The Flask Mega-Tutorial Part XII: Dates and Times (/post/the-flask-mega-tutorial-part-xii-dates-and-times)

February 20 2018

Posted by Miguel Grinberg (/author/Miguel Grinberg) under Python (/category/Python), Programming (/category/Programming), Flask (/category/Flask).

This is the twelfth installment of the Flask Mega-Tutorial series, in which I'm going to tell you how to work with dates and times in a way that works for all your users, regardless of where they reside.

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) (this article)
- 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)
- 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).

One of the aspects of my Microblog application that I have ignored for a long time is the display of dates and times. Until now, I've just let Python render the `datetime` object in the `User` model, and have completely ignored the one in the `Post` model.

The GitHub links for this chapter are: Browse (https://github.com/miguelgrinberg/microblog/tree/v0.12), Zip (https://github.com/miguelgrinberg/microblog/archive/v0.12.zip), Diff (https://github.com/miguelgrinberg/microblog/compare/v0.11...v0.12).

Timezone Hell

Using Python on the server to render dates and times that are presented to users on their web browsers is really not a good idea. Consider the following example. I'm writing this at 4:06PM on September 28th, 2017. My timezone at the time I'm writing this is PDT (or UTC-7 if you prefer). Running in a Python interpreter I get the following:

```
>>> from datetime import datetime
>>> str(datetime.now())
'2017-09-28 16:06:30.439388'
>>> str(datetime.utcnow())
'2017-09-28 23:06:51.406499'
```

The `datetime.now()` call returns the correct time for my location, while the `datetime.utcnow()` call returns the time in the UTC time zone. If I could ask many people living in different parts of the world to run the above code all at that same time with

me, the `datetime.now()` function will return different results for each person, but `datetime.utcnow()` will always return the same time, regardless of location. So which one do you think is better to use in a web application that will very likely have users located all over the world?

It is pretty clear that the server must manage times that are consistent and independent of location. If this application grows to the point of needing several production servers in different regions around the world, I would not want each server to write timestamps to the database in different timezones, because that would make working with these times impossible. Since UTC is the most used uniform timezone and is supported in the `datetime` class, that is what I'm going to use.

But there is an important problem with this approach. For users in different timezones, it will be awfully difficult to figure out when a post was made if they see times in the UTC timezone. They would need to know in advance that the times are in UTC so that they can mentally adjust the times to their own timezone. Imagine a user in the PDT timezone that posts something at 3:00pm, and immediately sees that the post appears with a 10:00pm UTC time, or to be more exact 22:00. That is going to be very confusing.

While standardizing the timestamps to UTC makes a lot of sense from the server's perspective, this creates a usability problem for users. The goal of this chapter is to address this problem while keeping all the timestamps managed in the server in UTC.

Timezone Conversions

The obvious solution to the problem is to convert all timestamps from the stored UTC units to the local time of each user. This allows the server to continue using UTC for consistency, while an on-the-fly conversion tailored to each user solves the usability problem. The tricky part of this solution is to know the location of each user.

Many websites have a configuration page where users can specify their timezone. This would require me to add a new page with a form in which I present users with a dropdown with the list of timezones. Users can be asked to enter their timezone when they access the site for the first time, as part of their registration.

While this is a decent solution that solves the problem, it is a bit odd to ask users to enter a piece of information that they have already configured in their operating system. It seems it would be more efficient if I could just grab the timezone setting from their computers.

As it turns out, the web browser knows the user's timezone, and exposes it through the standard date and time JavaScript APIs. There are actually two ways to take advantage of the timezone information available via JavaScript:

- The "old school" approach would be to have the web browser somehow send the timezone information to the server when the user first logs on to the application. This could be done with an Ajax (http://en.wikipedia.org/wiki/Ajax_(programming)) call, or much more simply with a meta refresh tag (http://en.wikipedia.org/wiki/Meta_refresh). Once the server knows the timezone it can keep it in the user's session or write it to the user's entry in the database, and from then on adjust all timestamps with it at the time templates are rendered.
- The "new school" approach would be to not change a thing in the server, and let the conversion from UTC to local timezone happen in the client, using JavaScript.

Both options are valid, but the second one has a big advantage. Knowing the timezone of the user isn't always enough to present dates and times in the format expected by the user. The browser has also access to the system locale configuration, which specifies things like AM/PM vs. 24 hour clock, DD/MM/YYYY vs. MM/DD/YYYY and many other cultural or regional styles.

And if that isn't enough, there is yet one more advantage for the new school approach. There is an open-source library that does all this work!

Introducing Moment.js and Flask-Moment

Moment.js (http://momentjs.com) is a small open-source JavaScript library that takes date and time rendering to another level, as it provides every imaginable formatting option, and then some. And a while ago I created Flask-Moment, a small Flask extension that makes it very easy to incorporate moment.js into your application.

So let's start by installing Flask-Moment:

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

This extension is added to a Flask application in the usual way:

app/__init__.py: Flask-Moment instance.

```
# ...
from flask_moment import Moment

app = Flask(__name__)
# ...
moment = Moment(app)
```

Unlike other extensions, Flask-Moment works together with *moment.js*, so all templates of the application must include this library. To ensure that this library is always available, I'm going to add it in the base template. This can be done in two ways. The most direct way is to explicitly add a `<script>` tag that imports the library, but Flask-Moment makes it easier, by exposing a `moment.include_moment()` function that generates the `<script>` tag:

app/templates/base.html: Including moment.js in the base template.

```
...

{% block scripts %}
    {{ super() }}
    {{ moment.include_moment() }}
{% endblock %}
```

The `scripts` block that I added here is another block exported by Flask-Bootstrap's base template. This is the place where JavaScript imports are to be included. This block is different from previous ones in that it already comes with some content defined in the base template. All I want to do is add the moment.js library, without losing the base contents. And this is achieved with the `super()` statement, which preserves the content from the base template. If you define a block in your template without using `super()`, then any content defined for this block in the base template will be lost.

Using Moment.js

Moment.js makes a `moment` class available to the browser. The first step to render a timestamp is to create an object of this class, passing the desired timestamp in ISO 8601 (http://en.wikipedia.org/wiki/ISO_8601) format. Here is an example:

```
t = moment('2017-09-28T21:45:23Z')
```

If you are not familiar with the ISO 8601 standard format for dates and times, the format is as follows: `{{ year }}-{{ month }}-{{ day }}T{{ hour }}:{{ minute }}:{{ second }} {{ timezone }}`. I already decided that I was only going to work with UTC timezones, so the last part is always going to be `Z`, which represents UTC in the ISO 8601 standard.

The `moment` object provides several methods for different rendering options. Below are some of the most common options:

```
moment('2017-09-28T21:45:23Z').format('L')
"09/28/2017"
moment('2017-09-28T21:45:23Z').format('LL')
"September 28, 2017"
moment('2017-09-28T21:45:23Z').format('LLL')
"September 28, 2017 2:45 PM"
moment('2017-09-28T21:45:23Z').format('LLLL')
"Thursday, September 28, 2017 2:45 PM"
moment('2017-09-28T21:45:23Z').format('dddd')
"Thursday"
moment('2017-09-28T21:45:23Z').fromNow()
"7 hours ago"
moment('2017-09-28T21:45:23Z').calendar()
"Today at 2:45 PM"
```

This example creates a moment object initialized to September 28th 2017 at 9:45pm UTC. You can see that all the options I tried above are rendered in UTC-7, which is the timezone configured on my computer. You can enter the above commands in your browser's console, making sure the page on which you open the console has moment.js included. You can do it in microblog, as long as you made the changes above to include moment.js, or also on *https://momentjs.com/*.

Note how the different methods create different representations. With `format()` you control the format of the output with a format string, similar to the strftime (https://docs.python.org/3.6/library/time.html#time.strftime) function from Python. The `fromNow()` and `calendar()` methods are interesting because they render the timestamp in relation to the current time, so you get output such as "a minute ago" or "in two hours", etc.

If you were working directly in JavaScript, the above calls return a string that has the rendered timestamp. Then it is up to you to insert this text in the proper place on the page, which unfortunately requires some JavaScript to work with the DOM (https://en.wikipedia.org/wiki/Document_Object_Model). The Flask-Moment extension greatly simplifies the use of moment.js by enabling a `moment` object similar to the JavaScript one in your templates.

Let's look at the timestamp that appears in the profile page. The current *user.html* template lets Python generate a string representation of the time. I can now render this timestamp using Flask-Moment as follows:

app/templates/user.html: Render timestamp with moment.js.

```
            {% if user.last_seen %}
            <p>Last seen on: {{ moment(user.last_seen).format('LLL') }}</p>
            {% endif %}
```

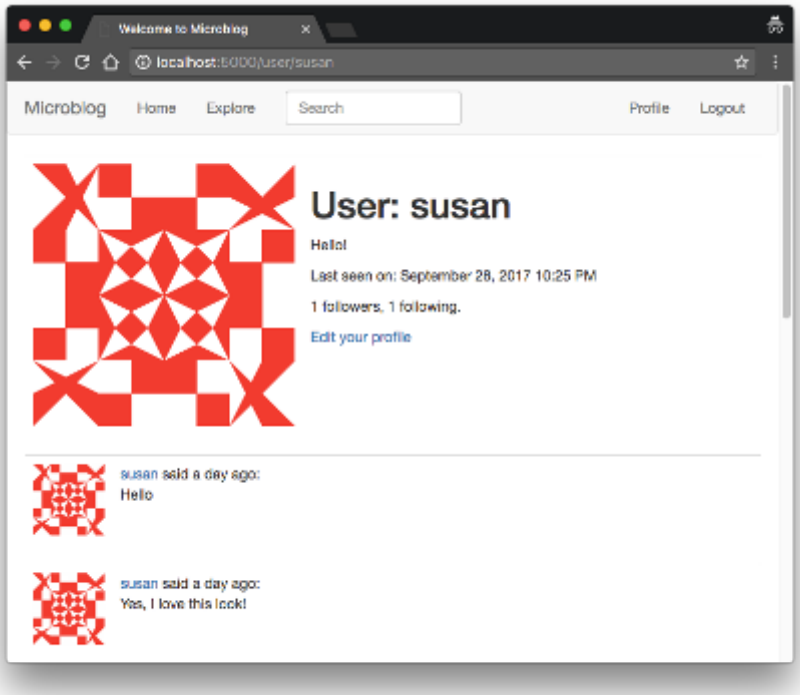
So as you can see, Flask-Moment uses a syntax that is similar to that of the JavaScript library, with one difference being that the argument to `moment()` is now a Python `datetime` object and not an ISO 8601 string. The `moment()` call issued from a template also automatically generates the required JavaScript code to insert the rendered timestamp in the proper place of the DOM.

The second place where I can take advantage of Flask-Moment and moment.js is in the *_post.html* sub-template, which is invoked from the index and user pages. In the current version of the template, each post preceded with a "username says:" line. Now I can add a timestamp rendered with `fromNow()`:

app/templates/_post.html: Render timestamp in post sub-template.

```
            <a href="{{ url_for('user', username=post.author.username) }}">
                {{ post.author.username }}
            </a>
            said {{ moment(post.timestamp).fromNow() }}:
            <br>
            {{ post.body }}
```

Below you can see how both these timestamps look when rendered with Flask-Moment and moment.js:



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)!

(https://patreon.com/miguelgrinberg)