument is the payload size in bytes. For streaming files, this can be particularly important as the ranges are calculated baed on content length. If `size` is omitted, then support for ranges is not advertise and ranges are never returned.

`timestamp` is expected to be in seconds since UNIX epoch, and is used to calculate Last-Modified HTTP headers, as well as handle If-Modified-Since header. If omitted, current time is used, and If-Modified-Since is never checked.

Note: The returned response is a completely new response object. Modifying the reponse object in the current request context is not going to affect the object returned by this function. You should modify the object returned by this function instead.

Example:

```
    def some_handler():
        import StringIO
        f = StringIO.StringIO('foo')
        return send_file(f, 'file.txt', 3, 1293281312)
```

The code is partly based on `bottle.static_file`, with the main difference being the use of file-like objects instead of files on disk.

3.8 Translation support (`bottle_utils.i18n`)

3.8.1 How it works

This module provides plugins and functions for translation and language selection.

Language selection is based on URLs. Each path in the app is augmented with locale prefix. Therefore, `/foo/bar/baz` becomes `/LOCALE/foo/bar/baz` where `LOCALE` is any of the locales chosen as translation targets. When one of the supported locales is found in the incoming request's path, then that locale is activated.

Translation is performed by calling several of the translation functions provied by this module. These are simple `gettext()` and `ngettext()` wrappers that are lazily evaluated with the help of *`Lazy`* class.

This module does not deal with message extraction or compilation. For this, you can use the standard GNU Gettext utilities.

3.8.2 Setting up the app for translation

To activate translations and language selection, you will need to configure the plugin and middleware.

Note: The bottle plugin class, *`I18NPlugin`*, double as WSGI middleware.

First prepare a list of languages you want to support:

```
LANGS = [
    ('de_DE', 'Deutsch'),
    ('en_US', 'English'),
    ('fr_FR', 'français'),
    ('es_ES', 'español'),
    ('zh_CN', '')
]
```

Also decide which locale you would like to use as default.

```
DEFAULT_LOCAL = 'en_US'
```

Finally you need to decide where you want to keep the locale directory where translations are looked up.

```
LOCALES_DIR = './locales'
```

To install the plugin and middle, you can simply pass the *`I18NPlugin`* class a bottle app object.

```
from bottle_utils.i18n import I18NPlugin
app = bottle.default_app()
wsgi_app = I18NPlugin(app,
                      languages=LANGS,
```

```
                        default_locale=DEFAULT_LOCALE,
                        locale_dir=LOCALES_DIR)
```

This installs both the Bottle plugin and the WSGI middleware, and returns a WSGI application object.

If, for any reason, you do not want the i18n WSGI middleware to be the first in the stack, you can chain middleware as usual:

```
from bottle_utils.i18n import I18NPlugin
app = bottle.default_app()
wsgi = SomeMiddleware(app)
wsgi = I18NPlugin(wsgi, *other_args)
wsgi.install_plugin(app)
wsgi = SomeOtherPlugin(wsgi)
```

The `install_plugin()` method only works on the wsgi app returned from the plugin class. After wrapping with another plugin, it is no longer available so it must be called immediately.

3.8.3 Translating in Python code

To translate in Python code, use the *`lazy_gettext()`*, *`lazy_ngettext()`*, and similar translation functions.

`lazy_gettext()` is usually imported as _, which is a common convention (alias) for `gettext()`. Other methods are aliased without the `lazy_` prefix.

```
from bottle_utils.i18n import lazy_ngettext as ngettext, lazy_gettext as _

def handler():
    return _('This is a translatable string')
```

This is a convention that allows the gettext utilities to successfully extract the translation strings.

> **Warning:** The translation functions provided by this module **do not work outside of request context**. If you call them in a separate thread or a subprocess, you will get an exception. If your design allows for it, convert the lazy instances to strings before passing them to code running outside the request context.

3.8.4 Translating in templates

Translating in templates is highly dependent on your template engine. Some engines like Jinja2 may provide their own i18n mechanisms. In engines like SimpleTemplate and Mako, the process is pretty straightfoward. The translation methods are available in the templates using the naming convention discussed in the *Translating in Python code* section.

```
<p>{{ _('Current time') }}: {{ time }}</p>
```

Note: In template engines that use Python in templates (SimpleTemplate, Mako, etc), the similarity between Python syntax and template syntax (the Python portion of the template anyway) allows us to extract messages from the templates the same way we do from Python code simply by asking the `xgettext` tool to treat the template files as Python source code.

3.8.5 Module contents

class bottle_utils.i18n.**I18NPlugin**(*app*, *langs*, *default_locale*, *locale_dir*, *domain='messages'*, *noplugin=False*)

Bottle plugin and WSGI middleware for handling i18n routes. This class is a middleware. However, if the `app` argument is a `Bottle` object (bottle app), it will also install itself as a plugin. The plugin follows the version 2 API and implements the `apply()` method which applies the plugin to all routes. The plugin and middleware parts were merged into one class because they depend on each other and can't really be used separately.

During initialization, the class will set up references to locales, directory paths, and build a mapping between locale names and appropriate gettext translation APIs. The translation APIs are created using the `gettext.translation()` call. This call tries to access matching .mo file in the locale directory, and will emit a warning if such file is not found. If a .mo file does not exist for a given locale, or it is not readable, the API for that locale will be downgraded to generic gettext API.

The class will also update the `bottle.BaseTemplate.defaults` dict with translation-related methods so they are always available in templates (at least those that are rendered using bottle's API. The following variables become available in all templates:

- `_`: alias for *lazy_gettext()*
- `gettext`: alias for *lazy_gettext()*
- `ngettext`: alias for *lazy_ngettext()*
- `pgettext`: alias for *lazy_pgettext()*
- `npgettext`: alias for `lazy_pngettext()`
- `languages`: iterable containing available languages as `(locale, name)` tuples

In addition, two functions for generating i18n-specific paths are added to the default context:

- *i18n_path()*
- *i18n_url()*

The middleware itself derives the desired locale from the URL. It does not read cookies or headers. It only looks for the `/ll_cc/` prefix where `ll` is the two-ltter language ID, and `cc` is country code. If it finds such a prefix, it will set the locale in the envionment dict (`LOCALE` key) and fix the path so it doesn't include the prefix. This allows the bottle app to have routes matching any number of locales. If it doesn't find the prefix, it will redirect to the default locale.

If there is no appropriate locale, and `LOCALE` key is therfore set to `None`, the plugin will automatically respond with a 302 redirect to a location of the default locale.

The plugin reads the `LOCALE` key set by the middleware, and aliases the API for that locale as `request.gettext`. It also sets `request.locale` attribute to the selected locale. These attributes are used by the `lazy_gettext`()` and *lazy_ngettext()*, as well as *i18n_path()* and *i18n_url()* functions.

The plugin installation during initialization can be competely suppressed, if you wish (e.g., you wish to apply the plugin yourself some other way).

The locale directory should be in a format which `gettext.translations()` understands. This is a path that contains a subtree matching this format:

```
locale_dir/LANG/LC_MESSAGES/DOMAIN.mo
```

The `LANG` should match any of the supported languages, and `DOMAIN` should match the specified domain.

match_locale(*path*)
Match the locale based on prefix in request path. You can customize this method for a different way of obtaining locale information.

Returning `None` from this method causes the plugin to use the default locale.

The return value of this method is stored in the environment dictionary as `LOCALE` key. It is then used by the plugin part of this class to provide translation methods to the rest of the app.

set_locale(*locale*)
Store the passed in `locale` in a 'locale' cookie, which is used to override the value of the global `default_locale`.

static strip_prefix(*path*, *locale*)
Strip the locale prefix from the path. This static method is used to recalculate the request path that should be passed to Bottle. The return value of this method replaces the `PATH_INFO` key in the environment dictionary, and the original path is saved in `ORIGINAL_PATH` key.

bottle_utils.i18n.**dummy_gettext**(*message*)
Mimic `gettext()` function. This is a passthrough function with the same signature as `gettext()`. It can be used to simulate translation for applications that are untranslated, without the overhead of calling the real `gettext()`.

bottle_utils.i18n.**dummy_ngettext**(*singular*, *plural*, *n*)
Mimic `ngettext()` function. This is a passthrough function with the same signature as `ngettext()`. It can be used to simulate translation for applications that are untranslated, without the overhead of calling the real `ngettext()`.

This function returns the verbatim singular message if `n` is 1, otherwise the verbatim plural message.

bottle_utils.i18n.**dummy_npgettext**(*context*, *singular*, *plural*, *n*)
Mimic `npgettext()` function. This is a passthrough function with teh same signature as `npgettext()`. It can be used to simulate translation for applications that are untranslated, without the overhead of calling the real `npgettext()` function.

bottle_utils.i18n.**dummy_pgettext**(*context*, *message*)
Mimic `pgettext()` function. This is a passthrough function with the same signature as `pgettext()`. It can be used to simulate translation for applications that are untranslated, without the overhead of calling the real ‘‘pgettext()‘.

bottle_utils.i18n.**full_path**()
Calculate full path including query string for current request. This is a helper function used by *i18n_path()*. It uses the current request context to obtain information about the path.

bottle_utils.i18n.**i18n_path**(**args*, ***kwargs*)
Return current request path or specified path for given or current locale. This function can be used to obtain paths for different locales.

If no `path` argument is passed, the *full_path()* is called to obtain the full path for current request.

If `locale` argument is omitted, current locale is used.

bottle_utils.i18n.**i18n_url**(**args*, ***kwargs*)
Return a named route in localized form. This function is a light wrapper around Bottle's `get_url()` function. It passes the result to *i18n_path()*.

If `locale` keyword argument is passed, it will be used instead of the currently selected locale.

bottle_utils.i18n.**i18n_view**(*tpl_base_name=None*, ***defaults*)
Renders a template with locale name as suffix. Unlike the normal view decorator, the template name should not have an extension. The locale names are appended to the base template name using underscore ('_') as separator, and lower-case locale identifier.

Any additional keyword arguments are used as default template variables.

For example:

```
@i18n_view('foo')
def render_foo():
    # Renders 'foo_en' for English locale, 'foo_fr' for French, etc.
    return
```

bottle_utils.i18n.**lazy_gettext**(**args*, ***kwargs*)
Lazily evaluated version of `gettext()`.

This function uses the appropriate Gettext API object based on the value of `bottle.request.gettext` set by the plugin. It will fail with `AttributeError` exception if the plugin is not installed.

bottle_utils.i18n.**lazy_ngettext**(**args*, ***kwargs*)
Lazily evaluated version of `ngettext()`.

This function uses the appropriate Gettext API object based on the value of `bottle.request.gettext` set by the plugin. It will fail with `AttributeError` exception if the plugin is not installed.

bottle_utils.i18n.**lazy_npgettext**(*context*, *singular*, *plural*, *n*)
bottle_utils.i18n.lazy_ngettext() wrapper with message context.

This function is a wrapper around *bottle_utils.i18n.lazy_ngettext()* that provides message context. It is useful in situations where messages are used in several different contexts for which separate translations may be required for different languages.

The function itself is not lazy, but it returns the return value of `lazy_ngettext()`, and it is effectively lazy. Hence the name.

bottle_utils.i18n.**lazy_pgettext**(*context*, *message*)
lazy_gettext() wrapper with message context.

This function is a wrapper around *lazy_gettext()* that provides message context. It is useful in situations where short messages (usually one word) are used in several different contexts for which separate translations may be needed in different languages.

The function itself is not lazily evaluated, but its return value comes from `lazy_gettext()` call, and it is effectively lazy as a result.

3.9 Lazy evaluation (`bottle_utils.lazy`)

This module provides classes and decorators for lazy evaluation of callables.

Lazily evaluated functions and methods return immediately without performing any work, and their work is performed at some later point in time when the results are actually needed. This is useful in situation when the result may or may not be needed (e.g., some branching may occur) or when evaluation of the result may not be possible at the time of a call but we know it will be possible later.

In Bottle Utils, lazy evaluation is used extensively in the *i18n* module. Because evaluating the translation context requires a request context, and it may not be available where the call happens, lazy evaluation postpones evaluation of the translation function until we are sure the request context exists (e.g., some translated string is stored as a defined constant somewhere and used later in the request handler).

3.9.1 How it works

When a function is wrapped in a `Lazy` object, it behaves like result of evaluating that function. It stores the function in one of its properties and waits for your code to actually try to use the result. If your code never uses the result, the wrapped function is never called. When your code uses the result, the function is then evaluated for the first time. This type of object is sometimes also referred to as a proxy.

The idea of 'using' the result is defined as an attempt to coerce or perform any action that triggers any of the magic methods. This includes things like adding or subtracting from another value, calling `str()`, `bool()` and similar methods, attempting string interpolation with either `format()` method or `%` operator, accessing indices or keys using subscript notation, and so on. Attempting to access methods and properties also counts as 'using'.

One caveat of this behavior is that, because lazy functions are created in one context and potentially evaluated in another, the state in which they are evaluated may change in unpredictable ways. On the other hand, this may create opportunities that would not exist without lazy evaluation.

More on lazy evaluation in general can be found in Content Creation Wiki.

3.9.2 Module contents

class `bottle_utils.lazy.`**`Lazy`**(*_func*, **args*, ***kwargs*)

Lazy proxy object. This proxy always evaluates the function when it is used.

Any positional and keyword arguments that are passed to the constructor are stored and passed to the function except the `_func` argument which is the function itself. Because of this, the wrapped callable cannot use an argument named `_func` itself.

class `bottle_utils.lazy.`**`CachingLazy`**(*_func*, **args*, ***kwargs*)

Caching version of the *`Lazy`* class. Unlike the parent class, this class only evaluates the callable once, and remembers the resutls. On subsequent use, it returns the original result. This is probably closer to the behavior of a normal return value.

`bottle_utils.lazy.`**`lazy`**(*fn*)

Convert a function into lazily evaluated version. This decorator causes the function to return a *`Lazy`* proxy instead of the actual results.

Usage is simple:

```
@lazy
def my_lazy_func():
    return 'foo'
```

`bottle_utils.lazy.`**`caching_lazy`**(*fn*)

Convert a function into cached lazily evaluated version. This decorator modifies the function to return a *`CachingLazy`* proxy instead of the actual result.

This decorator has no arguments:

```
@caching_lazy
def my_lazy_func():
    return 'foo'
```

3.10 Social metadata (`bottle_utils.meta`)

This module provides classes for working with social metadata (Facebook, Google+, Twitter).

3.10.1 How it works

There are two flavors of metadata classes, *SimpleMetadata* and *Metadata*. Both render the title tag and description met tag, while the latter also renders a full set of social meta tags used by Facebook, Google+, and Twitter.

Using any of the two classes, you normally instantiate an object passing it metadata as constructor arguments, and then simply use the objects as string values in the template. You can also call the `render()` method on the object, but that is redundant except in rare cases where `str` type is expected.

Example usage is provided for both of the classes.

Warning: Authors of this module use social metadata only very rarely. As such, the features found in this module may not always be up to date. Please file issues you find with the social metadata support to the GitHub issue tracker.

3.10.2 Classes

class `bottle_utils.meta.`**MetaBase**

Base class for metadata. This class is a simple placeholder to collect base functionality for various subclasses.

Currently, the only functionality this base class provides is calling *render()* method when `__str__()` magic method is called on the class.

render()

Render the meta object into HTML. In the base class this method renders to empty string.

class `bottle_utils.meta.`**Metadata**(*title=u''*, *description=u''*, *thumbnail=u''*, *url=u''*)

Complete set of social meta tags. This class renders a complete set of social meta tags including schema.org properties, OpenGraph tags, and Twitter Cards markup.

The meta tags are only rendered for the arguments that are specified (i.e., one or more of the `title`, `description`, `thumbnail`, `url`). The arguments map to common properties that have the following meanings:

- title: page title
- description: page description (usually used as message shown next to the post on a social network)
- thumbnail: (also known as 'image') appears as a thumbnail or banner alongside the post on a social network
- url: canonical URL of the page (usually an URL that does not include any query parameters that change it's appearance or generate unnecessary data, such as Google Analytics campaign tags, etc), which is used instead of the URL that was shared

To render social tags, simply instantiate an object using meta data of your choosing and render the object in template (treat is as a string).

Here is an example of what it may look like in a handler function:

```
@app.get('/my/shareable/page')
@bottle.view('page')
def handler():
    m = Metadata(title='My page',
                 description='A page about sharing',
                 thumbnail='/static/images/awesome.png',
                 url=bottle.request.path)
    return dict(meta=m)
```

And here is a template:

```
<html>
    <head>
        <meta charset="utf-8">
        {{! meta }}
    </head>
    <body>
    ....
    </body>
</html>
```

Note: In template engines that support automatic escaping of HTML, you need to suppress escaping. For instance, in SimpleTemplates, using `{{! }}` instead of `{{ }}` accomplishes this.

This class does not render any of the other numerous tags (authorship tags, for instance). However, the instance methods it provides can be used to render them.

For example, to render a Twitter creator tag in a template that has access to any instance of this class:

```
{{! meta.twitterprop('creator', '@OuternetForAll') }}
```

Note: When it comes to thumbnails and canonical URLs, the social networks usually expect to see a full URL (including scheme and hostname). However, it may not feel right to hard-code these things. This class automatically converts paths to full URLs based on request data, so passing paths is fine.

itemprop(*name*, *value*)

Render schema.org itemprop meta tag. This method renders a schema.org itemprop meta tag which uses the `itemprop` attribute to designate the name of the tag.

This form is used by Google, but it's otherwise an open standard for semantic markup. This method only renders meta tags, and not every other kind of markup that schema.org uses.

The arguments are rendered into the following markup:

```
<meta itemprop="$name" content="$value">
```

static **make_full**(*url*)

Convert an input to full URL if not already a full URL. This static method will ensure that the specified url is a full URL.

This method only checks if the provided URL starts with 'http', though, so it is possible to trick it using a path that looks like 'httpfoo': it is clearly not a full URL, but will be treated as one. If the input value is user-supplied, please perform a more through check.

Under the hood, this method uses *bottle_utils.common.full_url()* to convert paths to full URLs.

Parameters **url** – path or full URL

Returns full URL as per request data

nameprop(*namespace*, *name*, *value*)

Render a generic name property. This method renders a generic name property meta tag that uses `name` attribute to designate the tag name.

Each tag name consist of namespace and name parts. Most notably, Twitter Card markup uses this form.

The arguments are rendered like this:

```
<meta name="$namespace:$name" content="$value">
```

ogprop(*name*, *value*)
Renders OpenGraph meta tag. This method renders a property meta tag that uses 'og' namespace.

The arguments are rendered like this:

```
<meta property="og:$name" content="$value">
```

prop(*namespace*, *name*, *value*)
Render a generic property meta tag. This method renders a generic property meta tags that uses `property` attribute to designate the tag name.

Each tag name consists of two parts: namespace and name. Most notably, OpenGraph uses this form with 'og' namespace.

The arguments are rendered like this:

```
<meta property="$namespace:$name" content="$value">
```

render()
Render the meta object into HTML.

twitterprop(*name*, *value*)
Renders Twitter Card markup. This method renders Twitter Card markup meta data. That is a name property meta tag with 'twitter' namespace.

The arguments are rendered like so:

```
<meta name="twitter:$name" content="$value">
```

class bottle_utils.meta.**SimpleMetadata**(*title=u''*, *description=u''*)
The basic (classic) metadata. This class is used to render title tag and description meta tag.

Both title and description are option. If neither is supplied, it is rendered into empty string.

Here is a simple example handler:

```
@app.get('/my/shareable/page')
@bottle.view('page')
def handler():
    m = SimpleMetadata(title='My page',
                       description='A page about sharing')
    return dict(meta=m)
```

In the template, simply render this object in `<head>` section where you would normally have the `<title>` tag.:

```
<html>
    <head>
        <meta charset="utf-8">
        {{! meta }}
    </head>
    <body>
    ...
    </body>
</html>
```

This renders the following tags:

```
<title>My Page</title>
<meta name="description" content="A page about sharing">
```

Note: In template engines that support automatic escaping of HTML, you need to suppress escaping. For instance, in SimpleTemplates, using {{! }} instead of {{ }} accomplishes this.

static meta(*attr*, *name*, *value*)

Render a generic <meta> tag. This function is the basis for rendering most of the social meta tags.

The arguments are rendered like this:

```
<meta $attr="$name" content="$value">
```

render()

Render the meta object as HTML.

simple(*name*, *value*)

Render a simple 'name' meta tag. This function renders a meta tag that uses the 'name' attribute.

The arguments are rendered like this:

```
<meta name="$name" content="$value">
```

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

b

L

M

N

O

P

Q

R

S

T

U

V

Y