Sanic Babel Documentation

Release 0.3.0

Lix Xu

May 11, 2021

Contents

NOTICE: Most of the codes are from flask-babel, and updated to match Sanic.

sanic-babel is an extension to Sanic that adds i18n and l10n support to any Sanic 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:

```
$ python3 -m pip install sanic-babel
```

or alternatively:

```
$ pip3 install sanic-babel
```

Please note that sanic-babel 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 sanic import Sanic
from sanic_babel import Babel

app = Sanic(__name__)
app.config.from_pyfile('mysettings.cfg')
babel = Babel(app, configure_jinja=False)
# or if app.ctx.jinja_env already there
# babel = Babel(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:

```
@babel.localeselector
def get_locale(request):
    # if a user is logged in, use the locale from the user settings
```

(continues on next page)

(continued from previous page)

```
    if request['current_user'] is not None:
        return request['current_user'].lang
    # otherwise try to guess the language from the user accept
    # header the browser transmits. The first wins.
    langs = request.headers.get('accept-language')
    if langs:
        return langs.split(';')[0].split(',')[0].replace('-', '_')

@babel.timezoneselector
def get_timezone(request):
    if request['current_user'] is not None:
        return request['current_user'].timezone
```

The example above assumes that the current user is stored on the `request` object as key name of *current_user*.

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 `SanicTestClient`:

```
>>> app = Sanic('test_text')
>>> @app.route('/')
>>> async def handler(request):
>>>     return text('Hello')
>>> request, response = app.test_client.get('/')
```

Here some examples:

```
>>> from sanic_babel import Babel, format_datetime
>>> from datetime import datetime
>>> babel = Babel(app, configure_jinja=False)
>>> format_datetime(datetime(1987, 3, 5, 17, 12), request=request)
'Mar 5, 1987 5:12:00 PM'
>>> format_datetime(datetime(1987, 3, 5, 17, 12), 'full', request=request)
'Thursday, March 5, 1987 5:12:00 PM World (GMT) Time'
>>> format_datetime(datetime(1987, 3, 5, 17, 12), 'short', request=request)
'3/5/87 5:12 PM'
>>> format_datetime(datetime(1987, 3, 5, 17, 12), 'dd mm yyy', request=request)
'05 12 1987'
>>> format_datetime(datetime(1987, 3, 5, 17, 12), 'dd mm yyyy', request=request)
'05 12 1987'
```

And again with a different language:

```
>>> app.config['BABEL_DEFAULT_LOCALE'] = 'de'
>>> from sanic_babel import refresh; refresh(request)
```

(continues on next page)

(continued from previous page)

```
>>> format_datetime(datetime(1987, 3, 5, 17, 12), 'EEEE, d. MMMM yyyy H:mm',␣
↪request=request)
'Donnerstag, 5. März 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:

```python
from sanic_babel import gettext, ngettext

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

NOTICE: If you're using sanic-jinja2, the *gettext* and *ngettext* are already partial functions with request when used in templates, so no need to pass *request* when using them in templates.

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:

```python
from sanic_babel import lazy_gettext

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

# in the other file that needs actual lazy string
@app.route('/')
async def index(request):
    s = str(success_message(request))
    # or
    # success_message(request)
    # s = str(success_message)
    return text(s)
```

NOTICE: `lazy_gettext()` needs *request* before accessing actual string value. You can use *str(lazy_text(request))* or call *lazy_text(request)* once, then you can do others like *flask-babel*.

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

CHAPTER 5

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-Babel 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.

CHAPTER 6

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 7

API

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

7.1 Configuration

class sanic_babel.**Babel**(*app=None*, *default_locale='en'*, *default_timezone='UTC'*, *date_formats=None*, *configure_jinja=True*)

Central controller class that can be used to configure how sanic-babel behaves. Each application that wants to use sanic-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*)

Set up this instance for use with *app*, if no app was passed to the constructor.

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.

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'`)

7.2 Context Functions

`sanic_babel.`**`get_translations`**(*request=None*)

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.

`sanic_babel.`**`get_locale`**(*request=None*)

Returns the locale that should be used for this request as *babel.Locale* object. This returns *Locale.parse('en')* if used outside of a request.

`sanic_babel.`**`get_timezone`**(*request=None*)

Returns the timezone that should be used for this request as *pytz.timezone* object. This returns *UTC* if used outside of a request.

7.3 Datetime Functions

`sanic_babel.`**`to_user_timezone`**(*datetime*, *request=None*)

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).

`sanic_babel.`**`to_utc`**(*datetime*, *request=None*)

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

`sanic_babel.`**`format_datetime`**(*datetime=None*, *format=None*, *rebase=True*, *request=None*)

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 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 *datetimeformat*.

`sanic_babel.`**`format_date`**(*date=None*, *format=None*, *rebase=True*, *request=None*)

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*.

`sanic_babel.`**`format_time`**(*time=None*, *format=None*, *rebase=True*, *request=None*)

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*.

`sanic_babel.`**`format_timedelta`**(*datetime_or_timedelta*, *granularity='second'*, *add_direction=False*, *threshold=0.85*, *request=None*)

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*.

7.4 Gettext Functions

`sanic_babel.`**`gettext`**(*string*, *request=None*, ***variables*)

Translates a string with the current locale and passes in the given keyword arguments as mapping to a string formatting string.

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

`sanic_babel.`**`ngettext`**(*singular*, *plural*, *num*, *request=None*, ***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('%(num)d Apple', '%(num)d Apples', request=request,
         num=len(apples))
```

`sanic_babel.`**`pgettext`**(*context*, *string*, *request=None*, ***variables*)

Like `gettext()` but with a context.

`sanic_babel.`**`npgettext`**(*context*, *singular*, *plural*, *num*, *request=None*, ***variables*)

Like `ngettext()` but with a context.

`sanic_babel.`**`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.

NOTE: As *sanic* does not provide something like *ctx_stack*, the *lazy object* should call with *request* before using as an actual string.

Example:

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

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

`sanic_babel.`**`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.

7.5 Low-Level API

`sanic_babel.`**`refresh`**(*request=None*)

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(request)
jinja.flash(gettext('Language was changed', request))
```

NOTICE: `jinja.flash()` function is from *sanic-jinja2* package.

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

`sanic_babel.`**`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', request):
    send_email(gettext('Hello!', request), ...)
```

Parameters

- **`locale`** – The locale to temporary switch to (ex: 'en_US').
- **`request`** – the current Request object

Python Module Index

Index