A Django custom middleware story

by Laura Barluzzi

Background: users and preferred languages

During the first two weeks of my internship with the open-source non-profit Cadasta.org, I added the option for users of the Cadasta platform to select a "preferred language". Once a user sets the preferred language, the platform should be displayed to that user in this language.

In order to implement this functionality, I had to:

1. Add a language field into the User model in Django (backend)
2. Add a dropdown field in the user interface enabling editing of the language field (frontend)
3. Activate the translation to the user's language whenever they log in or edit their profile (logic)

While the first two steps were extremely easy to achieve thanks to Django models [1] and Django form models [2], the implementation of the third task was surprisingly non-obvious. Django makes it simple to localize a text to a given language via gettext [3] and to activate a default translation language in a template via set_language [4], however, Django does not provide a built-in way to activate a translation language based on a value stored in a model… a problem that many developers are facing as shown by numerous questions on StackOverflow [5]. I realized I had to create my own *custom* solution.

Solution: Django custom middlewares

I wanted to write the code to activate the language in one single place, to reduce redundancy. In order to do so, the best option would have been to activate the translation whenever there was a request. This brought me to think about adding a "middleware".

According to Wikipedia [6], a middleware is defined as:

[...] computer software that provides services to software applications beyond those available from the operating system. It can be described as "software glue." [...]

Specifically, in the Django context, Vitor Freitas [7] defines a middleware as:

[...] a regular Python class that hooks into Django's request/response life cycle [...]

In the Cadasta context, we can visualize a middleware with the following diagram:

```
User → request → middleware → Cadasta Django application
User ← response ← middleware ← Cadasta Django application
```

As previously stated, Django does not come with a middleware for the User-specific localization use-case, so I had to build a *custom middleware*.

Django provides some instructions on how to create Django custom middlewares [8]. However, the official documentation is highly misleading since in their main example they don't clearly mention that if you don't return a response from a middleware you should return None. Furthermore, the official documentation doesn't provide guidance on where to locate the middleware in the code. Nevertheless, with the help of some examples on StackOverflow [9] and this How to Create a Custom Django Middleware [10] tutorial, I managed to successfully implement my custom middleware:

```
from django.utils import translation

class UserLanguageMiddleware(object):
        def process_response(self, request, response):
        user = getattr(request, 'user', None)
        if not user:
                return response

        if not user.is_authenticated:
                return response

        user_language = getattr(user, 'language', None)
```

```
            if not user_language:
                return response

            current_language = translation.get_language()
            if user_language == current_language:
                return response

            translation.activate(user_language)
            request.session[translation.LANGUAGE_SESSION_KEY] = user_language

        return response
```

This middleware class is going to be called for every request. It augments the request processing by accessing the user's preferred language (stored in the user.language field) and by activating the translation of the entire platform. Furthermore, this code also sets the LANGUAGE_SESSION_KEY session variable [11] to the preferred language to ensure that the correct language is used in all localization code-paths in the application.

To integrate this middleware with the Django application, I had to add the fully qualified name of the middleware to the MIDDLEWARE_CLASSES list in settings/default.py:

```
MIDDLEWARE_CLASSES = (
    'django.contrib.sessions.middleware.SessionMiddleware',
    # ...
    accounts.middleware.UserLanguageMiddleware',
)
```

Conclusion: clean code thanks to Django custom middlewares

Implementing the preferred language feature via a custom middleware worked very smoothly and required the addition of just 1 file (with respective test) which encapsulates the code and reduces the scope of the change. My custom middleware has been merged into the master branch [12] of cadasta-platform and it is published as a package on PyPI [13] to be be re-used for any Django application that implements per-user localization.

References

[1] https://docs.djangoproject.com/en/1.11/topics/db/models/

[2] https://docs.djangoproject.com/en/1.11/topics/forms/modelforms/

[3] https://docs.djangoproject.com/en/1.11/topics/i18n/translation/#standard-translation

[4] https://docs.djangoproject.com/en/1.11/topics/i18n/translation/#set-language-redirect-view

[5] https://stackoverflow.com/q/45109610/3817588

[6] https://en.wikipedia.org/wiki/Middleware

[7] https://simpleisbetterthancomplex.com/page17/

[8] https://docs.djangoproject.com/en/1.11/topics/http/middleware/

[9] https://stackoverflow.com/a/32242074/3817588

[10] https://simpleisbetterthancomplex.com/tutorial/2016/07/18/how-to-create-a-custom-django-middleware.html

[11] https://docs.djangoproject.com/en/2.0/ref/utils/#django.utils.translation.LANGUAGE_SESSION_KEY

[12] https://github.com/Cadasta/cadasta-platform/blob/9d03c29/cadasta/accounts/middleware.py

[13] https://pypi.python.org/pypi/django-user-language-middleware/