The Flask Mega-Tutorial, Part XIII: I18n and L10n

Posted by Miguel Grinberg *on* December 3, 2023 *under* Python Flask Programming

This is the thirteenth installment of the Flask Mega-Tutorial series, in which I'm going to tell you how to expand Microblog to support multiple languages. As part of that effort you will also learn about creating your own CLI extensions to the flask command.

You are reading the 2024 edition of the Flask Mega-Tutorial. The complete course is also available to order in e-book and paperback formats from Amazon. Thank you for your support!

If you are looking for the 2018 edition of this course, you can find it here.

For your reference, here is the complete list of articles in this series:

- Chapter 1: Hello, World!
- Chapter 2: Templates
- Chapter 3: Web Forms
- Chapter 4: Database
- Chapter 5: User Logins
- Chapter 6: Profile Page and Avatars
- Chapter 7: Error Handling
- Chapter 8: Followers
- Chapter 9: Pagination
- Chapter 10: Email Support
- Chapter 11: Facelift
- Chapter 12: Dates and Times
- Chapter 13: I18n and L10n (this article)
- Chapter 14: Ajax
- Chapter 15: A Better Application Structure
- Chapter 16: Full-Text Search
- Chapter 17: Deployment on Linux
- Chapter 18: Deployment on Heroku
- Chapter 19: Deployment on Docker Containers
- Chapter 20: Some JavaScript Magic
- Chapter 21: User Notifications

The topics of this chapter are Internationalization and Localization, commonly abbreviated I18n and L10n. To make my application friendly to people who do not speak English, I'm going to implement a translation workflow that, with the help of a team of translators, will allow me to offer the application to users in a choice of languages.

The GitHub links for this chapter are: [Browse](#), [Zip](#), [Diff](#).

Introduction to Flask-Babel

As you can probably guess, there is a Flask extension that makes working with translations very easy. The extension is called [Flask-Babel](#) and is installed with pip:

```
(venv) $ pip install flask-babel
```

As part of this chapter, I'm going to show you how to translate the application into Spanish, as I happen to speak that language. I could also work with translators fluent in other languages and support those as well. To keep track of the list of supported languages, I'm going to add a configuration variable:

config.py: Supported languages list.

```
class Config:
    # ...
    LANGUAGES = ['en', 'es']
```

I'm using two-letter language codes for this application, but if you need to be more specific, a country code can be added as well. For example, you could use `en-US`, `en-GB` and `en-CA` to support American, British and Canadian English as different languages.

The `Babel` instance is initialized a `locale_selector` argument, which must be set to a function that will be invoked for each request. The function can look at the user request and pick the best language translation to use for that request. Here is the initialization of the Flask-Babel extension:

```python
from flask import request
# ...
from flask_babel import Babel

def get_locale():
    return request.accept_languages.best_match(app.config['LANGUAGES']

app = Flask(__name__)
# ...
babel = Babel(app, locale_selector=get_locale)
# ...
```

Here I'm using an attribute of Flask's `request` object called `accept_languages`. This object provides a high-level interface to work with the Accept-Language header that clients send with a request. This header specifies the client language and locale preferences as a weighted list. The contents of this header can be configured in the browser's preferences page, with the default being usually imported from the language settings in the computer's operating system. Most people don't even know such a setting exists, but this is useful as users can provide a list of preferred languages, each with a weight. In case you are curious, here is an example of a complex `Accept-Languages` header:

```
Accept-Language: da, en-gb;q=0.8, en;q=0.7
```

This says that Danish (`da`) is the preferred language (with default weight = 1.0), followed by British English (`en-GB`) with a 0.8 weight, and as a last option generic English (`en`) with a 0.7 weight.

To select the best language, you need to compare the list of languages requested by the client against the languages the application supports, and using the client provided weights, find the best language. The logic to do this is somewhat complicated, but it is all encapsulated in the `best_match()` method of `request.accept_languages`, which takes the list of languages offered by the application as an argument and returns the best choice.

Marking Texts to Translate In Python Source Code

Okay, so now comes the bad news. The normal workflow when making an application available in multiple languages is to mark all the texts that need translations in the source code. After the texts are marked, Flask-Babel will scan all the files and extract those texts into a separate translation file using the gettext tool. Unfortunately this is a tedious task that needs to be done to enable translations.

I'm going to show you a few examples of this marking here, but you can get the complete set of changes from the GitHub repository link for this chapter shown above.

The way texts are marked for translation is by wrapping them in a function call that as a convention is called `_()`, just an underscore. The simplest cases are those where literal strings appear in the source code. Here is an example `flash()` statement:

```
from flask_babel import _
# ...
flash(_('Your post is now live!'))
```

The idea is that the `_()` function wraps the text in the base language (English in this case). This function will use the language selected by the `get_locale()` function to find the correct translation for a given client. The `_()` function then returns the translated text, which in this case will become the argument to `flash()`.

Unfortunately not all cases are that simple. Consider this other `flash()` call from the application:

```
flash(f'User {username} not found.')
```

This text has a dynamic component that is inserted in the middle of the static text. The `_()` function has a syntax that supports this type of texts, but it is based on the older string substitution syntax from Python:

```
flash(_('User %(username)s not found.', username=username))
```

There is an even harder case to handle. Some string literals are assigned outside a web request, usually when the application is starting up, so at the time these texts are evaluated there is no way to know what language to use. An example of this are the labels associated with form

fields. The only solution to handle those texts is to find a way to delay the evaluation of the strings until they are used, which is going to be under an actual request. Flask-Babel provides a *lazy evaluation* version of `_()` that is called `lazy_gettext()`:

```python
from flask_babel import lazy_gettext as _l

class LoginForm(FlaskForm):
    username = StringField(_l('Username'), validators=[DataRequired()]
    # ...
```

Here I'm importing this alternative translation function and renaming it to `_l()` so that it looks similar to the original `_()`. This new function wraps the text in a special object that triggers the translation to be performed later, when the string is used inside a request.

The Flask-Login extension flashes a message any time it redirects the user to the login page. This message is in English and comes from the extension itself. To make sure this message also gets translated, I'm going to override the default message and provide my own, wrapper with the `_l()` function for lazy processing:

```python
login = LoginManager(app)
login.login_view = 'login'
login.login_message = _l('Please log in to access this page.')
```

Marking Texts to Translate In Templates

In the previous section you've seen how to mark translatable texts in Python source code, but that is only a part of this process, as template files also have text. The `_()` function is also available in templates, so the process is fairly similar. For example, consider this snippet of HTML from *404.html*:

```html
<h1>File Not Found</h1>
```

The translation enabled version becomes:

```html
<h1>{{ _('File Not Found') }}</h1>
```

Note that here in addition to wrapping the text with `_()`, the `{{ ... }}` needs to be added, to force the `_()` to be evaluated instead of being considered a literal in the template.

For more complex phrases that have dynamic components, arguments can also be used:

```
<h1>{{ _('Hi, %(username)s!', username=current_user.username) }}</h1>
```

There is a particularly tricky case in *_post.html* that took me a while to figure out:

```
{% set user_link %}
    <a href="{{ url_for('user', username=post.author.username)
        {{ post.author.username }}
    </a>
{% endset %}
{{ _('%(username)s said %(when)s',
    username=user_link, when=moment(post.timestamp).fromNow())
```

The problem here is that I wanted the `username` to be a link that points to the profile page of the user, not just the name, so I had to create an intermediate variable called `user_link` using the `set` and `endset` template directives, and then pass that as an argument to the translation function.

As I mentioned above, you can [download](#) a version of the application with all the translatable texts in Python source code and templates marked.

Extracting Text to Translate

Once you have the application with all the `_()` and `_l()` in place, you can use the `pybabel` command to extract them to a *.pot* file, which stands for *portable object template*. This is a text file that includes all the texts that were marked as needing translation. The purpose of this file is to serve as a template to create translation files for each language.

The extraction process needs a small configuration file that tells `pybabel` what files should be scanned for translatable texts. Below you can see the *babel.cfg* that I created for this application:

> *babel.cfg*: PyBabel configuration file.

```
[python: app/**.py]
[jinja2: app/templates/**.html]
```

These lines define the filename patterns for Python and Jinja template files respectively. Flask-Babel will look for any files matching these patterns and scan them for texts that are wrapped for translation.

To extract all the texts to a *.pot* file, you can use the following command:

```
(venv) $ pybabel extract -F babel.cfg -k _l -o messages.pot .
```

The `pybabel extract` command reads the configuration file given in the `-F` option, then scans all the code and template files in the directories that match the configured sources, starting from the directory given in the command (the current directory or `.` in this case). By default, `pybabel` will look for `_()` as a text marker, but I have also used the lazy version, which I imported as `_l()`, so I need to tell the tool to look for those too with the `-k _l`. The `-o` option provides the name of the output file.

I should note that the *messages.pot* file is not a file that needs to be incorporated into the project. This is a file that can be easily regenerated whenever it is needed, simply by running the command above again. So there is no need to commit this file to source control.

Generating a Language Catalog

The next step in the process is to create a translation for each language that will be supported in addition to the base one, which in this case is English. I said I was going to start by adding Spanish (language code `es`), so this is the command that does that:

```
(venv) $ pybabel init -i messages.pot -d app/translations -l es
creating catalog app/translations/es/LC_MESSAGES/messages.po based on
```

The `pybabel init` command takes the *messages.pot* file as input and writes a new language catalog to the directory given in the `-d` option for the language specified in the `-l` option. I'm going to be installing all the translations in the *app/translations* directory, because that is where

Flask-Babel will expect translation files to be by default. The command will create a *es* subdirectory inside this directory for the Spanish data files. In particular, there will be a new file named *app/translations/es/LC_MESSAGES/messages.po*, that is where the translations need to be made.

If you want to support other languages, just repeat the above command with each of the language codes you want, so that each language gets its own repository with a *messages.po* file.

This `messages.po` file that created in each language repository uses a format that is a standard for language translations, the format used by the gettext utility. Here are a few lines from the beginning of the Spanish *messages.po*:

```
# Spanish translations for PROJECT.
# Copyright (C) 2021 ORGANIZATION
# This file is distributed under the same license as the PROJECT proje
# FIRST AUTHOR <EMAIL@ADDRESS>, 2021.
#
msgid ""
msgstr ""
"Project-Id-Version: PROJECT VERSION\n"
"Report-Msgid-Bugs-To: EMAIL@ADDRESS\n"
"POT-Creation-Date: 2021-06-29 23:23-0700\n"
"PO-Revision-Date: 2021-06-29 23:25-0700\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language: es\n"
"Language-Team: es <LL@li.org>\n"
"Plural-Forms: nplurals=2; plural=(n != 1)\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=utf-8\n"
"Content-Transfer-Encoding: 8bit\n"
"Generated-By: Babel 2.5.1\n"

#: app/email.py:21
msgid "[Microblog] Reset Your Password"
msgstr ""

#: app/forms.py:12 app/forms.py:19 app/forms.py:50
msgid "Username"
msgstr ""

#: app/forms.py:13 app/forms.py:21 app/forms.py:43
msgid "Password"
msgstr ""
```

If you skip the header, you can see that what follows is a list of strings that were extracted from the `_()` and `_l()` calls. For each text, you get a reference to the location of the text in your application. Then the `msgid` line contains the text in the base language, and the `msgstr` line that follows contains an empty string. Those empty strings need to be edited to have the version of the text in the target language.

There are many translation applications that work with `.po` files. If you feel comfortable editing the text file, then that's sufficient, but if you are working with a large project it may be recommended to work with a specialized translation editor. The most popular translation application is the open-source poedit, which is available for all major operating systems. If you are familiar with vim, then the po.vim plugin gives some key mappings that make working with these files easier.

Below you can see a portion of the Spanish *messages.po* after I added the translations:

```
#: app/email.py:21
msgid "[Microblog] Reset Your Password"
msgstr "[Microblog] Nueva Contraseña"

#: app/forms.py:12 app/forms.py:19 app/forms.py:50
msgid "Username"
msgstr "Nombre de usuario"

#: app/forms.py:13 app/forms.py:21 app/forms.py:43
msgid "Password"
msgstr "Contraseña"
```

The download package for this chapter also contains this file with all the translations in place, so that you don't have to worry about that part for this application.

The *messages.po* file is a sort of source file for translations. When you want to start using these translated texts, this file needs to be *compiled* into a format that is efficient to be used by the application at run-time. To compile all the translations for the application, you can use the `pybabel compile` command as follows:

```
(venv) $ pybabel compile -d app/translations
compiling catalog app/translations/es/LC_MESSAGES/messages.po to
app/translations/es/LC_MESSAGES/messages.mo
```

This operation adds a *messages.mo* file next to *messages.po* in each language repository. The *.mo* file is the file that Flask-Babel will use to load translations for the application.

After you create the *messages.mo* file for Spanish or any other languages you added to the project, these languages are ready to be used in the application. If you want to see how the application looks in Spanish, you can edit the language configuration in your web browser to have Spanish as the preferred language. For Chrome, this is in the Advanced part of the Settings page:
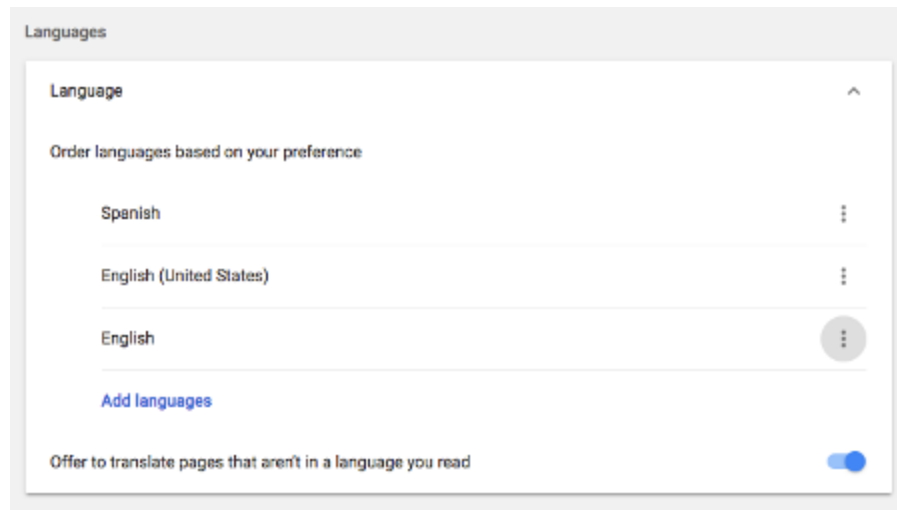
If you prefer not to change your browser settings, the other alternative is to force a language by making the `get_locale()` function always return the language that you want to use. For Spanish, this would be how you would do it:

app/__init__.py: Select best language.
| ```
def get_locale():
    # return request.accept_languages.best_match(app.config['LANGUAGES
    return 'es'
``` |

Running the application with the browser configured for Spanish, or the `get_locale()` function returning `es` will make all the texts appear in Spanish when you use the application.

Updating the Translations

One common situation when working with translations is that you may want to start using a translation file even if it is incomplete. That is

totally fine, you can compile an incomplete *messages.po* file and any translations that are available will be used, while any missing ones will use the base language. You can then continue working on the translations and compile again to update the *messages.mo* file as you make progress.

Another common scenario occurs if you missed some texts when you added the `_()` wrappers. In this case you are going to see that those texts that you missed are going to remain in English, because Flask-Babel knows nothing about them. In this situation you'll want to add the `_()` or `_l()` wrappers when you detect texts that don't have them, and then do an update procedure, which involves two steps:

```
(venv) $ pybabel extract -F babel.cfg -k _l -o messages.pot .
(venv) $ pybabel update -i messages.pot -d app/translations
```

The `extract` command is identical to the one I issued earlier, but now it will generate a new version of *messages.pot* with all the previous texts plus anything new that you recently wrapped with `_()` or `_l()`. The `update` call takes the new `messages.pot` file and merges it into all the *messages.po* files associated with the project. This is going to be an intelligent merge, in which any existing texts will be left alone, while only entries that were added or removed in *messages.pot* will be affected.

After the *messages.po* are updated, you can go ahead and translate any new texts, then compile the messages one more time to make them available to the application.

Translating Dates and Times

Now I have a complete Spanish translation for all the texts in Python code and templates, but if you run the application in Spanish and are a good observer, you will notice that there are still a few things that appear in English. I'm referring to the timestamps generated by Flask-Moment and moment.js, which obviously have not been included in the translation effort because none of the texts generated by these packages are part of the source code or templates of the application.

The moment.js library does support localization and internationalization, so all I need to do is configure the proper language. Flask-Babel returns the selected language and locale for a given request via the

`get_locale()` function, so what I'm going to do is add the locale to the `g` object in the `before_request` handler, so that I can then access it from the base template:

app/routes.py: Store selected language in flask.g.

```python
# ...
from flask import g
from flask_babel import get_locale

# ...

@app.before_request
def before_request():
    # ...
    g.locale = str(get_locale())
```

The `get_locale()` function from Flask-Babel returns a locale object, but I just want to have the language code, which can be obtained by converting the object to a string. Now that I have `g.locale`, I can access it from the base template to configure moment.js with the correct language:

app/templates/base.html: Set locale for moment.js.

```html
    ...

    {{ moment.include_moment() }}
    {{ moment.lang(g.locale) }}
  </body>
</html>
```

And now all dates and times should appear in the same language as the text. Below you can see how the application looks in Spanish:
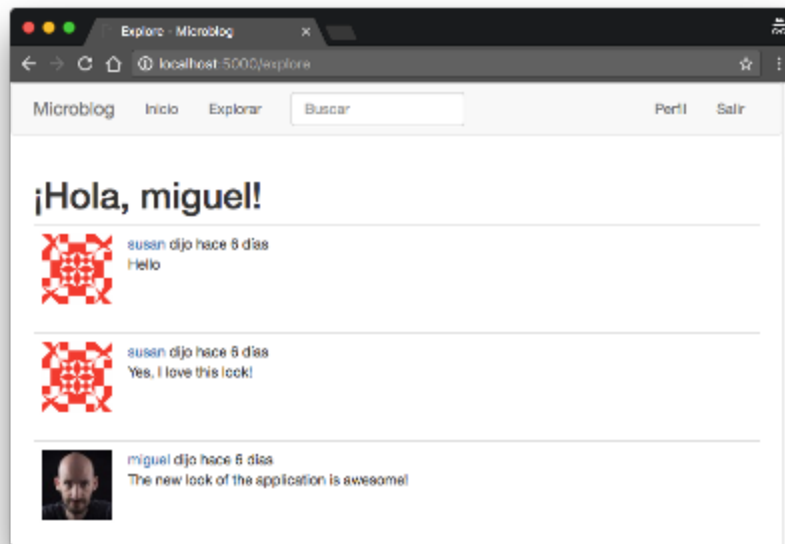
At this point, all texts except those that were provided by the user in blog posts or profile descriptions should be translatable into other languages.

Command-Line Enhancements

You will probably agree with me that the `pybabel` commands are a bit long and difficult to remember. I'm going to use this opportunity to show you how you can create custom commands that are integrated with the `flask` command. So far, you've seen me use `flask run`, `flask shell`, and several `flask db` sub-commands provided by the Flask-Migrate extension. It is actually easy to add application-specific commands to `flask` as well. So what I'm going to do now is create a few simple commands that trigger the `pybabel` commands with all the arguments that are specific to this application. The commands that I'm going to add are:

- `flask translate init LANG` to add a new language
- `flask translate update` to update all language repositories
- `flask translate compile` to compile all language repositories

The `babel extract` step is not going to be a command, because generating the *messages.pot* file is always a prerequisite to running either the `init` or the `update` commands, so the implementation of these commands will generate the translation template file as a temporary file.

Flask relies on [Click](#) for all its command-line operations. Commands like `translate`, which are a root for several sub-commands are created via the `app.cli.group()` decorator. I'm going to put these commands in a new module called *app/cli.py*:

---

app/cli.py: Translate command group.

```python
from app import app

@app.cli.group()
def translate():
    """Translation and localization commands."""
    pass
```

The name of the command comes from the name of the decorated function, and the help message comes from the *docstring*. Since this is a parent command that only exists to provide a base for the sub-commands, the function itself does not need to do anything.

The `update` and `compile` are easy to implement, because they do not take any arguments:

---

app/cli.py: Update and compile sub-commands.

```python
import os

# ...

@translate.command()
def update():
    """Update all languages."""
    if os.system('pybabel extract -F babel.cfg -k _l -o messages.pot .
        raise RuntimeError('extract command failed')
    if os.system('pybabel update -i messages.pot -d app/translations')
        raise RuntimeError('update command failed')
    os.remove('messages.pot')

@translate.command()
def compile():
    """Compile all languages."""
    if os.system('pybabel compile -d app/translations'):
        raise RuntimeError('compile command failed')
```

Note how the decorator from these functions is derived from the `translate` parent function. This may seem confusing, since `translate()` is a function, but it is the standard way in which Click

builds groups of commands. Same as with the `translate()` function, the docstrings for these functions are used as help message in the `--help` output.

You can see that for all commands, I run them and make sure that the return value is zero, which implies that the command did not return any error. If the command errors, then I raise a `RuntimeError`, which will cause the script to stop. The `update()` function combines the `extract` and `update` steps in the same command, and if everything is successful, it deletes the *messages.pot* file after the update is complete, since this file can be easily regenerated when needed again.

The `init` command takes the new language code as an argument. Here is the implementation:

app/cli.py: Init sub-command.

```python
import click

@translate.command()
@click.argument('lang')
def init(lang):
    """Initialize a new language."""
    if os.system('pybabel extract -F babel.cfg -k _l -o messages.pot .
        raise RuntimeError('extract command failed')
    if os.system(
            'pybabel init -i messages.pot -d app/translations -l ' + l
        raise RuntimeError('init command failed')
    os.remove('messages.pot')
```

This command uses the `@click.argument` decorator to define the language code. Click passes the value provided in the command to the handler function as an argument, and then I incorporate the argument into the `init` command.

The final step to enable these commands to work is to import them, so that the commands get registered. I decided to do this in the *microblog.py* file in the top-level directory:

microblog.py: Register command-line commands.

```python
from app import cli
```

Here, the only thing I need to do is import the new *cli.py* module, there is no need to do anything with it, as the import causes the command decorators to run and register each command.

At this point, running `flask --help` will list the `translate` command as an option. And `flask translate --help` will show the three sub-commands that I defined:

```
(venv) $ flask translate --help
Usage: flask translate [OPTIONS] COMMAND [ARGS]...

  Translation and localization commands.

Options:
  --help  Show this message and exit.

Commands:
  compile  Compile all languages.
  init     Initialize a new language.
  update   Update all languages.
```

So now, the workflow is much simpler and there is no need to remember long and complicated commands. To add a new language, you use:

```
(venv) $ flask translate init <language-code>
```

To update all the languages after making changes to the `_()` and `_l()` language markers:

```
(venv) $ flask translate update
```

And to compile all languages after updating the translation files:

```
(venv) $ flask translate compile
```

Continue on to the next chapter.

Buy me a coffee?

Thank you for visiting my blog! If you enjoyed this article, please consider supporting my work and keeping me caffeinated with a small one-time donation through Buy me a coffee. Thanks!

Share this post

43 comments

#1 **hcoco1** said a year ago

Miguel, thank you for this great tutorial. I can't get the app in Spanish. I clone your repo, install requirements.txt, and flask run. The app works, but in English and some lines in Spanish like: "Last seen on: 11 de diciembre de 2023 23:52" and "said hace 5 minutos" . I've noticed that my messages.po said :""Generated-By: Babel 2.13.1\n"" while yours said: ""Generated-By: Babel 2.5.1\n"". Do you think that babel and flask babel versions could be the problem? thanks again for teaching Flask through the Flask Megatutorial

#2 **Miguel Grinberg** said a year ago

@hcoco1: Are you following the tutorial or just trying to use the app? You seem to be missing steps, the translations need to be compiled.

#3 **Vladimir Kuzmenkov** said a year ago

Hi, Miguel,
It is often useful to let users switch between languages, say via a dropdown icon somewhere in the top right of the screen.
Yes, it is easy to add such menu in html css.
Yet, would you give some guidance on which steps are important to implement it within our python code?
Thanks!

#4 **Miguel Grinberg** said a year ago

@Vladimir: in general this is not necessary, because as you've seen in this article, the browser tells the server what are the language preferences of the user. But if you want to allow the user to select a language explicitly, you can store the chosen language for each user in the user table, and then return this language in the `get_locale()` function.

#5 **R Kovacs** said a year ago

Happy New Year!
I really appreciate this detailed tutorial, I am learning a great deal. I am looking at an issue in my work following the tutorial, the es .po file is out of sync with the posted code. For example, in the github for chapter 13 the **init**.py file the first string which is translated is on line 20, however the .po file is looking for it on line 24. I was hoping the 'Update Translations' using extract and update would have updated it. Perhaps I've got lost at this step of the process. Is it expected that the extract/update would have resolved the change in line numbers? Hey, enjoy the New Years weekend. No rush to respond.
Thanks!

#6 **Miguel Grinberg** said a year ago

@R: Is there anything not working with regards to the translations? The line numbers are informational only, what matters is the actual text.

#7 **R Kovacs** said a year ago

I had modified the app/**init**.py file to add the method 'get_locale():' to return the string 'es'. Everything stayed in English for me. I went back and tried the other method of adding Spanish to the configuration in the Browser, and yes it now works as expected. I'm back on track following the tutorial again. Thanks!

#8 **Miguel Grinberg** said a year ago

@R: It is the same `get_locale()` function that reads your browser preferences and decides the language, so forcing `"es"` in that function should have also worked.

#9 **R Kovacs** said a year ago

I took a closer look, and even with the browser having the top selection as Spanish, only the date/time is translated, not the strings from the .po file. The .mo file exists, and was created without error. I even have updated and compiled it again. I'll keep working at it, these disciplines (I18n and L10n) I really do want to learn and understand. Let me keep working at it a little longer and I'll let you know my progress. (I learn and remember a lot by debugging) Thanks again.

#10 **Miguel Grinberg** said a year ago

@R: The working code is available from the download link at the top of the article, by the way. You can try with my code if yours does not want to work.

#11 **Radim G** said a year ago

Hi Miguel, Is it possible that it still returns Error: No such command 'translate'., even though I did everything the same as you? I haven't had a problem with the tutorial so far...thanks

#12 **Miguel Grinberg** said a year ago

@Radim: Have you followed the tutorial from the very beginning? I suspect you have not set up your Flask application correctly. Check the value of your `FLASK_APP` variable and consult chapter 1 of this series to ensure it is correct.

#13 **Ciddu** said a year ago

hey Miguel, I have the same issue as Radim above, I can't manage to get flask translate to work, it returns back as no such command. Please advise

#14 **Miguel Grinberg** said a year ago

@Ciddu: I can't really add much. The working code is available for you to download from GitHub. If you get the problem with my version of the code, then it is your set up, so I would recommend to go back to Chapter 1 and make sure you have properly set up your application.

#15 **Jimbo_Jones_II** said a year ago

Hi Miguel, I updated just a few files in this chapter to get started with the translations (same as on your github): config.py, **init**.py , babel.cfg . When I run "(venv) $ pybabel extract -F babel.cfg -k _l -o messages.pot ." I get error "ValueError: Unknown extraction method 'jinja2'". I found it hard to debug this in the app (even rolled back to Xii code and started again) so I googled it and found a suggestion https://stackoverflow.com/a/20442638 to modify babel.cfg from "[jinja2: **.html]" to "** **[jinja2.ext:babel_extract[i18n]:** .html". After that I was able to generate the messages.pot file. I'm not sure of the significance of the change or whether this will cause any issues later on. Do you have any opinion on this error and work around?

#16 **Miguel Grinberg** said a year ago

@Jimbo_Jones_II: No, I don't believe your solution to be correct. That SO answer is from 2013, not something I would refer to in 2024. Your issue is likely that you have installed very old dependencies. You should check the requirements file in the microblog repo to see what versions I'm using, and make sure that you use those or newer.

#17 **Ashish M** said a year ago

Until now, I've got everything to work by following the instructions, and once resorting to the Zip file. Since I18N is less

immediately critical, I scrolled through this chapter without doing each step.

When I install from the Zip "baseline" file from Chapter 14, 'flask run' results in an "operational error" in the browser:

sqlalchemy.exc.OperationalError: (sqlite3.OperationalError) no such column: post.language
[SQL: SELECT post.id, post.body, post.timestamp, post.user_id, post.language
FROM post JOIN user AS user_1 ON user_1.id = post.user_id
LEFT OUTER JOIN (followers AS followers_1 JOIN user AS user_2 ON followers_1.follower_id = user_2.id) ON followers_1.followed_id = user_1.id
WHERE user_2.id = ? OR user_1.id = ? GROUP BY post.id, post.body, post.timestamp, post.user_id, post.language ORDER BY post.timestamp DESC
LIMIT ? OFFSET ?]
[parameters: (7, 7, 25, 0)]
(Background on this error at: https://sqlalche.me/e/20/e3q8)

Using the console logs, ChatGPT recommends the following steps, but they don't fix the problem:

% pip install langdetect
% pip install requests
% flask db migrate -m "Added language column to post table"
% flask db upgrade

I can try a completely clean install of the app (which will blow away the db?), but appreciate any other suggestions.

#18   Miguel Grinberg said a year ago

@ashish: it's hard for me to know what is the exact state your database is in, because you have blindly followed instructions from a bot, without fully understanding what you were doing (hint: the bot also doesn't have a good understanding!). So at this point it may be the case that your database is in a bad state and it would be easier to start over.

The correct approach when you skip chapters is to sync the database by running "flask db upgrade". You should do this every time you install a version of this application, and also when you

install any application based on Flask-Migrate. But as I said, after you've performed other destructive operations on your database this may now not work.

#19   **Ruben**   said a year ago

Hi Miguel,

Thank you for this awesome tutorial!
I'm trying to make the cli commands work, but the commands aren't showing up (in terminal: 'flask --help' and 'flask translate --help') after i add them to the microblog.py. Any idea why?

Kind regards

#20   Miguel Grinberg   said a year ago

@Ruben: I mentioned this already above several times. If my code does not work, then you haven't set up your application correctly. Go back to Chapter 1 to try to determine what you are missing.

#21   **Ruben**   said a year ago

@Miguel i found the error. I needed to do the export FLASK_APP=microblog.py again. Because my .flaskenv file was called flaskenv. (i literally missed the point). Thank you for pointing me in the right direction.

#22   **Torsten**   said a year ago

Hi everybody,

if you are working with PyCharm and you have defined FLASK_APP in the Run/Debug configuration of PyCharm, "translate" will not be recognised as a command (up to this point in the tutorial it was sufficient to do it like this). You really have to enter it in .flaskenv or the environmental settings of your computer.

Took me a while and hope it helps...

thank you Miguel! That's a very very nice introduction to Flask.

bless you!

I was having the same problem as #15, and couldn't find a requirements list in the download files as mentioned in #16 (I've later noticed that apparently they don't start getting included until v.15).

For me, the solution was this answer: https://stackoverflow.com/a/48241201 - setuptools needed updating

Thanks for the great tutorial, I've learned a lot and hope to learn more, much more.
I did everything step by step until this chapter.
I've read this chapter and installed flask-babel, downloaded the files from github, run flask db upgrade and decided to run pybabel, and that's what happened:
"(venv) C:\code\microblog>python -m flask shell
Python 3.12.3 (tags/v3.12.3:f6650f9, Apr 9 2024, 14:05:25) [MSC v.1938 64 bit (AMD64)] on win32
App: app
Instance: C:\code\microblog\instance

> *pybabel extract -F babel.cfg -k _l -o*
> *messages.pot .*
> *File "<console>", line 1*
> *pybabel extract -F babel.cfg -k _l -o*
> *messages.pot .*
> *^^^^^^^*
> *SyntaxError: invalid syntax*

> *"*
>
> *wtf? invalid syntaxt? what am I missing?*

Leave a Comment

Name

Email

Comment

Captcha

I'm not a robot

reCAPTCHA
Privacy - Terms

Submit

Flask Web Development, 2nd Edition

If you want to learn modern web development techniques with Python and Flask, you may find the second edition of my [O'Reilly book](#) useful.

[Click here to get this Book!](#)

About Miguel

Welcome to my blog!

I'm a software engineer and technical writer, currently living in Drogheda, Ireland.

You can also find me on [Github](#), [LinkedIn](#), [Bluesky](#), [Mastodon](#), [Twitter](#), [YouTube](#), and [Patreon](#).

Thank you for visiting!

Categories

- AI *2*
- Arduino *7*
- Authentication *10*
- Blog *1*
- C++ *5*
- CSS *1*
- Cloud *11*
- Database *23*

- Docker *5*
- Filmmaking *6*
- Flask *129*
- Games *1*
- IoT *8*
- JavaScript *36*
- MicroPython *9*
- Microdot *1*
- Microservices *2*
- Movie Reviews *5*
- Personal *3*
- Photography *7*
- Product Reviews *2*
- Programming *192*
- Project Management *1*
- Python *174*
- REST *7*
- Raspberry Pi *8*
- React *19*
- Reviews *1*
- Robotics *6*
- Security *12*
- Video *22*
- WebSocket *2*
- Webcast *3*
- Windows *1*