Flask-BabelPlus Documentation

Release 1.0.0

Peter Justin, Serge S. Koval, Armin Ronacher

Jun 02, 2020

Contents

Flask-BabelPlus is an extension to Flask that adds i18n and l10n support to any Flask application with the help of babel, pytz and speaklater. It has builtin support for date formatting with timezone support as well as a very simple and friendly interface to `gettext` translations.

CHAPTER 1

Installation

Install the extension with one of the following commands:

```
$ easy_install Flask-BabelPlus
```

or alternatively if you have pip installed:

```
$ pip install Flask-BabelPlus
```

Please note that Flask-BabelPlus requires Jinja 2.5. If you are using an older version you will have to upgrade or disable the Jinja support.

CHAPTER 2

Configuration

To get started all you need to do is to instanciate a `Babel` object after configuring the application:

```
from flask import Flask
from flask_babelplus import Babel

app = Flask(__name__)
app.config.from_pyfile('mysettings.cfg')
babel = Babel(app)
```

The main difference from Flask-BabelEx is, that you can configure Flask-BabelPlus when using the factory method of initializing extensions:

```
# Flask-BabelPlus
babel.init_app(app=app, default_domain=FlaskBBDomain(app))
```

The babel object itself can be used to configure the babel support further. Babel has two configuration values that can be used to change some internal defaults:

BABEL_DEFAULT_LOCALE	The default locale to use if no locale selector is registered. This defaults to `'en'`.
BABEL_DEFAULT_TIMEZONE	The timezone to use for user facing dates. This defaults to `'UTC'` which also is the timezone your application must use internally.

For more complex applications you might want to have multiple applications for different users which is where selector functions come in handy. The first time the babel extension needs the locale (language code) of the current user it will call a `localeselector()` function, and the first time the timezone is needed it will call a `timezoneselector()` function.

If any of these methods return *None* the extension will automatically fall back to what's in the config. Furthermore for efficiency that function is called only once and the return value then cached. If you need to switch the language between a request, you can `refresh()` the cache.

Example selector functions:

```
from flask import g, request

@babel.localeselector
def get_locale():
    # if a user is logged in, use the locale from the user settings
    user = getattr(g, 'user', None)
    if user is not None:
        return user.locale
    # otherwise try to guess the language from the user accept
    # header the browser transmits.  We support de/fr/en in this
    # example.  The best match wins.
    return request.accept_languages.best_match(['de', 'fr', 'en'])

@babel.timezoneselector
def get_timezone():
    user = getattr(g, 'user', None)
    if user is not None:
        return user.timezone
```

The example above assumes that the current user is stored on the `flask.g` object.

CHAPTER 3

Formatting Dates

To format dates you can use the *format_datetime()*, *format_date()*, *format_time()* and *format_timedelta()* functions. They all accept a datetime.datetime (or datetime.date, datetime.time and datetime.timedelta) object as first parameter and then optionally a format string. The application should use naive datetime objects internally that use UTC as timezone. On formatting it will automatically convert into the user's timezone in case it differs from UTC.

To play with the date formatting from the console, you can use the test_request_context() method:

```
>>> app.test_request_context().push()
```

Here some examples:

```
>>> from flask_babelplus import format_datetime
>>> from datetime import datetime
>>> format_datetime(datetime(1987, 3, 5, 17, 12))
u'Mar 5, 1987 5:12:00 PM'
>>> format_datetime(datetime(1987, 3, 5, 17, 12), 'full')
u'Thursday, March 5, 1987 5:12:00 PM World (GMT) Time'
>>> format_datetime(datetime(1987, 3, 5, 17, 12), 'short')
u'3/5/87 5:12 PM'
>>> format_datetime(datetime(1987, 3, 5, 17, 12), 'dd mm yyy')
u'05 12 1987'
>>> format_datetime(datetime(1987, 3, 5, 17, 12), 'dd mm yyyy')
u'05 12 1987'
```

And again with a different language:

```
>>> app.config['BABEL_DEFAULT_LOCALE'] = 'de'
>>> from flask_babelplus import refresh; refresh()
>>> format_datetime(datetime(1987, 3, 5, 17, 12), 'EEEE, d. MMMM yyyy H:mm')
u'Donnerstag, 5. M\xe4rz 1987 17:12'
```

For more format examples head over to the babel documentation.

CHAPTER 4

Using Translations

The other big part next to date formatting are translations. For that, Flask uses `gettext` together with Babel. The idea of gettext is that you can mark certain strings as translatable and a tool will pick all those up, collect them in a separate file for you to translate. At runtime the original strings (which should be English) will be replaced by the language you selected.

There are two functions responsible for translating: *gettext()* and *ngettext()*. The first to translate singular strings and the second to translate strings that might become plural. Here some examples:

```
from flask_babelplus import gettext, ngettext

gettext(u'A simple string')
gettext(u'Value: %(value)s', value=42)
ngettext(u'%(num)s Apple', u'%(num)s Apples', number_of_apples)
```

Additionally if you want to use constant strings somewhere in your application and define them outside of a request, you can use a lazy strings. Lazy strings will not be evaluated until they are actually used. To use such a lazy string, use the *lazy_gettext()* function:

```
from flask_babelplus import lazy_gettext

class MyForm(formlibrary.FormBase):
    success_message = lazy_gettext(u'The form was successfully saved.')
```

So how does Flask-BabelPlus find the translations? Well first you have to create some. Here is how you do it:

4.1 Translating Applications

First you need to mark all the strings you want to translate in your application with *gettext()* or *ngettext()*. After that, it's time to create a `.pot` file. A `.pot` file contains all the strings and is the template for a `.po` file which contains the translated strings. Babel can do all that for you.

First of all you have to get into the folder where you have your application and create a mapping file. For typical Flask applications, this is what you want in there:

```
[python: **.py]
[jinja2: **/templates/**.html]
extensions=jinja2.ext.autoescape,jinja2.ext.with_
```

Save it as `babel.cfg` or something similar next to your application. Then it's time to run the *pybabel* command that comes with Babel to extract your strings:

```
$ pybabel extract -F babel.cfg -o messages.pot .
```

If you are using the *`lazy_gettext()`* function you should tell pybabel that it should also look for such function calls:

```
$ pybabel extract -F babel.cfg -k lazy_gettext -o messages.pot .
```

This will use the mapping from the `babel.cfg` file and store the generated template in `messages.pot`. Now we can create the first translation. For example to translate to German use this command:

```
$ pybabel init -i messages.pot -d translations -l de
```

`-d translations` tells pybabel to store the translations in this folder. This is where Flask-BabelPlus will look for translations. Put it next to your template folder.

Now edit the `translations/de/LC_MESSAGES/messages.po` file as needed. Check out some gettext tutorials if you feel lost.

To compile the translations for use, `pybabel` helps again:

```
$ pybabel compile -d translations
```

What if the strings change? Create a new `messages.pot` like above and then let `pybabel` merge the changes:

```
$ pybabel update -i messages.pot -d translations
```

Afterwards some strings might be marked as fuzzy (where it tried to figure out if a translation matched a changed key). If you have fuzzy entries, make sure to check them by hand and remove the fuzzy flag before compiling.

Flask-BabelPlus looks for message catalogs in `translations` directory which should be located under Flask application directory. Default domain is "messages".

For example, if you want to have translations for German, Spanish and French, directory structure should look like this:

> translations/de/LC_MESSAGES/messages.mo translations/sp/LC_MESSAGES/messages.mo translations/fr/LC_MESSAGES/messages.mo

4.2 Translation Domains

By default, Flask-BabelPlus will use "messages" domain, which will make it use translations from the `messages.mo` file. It is not very convenient for third-party Flask extensions, which might want to localize themselves without requiring user to merge their translations into "messages" domain.

Flask-BabelPlus allows extension developers to specify which translation domain to use:

```
from flask_babelplus import Domain

mydomain = Domain(domain='myext')
```

(continues on next page)

(continued from previous page)

```
mydomain.lazy_gettext('Hello World!')
```

Domain contains all gettext-related methods (*gettext()*, *ngettext()*, etc).

In previous example, localizations will be read from the `myext.mo` files, but they have to be located in `translations` directory under users Flask application. If extension is distributed with the localizations, it is possible to specify their location:

```
from flask_babelplus import Domain

from flask_myext import translations
mydomain = Domain(translations.__path__[0])
```

`mydomain` will look for translations in extension directory with default (messages) domain.

It is also possible to change the translation domain used by default, either for each app or per request.

To set the *Domain* that will be used in an app, pass it to *Babel* on initialization:

```
from flask import Flask
from flask_babelplus import Babel, Domain

app = Flask(__name__)
domain = Domain(domain='myext')
babel = Babel(app, default_domain=domain)
```

Translations will then come from the `myext.mo` files by default.

To change the default domain in a request context, call the *as_default()* method from within the request context:

```
from flask import Flask
from flask_babelplus import Babel, Domain, gettext

app = Flask(__name__)
domain = Domain(domain='myext')
babel = Babel(app)

@app.route('/path')
def demopage():
    domain.as_default()

    return gettext('Hello World!')
```

`Hello World!` will get translated using the `myext.mo` files, but other requests will use the default `messages.mo`. Note that a *Babel* must be initialized for the app for translations to work at all.

CHAPTER 5

Troubleshooting

On Snow Leopard pybabel will most likely fail with an exception. If this happens, check if this command outputs UTF-8:

```
$ echo $LC_CTYPE
UTF-8
```

This is a OS X bug unfortunately. To fix it, put the following lines into your `~/.profile` file:

```
export LC_CTYPE=en_US.utf-8
```

Then restart your terminal.

CHAPTER 6

API

This part of the documentation documents each and every public class or function from Flask-BabelPlus.

6.1 Configuration

class `flask_babelplus.`**`Babel`**(*app=None*, ***kwargs*)

Central controller class that can be used to configure how Flask-Babel behaves. Each application that wants to use Flask-Babel has to create, or run `init_app()` on, an instance of this class after the configuration was initialized.

default_locale

The default locale from the configuration as instance of a *babel.Locale* object.

default_timezone

The default timezone from the configuration as instance of a *pytz.timezone* object.

init_app(*app*, *default_locale='en'*, *default_timezone='UTC'*, *date_formats=None*, *configure_jinja=True*, *default_domain=None*)

Initializes the Flask-BabelPlus extension.

Parameters

- **app** – The Flask application.
- **default_locale** – The default locale which should be used. Defaults to 'en'.
- **default_timezone** – The default timezone. Defaults to 'UTC'.
- **date_formats** – A mapping of Babel datetime format strings
- **configure_jinja** – If set to `True` some convenient jinja2 filters are being added.
- **default_domain** – The default translation domain.

list_translations()

Returns a list of all the locales translations exist for. The list returned will be filled with actual locale objects and not just strings.

New in version 0.6.

load_locale(*locale*)
Load locale by name and cache it. Returns instance of a *babel.Locale* object.

localeselector(*f*)
Registers a callback function for locale selection. The default behaves as if a function was registered that returns *None* all the time. If *None* is returned, the locale falls back to the one from the configuration.

This has to return the locale as string (eg: `'de_AT'`, ''*en_US*'')

timezoneselector(*f*)
Registers a callback function for timezone selection. The default behaves as if a function was registered that returns *None* all the time. If *None* is returned, the timezone falls back to the one from the configuration.

This has to return the timezone as string (eg: `'Europe/Vienna'`)

6.2 Context Functions

flask_babelplus.**get_locale**()
Returns the locale that should be used for this request as *babel.Locale* object. This returns *None* if used outside of a request.

flask_babelplus.**get_timezone**()
Returns the timezone that should be used for this request as *pytz.timezone* object. This returns *None* if used outside of a request.

6.3 Translation domains

class flask_babelplus.**Domain**(*dirname=None*, *domain='messages'*)
Localization domain. By default it will look for tranlations in the Flask application directory and "messages" domain - all message catalogs should be called `messages.mo`.

as_default()
Set this domain as the default one for the current request

get_translations()
Returns the correct gettext translations that should be used for this request. This will never fail and return a dummy translation object if used outside of the request or if a translation cannot be found.

get_translations_cache()
Returns a dictionary-like object for translation caching

get_translations_path(*app*)
Returns the translations directory path. Override if you want to implement custom behavior.

gettext(*string*, ***variables*)
Translates a string with the current locale and passes in the given keyword arguments as mapping to a string formatting string.

```
gettext(u'Hello World!')
gettext(u'Hello %(name)s!', name='World')
```

lazy_gettext(*string*, ***variables*)
Like `gettext()` but the string returned is lazy which means it will be translated when it is used as an actual string.

Example:

```
hello = lazy_gettext(u'Hello World')

@app.route('/')
def index():
    return unicode(hello)
```

lazy_ngettext (*singular*, *plural*, *num*, ***variables*)

Like `ngettext()` but the string returned is lazy which means it will be translated when it is used as an actual string.

Example:

```
a = lazy_ngettext(u'%(num)d Apple', u'%(num)d Apples', num=len(a))

@app.route('/')
def index():
    return unicode(a)
```

lazy_pgettext (*context*, *string*, ***variables*)

Like `pgettext()` but the string returned is lazy which means it will be translated when it is used as an actual string.

New in version 0.7.

ngettext (*singular*, *plural*, *num*, ***variables*)

Translates a string with the current locale and passes in the given keyword arguments as mapping to a string formatting string. The *num* parameter is used to dispatch between singular and various plural forms of the message. It is available in the format string as `%(num)d` or `%(num)s`. The source language should be English or a similar language which only has one plural form.

```
ngettext(u'%(num)d Apple', u'%(num)d Apples', num=len(apples))
```

npgettext (*context*, *singular*, *plural*, *num*, ***variables*)

Like `ngettext()` but with a context.

New in version 0.7.

pgettext (*context*, *string*, ***variables*)

Like `gettext()` but with a context.

Gettext uses the `msgctxt` notation to distinguish different contexts for the same `msgid`

For example:

```
pgettext(u'Button label', 'Log in')
```

Learn more about contexts here: https://www.gnu.org/software/gettext/manual/html_node/Contexts.html

New in version 0.7.

6.4 Datetime Functions

`flask_babelplus.`**to_user_timezone** (*datetime*)

Convert a datetime object to the user's timezone. This automatically happens on all date formatting unless rebasing is disabled. If you need to convert a `datetime.datetime` object at any time to the user's timezone (as returned by `get_timezone()` this function can be used).

`flask_babelplus.to_utc(datetime)`

Convert a datetime object to UTC and drop tzinfo. This is the opposite operation to `to_user_timezone()`.

`flask_babelplus.format_datetime(datetime=None, format=None, rebase=True)`

Return a date formatted according to the given pattern. If no `datetime` object is passed, the current time is assumed. By default rebasing happens which causes the object to be converted to the users's timezone (as returned by `to_user_timezone()`). This function formats both date and time.

The format parameter can either be `'short'`, `'medium'`, `'long'` or `'full'` (in which case the language's default for that setting is used, or the default from the `Babel.date_formats` mapping is used) or a format string as documented by Babel.

This function is also available in the template context as filter named *datetimeformat*.

`flask_babelplus.format_date(date=None, format=None, rebase=True)`

Return a date formatted according to the given pattern. If no `datetime` or `date` object is passed, the current time is assumed. By default rebasing happens which causes the object to be converted to the users's timezone (as returned by `to_user_timezone()`). This function only formats the date part of a `datetime` object.

The format parameter can either be `'short'`, `'medium'`, `'long'` or `'full'` (in which cause the language's default for that setting is used, or the default from the `Babel.date_formats` mapping is used) or a format string as documented by Babel.

This function is also available in the template context as filter named *dateformat*.

`flask_babelplus.format_time(time=None, format=None, rebase=True)`

Return a time formatted according to the given pattern. If no `datetime` object is passed, the current time is assumed. By default rebasing happens which causes the object to be converted to the users's timezone (as returned by `to_user_timezone()`). This function formats both date and time.

The format parameter can either be `'short'`, `'medium'`, `'long'` or `'full'` (in which cause the language's default for that setting is used, or the default from the `Babel.date_formats` mapping is used) or a format string as documented by Babel.

This function is also available in the template context as filter named *timeformat*.

`flask_babelplus.format_timedelta(datetime_or_timedelta, granularity='second', add_direction=False, threshold=0.85)`

Format the elapsed time from the given date to now or the given timedelta. This function is also available in the template context as filter named *timedeltaformat*.

6.5 Gettext Functions

These are just shortcuts for the default Flask domain.

`flask_babelplus.gettext()`

Equivalent to `Domain.gettext()`.

`flask_babelplus.ngettext()`

Equivalent to `Domain.ngettext()`.

`flask_babelplus.pgettext()`

Equivalent to `Domain.pgettext()`.

`flask_babelplus.npgettext()`

Equivalent to `Domain.npgettext()`.

`flask_babelplus.lazy_gettext()`

Equivalent to *Domain.lazy_gettext()*.

`flask_babelplus.`**`lazy_ngettext`**`()`

Equivalent to *Domain.lazy_ngettext()*.

`flask_babelplus.`**`lazy_pgettext`**`()`

Equivalent to *Domain.lazy_pgettext()*.

6.6 Low-Level API

`flask_babelplus.`**`refresh`**`()`

Refreshes the cached timezones and locale information. This can be used to switch a translation between a request and if you want the changes to take place immediately, not just with the next request:

```
user.timezone = request.form['timezone']
user.locale = request.form['locale']
refresh()
flash(gettext('Language was changed'))
```

Without that refresh, the `flash()` function would probably return English text and a now German page.

`flask_babelplus.`**`force_locale`**`(`**args*, ***kwds*`)`

Temporarily overrides the currently selected locale. Sometimes it is useful to switch the current locale to different one, do some tasks and then revert back to the original one. For example, if the user uses German on the web site, but you want to send them an email in English, you can use this function as a context manager:

```
with force_locale('en_US'):
    send_email(gettext('Hello!'), ...)
```

Parameters **`locale`** – The locale to temporary switch to (ex: 'en_US').

CHAPTER 7

Additional Information

7.1 Flask-BabelPlus Changelog

7.1.1 Version 2.3.0

Unreleased

- Add a lazy method for ngettext named `lazy_ngettext`.
- Remove the pytz.gae reference as `pytz` is available in the standard Python environment on the Google App Engine.

7.1.2 Version 2.2.0

Released on May 31th 2020.

- Dropped support for Python 2.7, 3.3 and 3.4
- Fixed a Babel deprecation warning for using `format_numbers` instead of `format_decimal`

7.1.3 Version 2.1.2

Released on May 29th 2020.

- Fix werkzeug import error `from werkzeug import ImmutableDict`. PR #3.

7.1.4 Version 2.1.1

Released on October 8th 2017.

- Fix cache not being set on domain.

7.1.5 Version 2.1.0

Released on May 29th 2017.

- Add speaklater module from 'Flask-Babel', which in turn is forked from 'mitsuhiko/speaklater' that includes some improvements and fixes.
- Timezone and Locale selectors are overridable now.

7.1.6 Version 2.0.0

Released on May 15th 2017.

- Split the whole extension in multiple smaller files and use a `_BabelState` object to safe the extensions state.
- It is no longer possible to use the formatting utilities without and initialized Flask-BabelPlus object.
- Add `force_locale` function which can temporarily overrides the currently selected locale.

7.1.7 Previous Versions

Prior to 1.6.0, no proper changelog was kept.

7.2 License


```
Copyright (c) 2016 by Peter Justin, Serge S. Koval, Armin Ronacher and
contributors.

Some rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are
met:

* Redistributions of source code must retain the above copyright
  notice, this list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above
  copyright notice, this list of conditions and the following
  disclaimer in the documentation and/or other materials provided
  with the distribution.

* The names of the contributors may not be used to endorse or
  promote products derived from this software without specific
  prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
```


(continues on next page)

(continued from previous page)

```
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

- search

Python Module Index

f

Index