

The Flask Mega-Tutorial Part XIV: Ajax

(/post/the-flask-mega-tutorial-part-xiv-ajax)

March 6 2018

Posted by [Miguel Grinberg \(/author/Miguel Grinberg\)](/author/Miguel Grinberg) under [Python \(/category/Python\)](/category/Python),
[Programming \(/category/Programming\)](/category/Programming), [Flask \(/category/Flask\)](/category/Flask), [JavaScript \(/category/JavaScript\)](/category/JavaScript).

[Tweet](#)[Like](#)[Share](#)

This is the fourteenth installment of the Flask Mega-Tutorial series, in which I'm going to add a live language translation feature, using the Microsoft translation service and a little bit of JavaScript.

For your reference, below is a list of the articles in this series.

- Chapter 1: Hello, World! (/post/the-flask-mega-tutorial-part-i-hello-world)
- Chapter 2: Templates (/post/the-flask-mega-tutorial-part-ii-templates)
- Chapter 3: Web Forms (/post/the-flask-mega-tutorial-part-iii-web-forms)
- Chapter 4: Database (/post/the-flask-mega-tutorial-part-iv-database)
- Chapter 5: User Logins (/post/the-flask-mega-tutorial-part-v-user-logins)
- Chapter 6: Profile Page and Avatars (/post/the-flask-mega-tutorial-part-vi-profile-page-and-avatars)
- Chapter 7: Error Handling (/post/the-flask-mega-tutorial-part-vii-error-handling)
- Chapter 8: Followers (/post/the-flask-mega-tutorial-part-viii-followers)
- Chapter 9: Pagination (/post/the-flask-mega-tutorial-part-ix-pagination)
- Chapter 10: Email Support (/post/the-flask-mega-tutorial-part-x-email-support)
- Chapter 11: Facelift (/post/the-flask-mega-tutorial-part-xi-facelift)
- Chapter 12: Dates and Times (/post/the-flask-mega-tutorial-part-xii-dates-and-times)
- Chapter 13: I18n and L10n (/post/the-flask-mega-tutorial-part-xiii-i18n-and-l10n)
- Chapter 14: Ajax (/post/the-flask-mega-tutorial-part-xiv-ajax) (this article)
- Chapter 15: A Better Application Structure (/post/the-flask-mega-tutorial-part-xv-a-better-application-structure)
- Chapter 16: Full-Text Search (/post/the-flask-mega-tutorial-part-xvi-full-text-search)
- Chapter 17: Deployment on Linux (/post/the-flask-mega-tutorial-part-xvii-deployment-on-linux)
- Chapter 18: Deployment on Heroku (/post/the-flask-mega-tutorial-part-xviii-deployment-on-heroku)
- Chapter 19: Deployment on Docker Containers (/post/the-flask-mega-tutorial-part-xix-deployment-on-docker-containers)
- Chapter 20: Some JavaScript Magic (/post/the-flask-mega-tutorial-part-xx-some-javascript-magic)

- Chapter 21: User Notifications (/post/the-flask-mega-tutorial-part-xxi-user-notifications)
- Chapter 22: Background Jobs (/post/the-flask-mega-tutorial-part-xxii-background-jobs)
- Chapter 23: Application Programming Interfaces (APIs) (/post/the-flask-mega-tutorial-part-xxiii-application-programming-interfaces-apis)

Note 1: If you are looking for the legacy version of this tutorial, it's here (/post/the-flask-mega-tutorial-part-i-hello-world-legacy).

Note 2: If you would like to support my work on this blog, or just don't have patience to wait for weekly articles, I am offering the complete version of this tutorial packaged as an ebook or a set of videos. For more information, visit courses.miguelgrinberg.com (<https://courses.miguelgrinberg.com>).

In this article I'm going to take a departure from the "safe zone" of server-side development and to work on a feature that has equally important server and client-side components. Have you seen the "Translate" links that some sites show next to user generated content? These are links that trigger a real time automated translation of content that is not in the user's native language. The translated content is typically inserted below the original version. Google shows it for search results in foreign languages. Facebook does it for posts. Twitter does it for tweets. Today I'm going to show you how to add the very same feature to Microblog!

The GitHub links for this chapter are: Browse (<https://github.com/miguelgrinberg/microblog/tree/v0.14>), Zip (<https://github.com/miguelgrinberg/microblog/archive/v0.14.zip>), Diff (<https://github.com/miguelgrinberg/microblog/compare/v0.13...v0.14>).

Server-side vs. Client-side

In the traditional server-side model that I've followed so far there is a client (a web browser commanded by a user) making HTTP requests to the application server. A request can simply ask for an HTML page, like when you click the "Profile" link, or it can trigger an action, like when you click the Submit button after editing your profile information. In both types of requests the server completes the request by sending a new web page to the client, either directly or by issuing a redirect. The client then replaces the current page with the new one. This cycle repeats for as long as the user stays on the application's web site. In this model the server does all the work, while the client just displays the web pages and accepts user input.

There is a different model in which the client takes a more active role. In this model, the client issues a request to the server and the server responds with a web page, but unlike the previous case, not all the page data is HTML, there is also sections of the page with code, typically written in Javascript. Once the client receives the page it displays the HTML portions, and executes the code. From then on you have an active client that can do work on its own without little or no contact with the server. In a strict client-side application the entire application is downloaded to the client with the initial page request, and then the application runs entirely on the client, only contacting the server to retrieve or store data and making dynamic changes to the appearance of that first and only web page. This type of applications are called Single Page Applications (http://en.wikipedia.org/wiki/Single-page_application) or SPAs.

Most applications are a hybrid between the two models and combine techniques of both. My Microblog application is mostly a server-side application, but today I will be adding a little bit of client-side action to it. To do real time translations of user posts, the client browser will send *asynchronous requests* to the server, to which the server will respond without causing a page refresh. The client will then insert the translations into the current page dynamically. This technique is known as Ajax ([http://en.wikipedia.org/wiki/Ajax_\(programming\)](http://en.wikipedia.org/wiki/Ajax_(programming))), which is short for Asynchronous JavaScript and XML (even though these days XML is often replaced with JSON).

Live Translation Workflow

The application has good support for foreign languages thanks to Flask-Babel, which would make it possible to support as many languages as I can find translators for. But of course, there is one element missing. Users are going to write blog posts in their own languages, so it is quite possible that a user will come across posts that are written in unknown languages. The quality of automated translations isn't always great, but in most cases it is good enough if all you want is to have a basic idea of what a text in another language means.

This is an ideal feature to implement as an Ajax service. Consider that the index or explore pages could be showing several posts, some of which might be in foreign languages. If I implement the translation using traditional server-side techniques, a request for a translation would cause the original page to get replaced with a new page. The fact is that requesting a translation for one out of many displayed blogs posts isn't a big enough action to require a full page update, this feature works much better if the translated text is dynamically inserted below the original text while leaving the rest of the page untouched.

Implementing live automated translations requires a few steps. First, I need a way to identify the source language of the text to translate. I also need to know the preferred language for each user, because I want to show a "translate" link only for posts written in other languages. When a translation link is offered and the user clicks on it, I will need to send the Ajax request to the server, and the server will contact a third-party translation API. Once the server sends back a response with the translated text, the client-side javascript code will dynamically insert this text into the page. As you can surely notice, there are a few non-trivial problems here. I'm going to look at these one by one.

Language Identification

The first problem is identifying what language a post was written in. This isn't an exact science, as it is not always possible to unequivocally detect a language, but for most cases, automated detection works fairly well. In Python, there is a good language detection library called `guess_language`. The original version of this package is fairly old and was never ported to Python 3, so I'm going to install a derived version that supports Python 2 and 3:

```
(venv) $ pip install guess_language-spirit
```

The plan is to feed each blog post to this package, to try to determine the language. Since doing this analysis is somewhat time consuming, I don't want to repeat this work every time a post is rendered to a page. What I'm going to do is set the source language for a post at the time it is submitted. The detected language is then going to be stored in the posts table.

The first step is to add a `language` field to the `Post` model:

app/models.py: Add detected language to Post model.

```
class Post(db.Model):  
    # ...  
    language = db.Column(db.String(5))
```

As you recall, each time there is a change made to the database models, a database migration needs to be issued:

```
(venv) $ flask db migrate -m "add language to posts"  
INFO [alembic.runtime.migration] Context impl SQLiteImpl.  
INFO [alembic.runtime.migration] Will assume non-transactional DDL.  
INFO [alembic.autogenerate.compare] Detected added column 'post.language'  
Generating migrations/versions/2b017edaa91f_add_language_to_posts.py ... done
```

And then the migration needs to be applied to the database:

```
(venv) $ flask db upgrade
INFO [alembic.runtime.migration] Context impl SQLiteImpl.
INFO [alembic.runtime.migration] Will assume non-transactional DDL.
INFO [alembic.runtime.migration] Upgrade ae346256b650 -> 2b017edaa91f, add language to posts
```

I can now detect and store the language when a post is submitted:

app/routes.py: Save language for new posts.

```
from guess_language import guess_language

@app.route('/', methods=['GET', 'POST'])
@app.route('/index', methods=['GET', 'POST'])
@login_required
def index():
    form = PostForm()
    if form.validate_on_submit():
        language = guess_language(form.post.data)
        if language == 'UNKNOWN' or len(language) > 5:
            language = ''
        post = Post(body=form.post.data, author=current_user,
                    language=language)
        # ...
```

With this change, each time a post is submitted, I run the text through the `guess_language` function to try to determine the language. If the language comes back as unknown or if I get an unexpectedly long result, I play it safe and save an empty string to the database. I'm going to adopt the convention that any post that have the language set to an empty string is assumed to have an unknown language.

Displaying a "Translate" Link

The second step is easy. What I'm going to do now is add a "Translate" link next to any posts that are not in the language the is active for the current user.

app/templates/_post.html: Add a translate link to posts.

```
{% if post.language and post.language != g.locale %}
<br><br>
<a href="#">{{ _('Translate') }}</a>
{% endif %}
```

I'm doing this in the `_post.html` sub-template, so that this functionality appears on any page that displays blog posts. The translate link will only appear on posts for which the language was detected, and this language does not match the language selected by the function decorated with Flask-Babel's `localeselector` decorator. Recall from Chapter 13

(/post/the-flask-mega-tutorial-part-xiii-i18n-and-l10n) that the selected locale is stored as `g.locale`. The text of the link needs to be added in a way that it can be translated by Flask-Babel, so I used the `_()` function when I defined it.

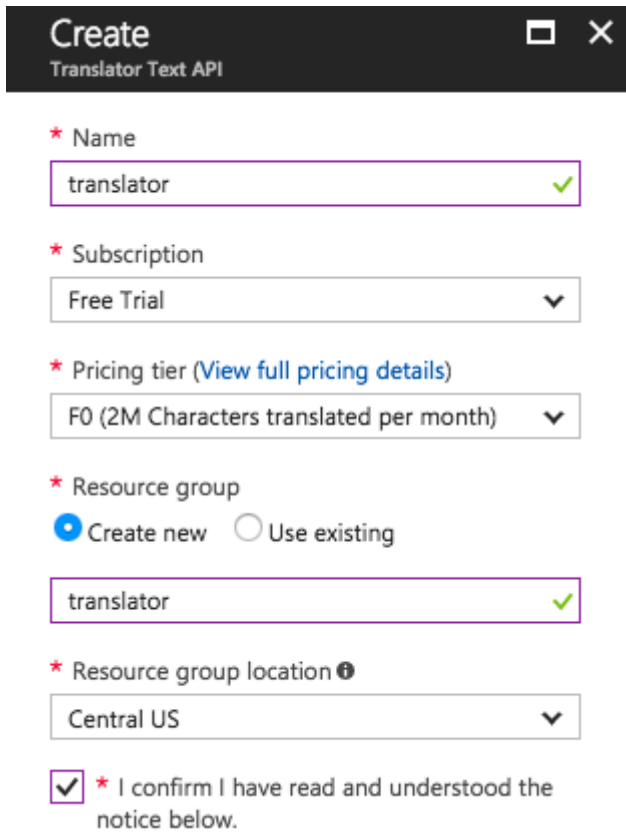
Note that I have not associated an action with this link yet. First I want to figure out how to carry out the actual translations.

Using a Third-Party Translation Service

The two major translation services are Google Cloud Translation API (<https://developers.google.com/translate/>) and Microsoft Translator Text API (<https://www.microsoft.com/en-us/translator/business/>). Both are paid services, but the Microsoft offering has an entry level option for low volume of translations that is free. Google offered a free translation service in the past but today, even the lowest service tier is paid. Because I want to be able to experiment with translations without incurring in expenses, I'm going to implement the Microsoft solution.

Before you can use the Microsoft Translator API, you will need to get an account with Azure (<https://azure.com>), Microsoft's cloud service. You can select the free tier, while you will be asked to provide a credit card number during the signup process, your card is not going to be charged while you stay on that level of service.

Once you have the Azure account, go to the Azure Portal and click on the "New" button on the top left, and then type or select the "Translator Text API". When you click the "Create" button, you will be presented with a form in which you define a new translator resource that will be added to your account. You can see below how I completed the form:



Create
Translator Text API

* Name
translator ✓

* Subscription
Free Trial ▼

* Pricing tier (View full pricing details)
F0 (2M Characters translated per month) ▼

* Resource group
☒ Create new ☐ Use existing
translator ✓

* Resource group location ⓘ
Central US ▼

☒ * I confirm I have read and understood the notice below.

When you click the "Create" button once again, the translator API resource will be added to your account. If you wait a few seconds, you will receive a notification in the top bar that the translator resource was deployed. Click the "Go to resource" button in the notification and then on the "Keys" option on the left sidebar. You will now see two keys, labeled "Key 1" and "Key 2". Copy either one of the keys to the clipboard and then enter it into an environment variable in your terminal (if you are using Microsoft Windows, replace `export` with `set`):

```
(venv) $ export MS_TRANSLATOR_KEY=<paste-your-key-here>
```

This key is used to authenticate with the translation service, so it needs to be added to the application configuration:

config.py: Add Microsoft Translator API key to the configuration.

```
class Config(object):  
    # ...  
    MS_TRANSLATOR_KEY = os.environ.get('MS_TRANSLATOR_KEY')
```

As always with configuration values, I prefer to install them in environment variables and import them into the Flask configuration from there. This is particularly important with sensitive information such as keys or passwords that enable access to third-party services. You definitely do not want to write those explicitly in the code.

The Microsoft Translator API is a web service that accepts HTTP requests. There are a few HTTP clients in Python, but the most popular and simple to use is the `requests` package. So let's install that into the virtual environment:

```
(venv) $ pip install requests
```

Below you can see the function that I coded to translate text using the Microsoft Translator API. I am putting in a new *app/translate.py* module:

app/translate.py: Text translation function.

```
import json
import requests
from flask_babel import _
from app import app

def translate(text, source_language, dest_language):
    if 'MS_TRANSLATOR_KEY' not in app.config or \
        not app.config['MS_TRANSLATOR_KEY']:
        return _('Error: the translation service is not configured.')
    auth = {'Ocp-Apim-Subscription-Key': app.config['MS_TRANSLATOR_KEY']}
    r = requests.get('https://api.microsofttranslator.com/v2/Ajax.svc'
                     '/Translate?text={}&from={}&to={}'.format(
                         text, source_language, dest_language),
                     headers=auth)
    if r.status_code != 200:
        return _('Error: the translation service failed.')
    return json.loads(r.content.decode('utf-8-sig'))
```

The function takes the text to translate and the source and destination language codes as arguments, and it returns a string with the translated text. It starts by checking that there is a key for the translation service in the configuration, and if it isn't there it returns an error. The error is also a string, so from the outside, this is going to look like the translated text. This ensures that in the case of an error, the user will see a meaningful error message.

The `get()` method from the `requests` package sends an HTTP request with a `GET` method to the URL given as the first argument. I'm using the `/v2/Ajax.svc/Translate` URL, which is an endpoint from the translation service that returns translations as a JSON payload. The text, source and destination languages need to be given as query string arguments in the URL, named `text`, `from` and `to` respectively. To authenticate with the service, I need to pass the key that I added to the configuration. This key needs to be given in a custom HTTP header with the name `Ocp-Apim-Subscription-Key`. I created the `auth` dictionary with this header and then pass it to `requests` in the `headers` argument.

The `requests.get()` method returns a response object, which contains all the details provided by the service. I first need to check that the status code is 200, which is the code for a successful request. If I get any other codes, I know that there was an error, so in that case I return an error string. If the status code is 200, then the body of the response has a JSON encoded string with the translation, so all I need to do is use the `json.loads()` function from the Python standard library to decode the JSON into a Python string that I can use. The `content` attribute of the response object contains the raw body of the response as a bytes object, which is converted to a UTF-8 string and sent to `json.loads()`.

Below you can see a Python console session in which I use the new `translate()` function:

```
>>> from app.translate import translate
>>> translate('Hi, how are you today?', 'en', 'es') # English to Spanish
'Hola, ¿cómo estás hoy?'
>>> translate('Hi, how are you today?', 'en', 'de') # English to German
'Are Hallo, how you heute?'
>>> translate('Hi, how are you today?', 'en', 'it') # English to Italian
'Ciao, come stai oggi?'
>>> translate('Hi, how are you today?', 'en', 'fr') # English to French
'Salut, comment allez-vous aujourd'hui ?'
```

Pretty cool, right? Now it's time to integrate this functionality with the application.

Ajax From The Server

I'm going to start by implementing the server-side part. When the user clicks the Translate link that appears below a post, an asynchronous HTTP request will be issued to the server. I'll show you how to do this in the next session, so for now I'm going to concentrate on implementing the handling of this request by the server.

An asynchronous (or Ajax) request is similar to the routes and view functions that I have created in the application, with the only difference that instead of returning HTML or a redirect, it just returns data, formatted as XML (<http://en.wikipedia.org/wiki/XML>) or more commonly JSON (<http://en.wikipedia.org/wiki/JSON>). Below you can see the translation view function, which invokes the Microsoft Translator API and then returns the translated text in JSON format:

app/routes.py: Text translation view function.

```
from flask import jsonify
from app.translate import translate

@app.route('/translate', methods=['POST'])
@login_required
def translate_text():
    return jsonify({'text': translate(request.form['text'],
                                     request.form['source_language'],
                                     request.form['dest_language'])})
```

As you can see, this is simple. I implemented this route as a `POST` request. There is really no absolute rule as to when to use `GET` or `POST` (or other request methods that you haven't seen yet). Since the client will be sending data, I decided to use a `POST` request, as that is similar to the requests that submit form data. The `request.form` attribute is a dictionary that Flask exposes with all the data that has included in the submission. When I worked with web forms, I did not need to look at `request.form` because Flask-WTF does all that work for me, but in this case, there is really no web form, so I have to access the data directly.

So what I'm doing in this function is to invoke the `translate()` function from the previous section passing the three arguments directly from the data that was submitted with the request. The result is incorporated into a single-key dictionary, under the key `text`, and the dictionary is passed as an argument to Flask's `jsonify()` function, which converts the dictionary to a JSON formatted payload. The return value from `jsonify()` is the HTTP response that is going to be sent back to the client.

For example, if the client wanted to translate the string `Hello, World!` to Spanish, the response from this request would have the follow payload:

```
{ "text": "Hola, Mundo!" }
```

Ajax From The Client

So now that the server is able to provide translations through the `/translate` URL, I need to invoke this URL when the user clicks the "Translate" link I added above, passing the text to translate and the source and destination languages. If you are not familiar with working with JavaScript in the browser this is going to be a good learning experience.

When working with JavaScript in the browser, the page currently being displayed is internally represented in as the Document Object Model or just the DOM. This is a hierarchical structure that references all the elements that exist in the page. The

JavaScript code running in this context can make changes to the DOM to trigger changes in the page.

Let's first discuss how my JavaScript code running in the browser can obtain the three arguments that I need to send to the translate function that runs in the server. To obtain the text, I need to locate the node within the DOM that contains the blog post body and read its contents. To make it easy to identify the DOM nodes that contain blog posts, I'm going to attach a unique ID to them. If you look at the `_post.html` template, the line that renders the post body just reads `{{ post.body }}`. What I'm going to do is wrap this content in a `` element. This is not going to change anything visually, but it gives me a place where I can insert an identifier:

app/templates/_post.html: Add an ID to each blog post.

```
<span id="post{{ post.id }}">{{ post.body }}</span>
```

This is going to assign a unique identifier to each blog post, with the format `post1`, `post2`, and so on, where the number matches the database identifier of each post. Now that each blog post has a unique identifier, given a ID value I can use jQuery to locate the `` element for that post and extract the text in it. For example, if I wanted to get the text for a post with ID 123 this is what I would do:

```
$('#post123').text()
```

Here the `$` sign is the name of a function provided by the jQuery library. This library is used by Bootstrap, so it was already included by Flask-Bootstrap. The `#` is part of the "selector" syntax used by jQuery, which means that what follows is the ID of an element.

I will also want to have a place where I will be inserting the translated text once I receive it from the server. What I'm going to do, is replace the "Translate" link with the translated text, so I also need to have a unique identifier for that node:

app/templates/_post.html: Add an ID to the translate link.

```
<span id="translation{{ post.id }}">
  <a href="#">{{ _('Translate') }}</a>
</span>
```

So now for a given post ID, I have a `post<ID>` node for the blog post, and a corresponding `translation<ID>` node where I will need to replace the Translate link with the translated text once I have it.

The next step is to write a function that can do all the translation work. This function will take the input and output DOM nodes, and the source and destination languages, issue the asynchronous request to the server with the three arguments needed, and finally

replace the Translate link with the translated text once the server responds. This sounds like a lot of work, but the implementation is fairly simple:

app/templates/base.html: Client-side translate function.

```
{% block scripts %}
...
<script>
    function translate(sourceElem, destElem, sourceLang, destLang) {
        $(destElem).html('');
        $.post('/translate', {
            text: $(sourceElem).text(),
            source_language: sourceLang,
            dest_language: destLang
        }).done(function(response) {
            $(destElem).text(response['text'])
        }).fail(function() {
            $(destElem).text("{{ _('Error: Could not contact server.') }}" );
        });
    }
</script>
{% endblock %}
```

The first two arguments are the unique IDs for the post and the translate link nodes. The last two argument are the source and destination language codes.

The function begins with a nice touch: it adds a *spinner* replacing the Translate link so that the user knows that the translation is in progress. This is done with jQuery, using the `$(destElem).html()` function to replace the original HTML that defined the translate link with new HTML content based on the `` link. For the spinner, I'm going to use a small animated GIF that I have added to the *app/static/loading.gif* directory, which Flask reserves for static files. To generate the URL that references this image, I'm using the `url_for()` function, passing the special route name `static` and giving the filename of the image as an argument. You can find the *loading.gif* image in the download package (<https://github.com/miguelgrinberg/microblog/tree/v0.14>) for this chapter.

So now I have a nice spinner that took the place of the Translate link, so the user knows to wait for the translation to appear. The next step is to send the `POST` request to the */translate* URL that I defined in the previous section. For this I'm also going to use jQuery, in this case the `$.post()` function. This function submits data to the server in a format that is similar to how the browser submits a web form, which is convenient because that allows Flask to incorporate this data into the `request.form` dictionary. The arguments to `$.post()` are two, first the URL to send the request to, and then a dictionary (or object, as these are called in JavaScript) with the three data items the server expects.

You probably know that JavaScript works a lot with callback functions, or a more advanced form of callbacks called *promises*. What I want to do now is indicate what I want done once this request completes and the browser receives the response. In JavaScript there is no such thing as waiting for something, everything is *asynchronous*. What I need to do instead is to provide a callback function that the browser will invoke when the response is received. And also as a way to make everything as robust as possible, I want to indicate what to do in the case an error has occurred, so that would be a second callback function to handle errors. There are a few ways to specify these callbacks, but for this case, using promises makes the code fairly clear. The syntax is as follows:

```
$.post(<url>, <data>).done(function(response) {  
    // success callback  
}).fail(function() {  
    // error callback  
})
```

The promise syntax allows you to basically "chain" the callbacks to the return value of the `$.post()` call. In the success callback, all I need to do is call `$(destElem).text()` with the translated text, which comes in a dictionary under the `text` key. In the case of an error, I do the same, but the text that I display is a generic error message, which I make sure is entered in the base template as a translatable text.

So now the only thing that is left is to trigger the `translate()` function with the correct arguments as a result of the user clicking a Translate link. There are also a few ways to do this, what I'm going to do is just embed the call to the function in the `href` attribute of the link:

app/templates/_post.html: Translate link handler.

```
<span id="translation{{ post.id }}">  
  <a href="javascript:translate(  
    '#post{{ post.id }}',  
    '#translation{{ post.id }}',  
    '{{ post.language }}',  
    '{{ g.locale }}';">{{ _('Translate') }}</a>  
</span>
```

The `href` element of a link can accept any JavaScript code if it is prefixed with `javascript:`, so that is a convenient way to make the call to the translation function. Because this link is going to be rendered in the server when the client requests the page, I can use `{{ }}` expressions to generate the four arguments to the function. Each post will have its own translate link, with its uniquely generated arguments. The `#` that you see as a prefix to the `post<ID>` and `translation<ID>` elements indicates that what follows is an element ID.

Now the live translation feature is complete! If you have set a valid Microsoft Translator API key in your environment, you should now be able to trigger translations. Assuming you have your browser set to prefer English, you will need to write a post in another language to see the "Translate" link. Below you can see an example:



In this chapter I introduced a few new texts that need to be translated into all the languages supported by the application, so it is necessary to update the translation catalogs:

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

For your own projects you will then need to edit the *messages.po* files in each language repository to include the translations for these new tests, but I have already created the Spanish translations in the download package for this chapter or the GitHub repository.

To publish the new translations, they need to be compiled:

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

Hello, and thank you for visiting my blog! If you enjoyed this article, please consider supporting my work on this blog on Patreon (<https://patreon.com/miguelgrinberg>)!

**BECOME A PATRON***(<https://patreon.com/miguelgrinberg>)*

Tweet

Like



Share

© 2012-2020 by Miguel Grinberg. All rights reserved. Questions? (mailto:webmaster_at_miguelgrinberg_dot_com)