# The Flask Mega-Tutorial, Part XX: Some JavaScript Magic

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

This is the twentieth installment of the Flask Mega-Tutorial series, in which I'm going to add a nice popup when you hover your mouse over a user's nickname.

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!

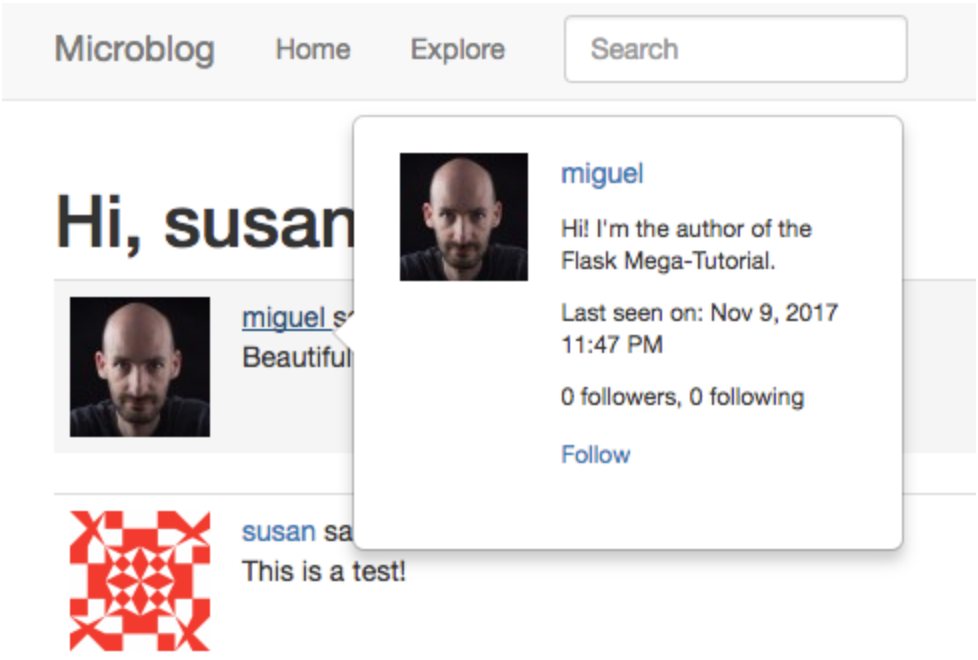
If you are looking for the 2018 edition of this course, you can find it here.

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

- Chapter 21: User Notifications
- Chapter 22: Background Jobs
- Chapter 23: Application Programming Interfaces (APIs)

Nowadays, it is impossible to build a web application that doesn't use at least a bit of JavaScript. As I'm sure you know, the reason is that JavaScript is the only language that runs natively in web browsers. In Chapter 14 you saw me add a simple JavaScript enabled link in a Flask template to provide real-time language translations of blog posts. In this chapter I'm going to dig deeper into the topic and show you another useful JavaScript trick to make the application more interesting and engaging to users.

A common user interface pattern for social sites in which users can interact with each other is to show a quick summary of a user in a popup panel when you hover over the user's name, anywhere it appears on the page. If you have never paid attention to this, go to Twitter, Facebook, LinkedIn, or any other major social network, and when you see a username, just leave your mouse pointer on top of it for a couple of seconds to see the popup appear. This chapter is going to be dedicated to building that feature for Microblog, of which you can see a preview below:




*The GitHub links for this chapter are: Browse, Zip, Diff.*

## Server-side Support

Before we delve into the client-side, let's get the server work that is necessary to support these user popups out of the way. The contents of the user popup are going to be returned by a new route, which is going to be a simplified version of the existing user profile route. Here is the view function:

*app/main/routes.py*: User popup view function.

```
@bp.route('/user/<username>/popup')
@login_required
def user_popup(username):
    user = db.first_or_404(sa.select(User).where(User.username == user
    form = EmptyForm()
    return render_template('user_popup.html', user=user, form=form)
```

This route is going to be attached to the */user/<username>/popup* URL, and will simply load the requested user and then render a template with it. The template is a shorter version of the one used for the user profile page:

*app/templates/user_popup.html*: User popup template.

```
<div>
  <img src="{{ user.avatar(64) }}" style="margin: 5px; float: left">
  <p><a href="{{ url_for('main.user', username=user.username) }}">{{ u
  {% if user.about_me %}<p>{{ user.about_me }}</p>{% endif %}
  <div class="clearfix"></div>
  {% if user.last_seen %}
  <p>{{ _('Last seen on') }}: {{ moment(user.last_seen).format('lll')
  {% endif %}
  <p>{{ _('%(count)d followers', count=user.followers_count()) }}, {{
  {% if user != current_user %}
    {% if not current_user.is_following(user) %}
    <p>
      <form action="{{ url_for('main.follow', username=user.username)
        {{ form.hidden_tag() }}
        {{ form.submit(value=_('Follow'), class_='btn btn-outline-prim
      </form>
    </p>
    {% else %}
    <p>
      <form action="{{ url_for('main.unfollow', username=user.username
        {{ form.hidden_tag() }}
        {{ form.submit(value=_('Unfollow'), class_='btn btn-outline-pr
      </form>
    </p>
    {% endif %}
```

```
  {% endif %}
</div>
```

The popover components that I will add in the following sections will invoke this route when the user hovers the mouse pointer over a username. In response the server will return the HTML content for the popup, which the client will display. When the user moves the mouse away the popup will be removed.

## Introduction to the Bootstrap Popover Component

In Chapter 11 I introduced you to the Bootstrap framework as a convenient way to create great looking web pages. So far, I have used only a minimal portion of this framework. Bootstrap comes bundled with many common UI elements, all of which have demos and examples in the Bootstrap documentation at *https://getbootstrap.com*. One of these components is the Popover, which is described in the documentation as a "small overlay of content, for housing secondary information". Exactly what I need!

Most Bootstrap components are defined through HTML markup that references the Bootstrap CSS definitions that add the nice styling. Some of the most advanced ones also require JavaScript. The standard way in which an application includes these components in a web page is by adding the HTML in the proper place, and then for the components that need scripting support, calling a JavaScript function that initializes it or activates it. The popover component does require JavaScript support.

To begin, I just need to decide which elements in the page are going to trigger popovers to appear. For this I'm going to use the clickable usernames that appear in each post. The *app/templates/_post.html* sub-template has the username already defined:

```
            <a href="{{ url_for('main.user', username=post.author.user
                {{ post.author.username }}
            </a>
```

Now according to the popover documentation, I need to create an object of class `bootstrap.Popover` for each of the links like the one above that appear on the page, and this will initialize the popovers. Unfortunately,

after reading this I ended up with more questions than answers, because this component does not appear to be designed to work in the way I need it to. The following is a list of problems I need to solve to implement this feature:

- There will be many username links in the page, one for each blog post displayed. I need to have a way to find all these links so that I can then initialize them as popovers in a JavaScript function that runs before the user has a chance to interact with the page.
- The popover examples in the Bootstrap documentation all provide the content of the popover as a `data-bs-content` attribute added to the target HTML element. That is really inconvenient for me, because I want to make an Ajax call to the server to get the content to display in the popover.
- When using the "hover" mode, the popup will stay visible for as long as you keep the mouse pointer within the target element. When you move the mouse away, the popup will go away. This has the ugly side effect that if the user wants to move the mouse pointer into the popup itself, the popup will disappear. I will need to figure out a way to extend the hover behavior to also include the popup, so that the user can move into the popup and, for example, click on a link there.

It is actually not that uncommon when working with browser based applications that things get complicated really fast. You have to think very specifically in terms of how the DOM elements interact with each other and make them behave in a way that gives the user a good experience. In the sections that follow I will be looking at the above problems one by one.

## Executing a Function On Page Load

The popover component needs to be explicitly initialized with JavaScript, so it is clear that I'm going to need to run some code as soon as each page loads that will search for all the links to usernames in the page, and initialize popover components from Bootstrap on them.

The standard way in modern browsers to run initialization code after the page finished loading is to define a handler for the `DOMContentLoaded` event. I can add this handler in the *app/templates/base.html* template, so that this runs on every page of the application:

*app/templates/base.html*: Run function after page load.

```
...
<script>
    // ...

    function initialize_popovers() {
      // write initialization code here
    }
    document.addEventListener('DOMContentLoaded', initialize_popovers)
</script>
```

As you see, I have added my initialization function inside the `<script>` element in which I defined the `translate()` function in Chapter 14.

## Finding DOM Elements with Selectors

My next problem is to write the JavaScript logic that finds all the user links in the page.

If you recall from Chapter 14, the HTML elements that were involved in the live translations had unique IDs. For example, a post with ID=123 had a `id="post123"` attribute added. Then I could use the `document.getElementById()` function to locate this element in the DOM. This function is part of a group that allow applications running in the browser to find elements based on their characteristics.

For the translation feature I had to find a specific element that had an `id` attribute, which uniquely identifies elements on the page. Another search option that is more appropriate when looking for groups of elements is to add a CSS `class` to them. Unlike `id`, the `class` attribute can be assigned to multiple elements, so it is ideal for finding all the links that need popovers. What I'm going to do is mark all the user links with a `class="user_popup"` attribute, and then I will get the list of links from JavaScript with `document.getElementsByClassName('user_popup')`. The return value in this case would be a collection of all the elements that have the class.

*app/templates/base.html*: Run function after page load.

```
...
<script>
    // ...
```

```
    function initialize_popovers() {
      const popups = document.getElementsByClassName('user_popup');
      for (let i = 0; i < popups.length; i++) {
        // create popover here
      }
    }
    document.addEventListener('DOMContentLoaded', initialize_popovers)
</script>
```

## Popovers and the DOM

In the previous section I've added initialization code that looks for all the elements in the page that have been assigned the class `user_popup`. This class needs to be added to the username links, which are defined in the *_post.html* template page.

*app/templates/_post.html*: User popup template.

```
...
        {% set user_link %}
            <a class="user_popup"
            href="{{ url_for('main.user', username=post.author.usernam
                {{ post.author.username }}
            </a>
        {% endset %}
...
```

If you are wondering where the popover HTML elements are defined, the good news is that I don't have to worry about that. When I create the `Popover` object the Bootstrap framework will dynamically insert the elements associated with the popup for me.

## Creating the Popover Components

Now I'm ready to create the Popover components for all the username links found on the page.

*app/templates/base.html*: Hover delay.

```
      function initialize_popovers() {
        const popups = document.getElementsByClassName('user_popup');
        for (let i = 0; i < popups.length; i++) {
          const popover = new bootstrap.Popover(popups[i], {
            content: 'Loading...',
```

```
            trigger: 'hover focus',
            placement: 'right',
            html: true,
            sanitize: false,
            delay: {show: 500, hide: 0},
            container: popups[i],
            customClass: 'd-inline',
          });
        }
      }
      document.addEventListener('DOMContentLoaded', initialize_popover
```

The `bootstrap.Popover` constructor takes the element that is getting a popover as first argument, and an object with options as second argument. Options include the content that will appear in the popup, what method to use to trigger the popup to appear or disappear (a click, hovering over the element, etc.), the placement of the popover, if the content is plain text or HTML, and a few more options.

I mentioned above that a big problem with this popover implementation is that the HTML contents that need to be displayed are obtained by making a request to the server. For that reason, I initialize the content with a `Loading...` text, which is going to be dynamically replaced once the HTML contents of the user are received from the server. In preparation for this, I set the `html` option to `true`, and also disable an option to sanitize the HTML content. Sanitizing HTML is a security feature that is very important to use when the content comes from users. In this use case, the HTML content is generated by the Flask server through a Jinja template, which sanitizes all dynamic contents by default.

The `delay` option configures the popover component to appear after half a second of hovering. No delay is configured when the popover is removed as a result of the user moving the mouse pointer away. The `container` option tells Bootstrap to insert the popover component as a child of the link element. This is a frequently recommended trick to allow the user to move the mouse pointer into the popover without causing the popover to disappear. The `customClass` option gives the `<div>` element that represents the popover component a display style of `inline`. This ensures that when the component is inserted it does not indirectly adds a line break at that place in the page due to the default display style for `<div>` elements being `block`.

## Ajax Requests

Ajax requests are not a new topic, as I have introduced this topic back in Chapter 14 as part of the live language translation feature. As before, I'm going to use the `fetch()` function to send an asynchronous request to the server.

The request needs to have the */user/<username>/popup* URL, which I added to the application at the start of this chapter. The response from this request is going to contain the HTML that I need to insert in the popover component, replacing the initial "Loading..." message.

My first problem is how to trigger the request at the time a popover is requested by the user by hovering over a user link. Going through the Popover documentation, the section that describes the events this component supports includes one called `show.bs.popover`, which fires when the popover component is about to be shown.

*app/templates/base.html*: Hover delay.

```
function initialize_popovers() {
  const popups = document.getElementsByClassName('user_popup');
  for (let i = 0; i < popups.length; i++) {
    const popover = new bootstrap.Popover(popups[i], {
      ...
    });
    popups[i].addEventListener('show.bs.popover', async (ev) =>
      // send request here
    });
  }
}
document.addEventListener('DOMContentLoaded', initialize_popover
```

The next issue is how to know what is the username that I need to include in the URL of this request. This name is in the text of the `<a>` link. The show event for the popover component receives an `ev` argument, with the event object. The element that triggered the event can be obtained with `ev.target`, and the expression to extract the text of this link is:

```
ev.target.innerText.trim()
```

The `innerText` property of a page element returns the text contents of the node. The text is returned without any trimming, so for example, if

you have the `<a>` in one line, the text in the following line, and the closing `</a>` in another line, `innerText` will return the newlines and extra whitespace that surrounds the text. To eliminate all that whitespace and leave just the text, I use the `trim()` JavaScript function.

And that is all the information I need to be able to issue the request to the server:

*app/templates/base.html*: XHR request.

```
function initialize_popovers() {
  const popups = document.getElementsByClassName('user_popup');
  for (let i = 0; i < popups.length; i++) {
    const popover = new bootstrap.Popover(popups[i], {
      ...
    });
    popups[i].addEventListener('show.bs.popover', async (ev) =>
      const response = await fetch('/user/' + ev.target.innerTex
      const data = await response.text();
      // update popover here
    });
  }
}
document.addEventListener('DOMContentLoaded', initialize_popover
```

Here I'm using the `fetch()` function to request the */popup* URL for the user represented by the link element. Unfortunately when building URLs directly in the JavaScript side I cannot use the `url_for()` from Flask, so in this case I have to concatenate the URL parts explicitly. Once I have the response, I extract the text from it in `data`. This is the HTML that needs to be stored in the popover component.

## Popover Update

So finally I can now update my popover component with the HTML that was received from the server and that is stored in the `data` constant:

*app/templates/base.html*: Display popover.

```
function initialize_popovers() {
  const popups = document.getElementsByClassName('user_popup');
  for (let i = 0; i < popups.length; i++) {
    const popover = new bootstrap.Popover(popups[i], {
      ...
    });
```

```
        popups[i].addEventListener('show.bs.popover', async (ev) =>
          if (ev.target.popupLoaded) {
            return;
          }
          const response = await fetch('/user/' + ev.target.innerTex
          const data = await response.text();
          const popover = bootstrap.Popover.getInstance(ev.target);
          if (popover && data) {
            ev.target.popupLoaded = true;
            popover.setContent({'.popover-body': data});
            flask_moment_render_all();
          }
        });
      }
    }
    document.addEventListener('DOMContentLoaded', initialize_popover
```

To do this update I first obtain the popover component, which Bootstrap makes available through the `getInstance()` method. If I have a popover component and HTML contents, then I call the popover's `setContent()` method to update its body.

As an optimization, I also set a `popupLoaded` attribute to `true` in the link element. This is so that I don't resend the same request if the popover is opened a second time. Note how I updated the start of the `show` event handler to check for this `popupLoaded` attribute and if it is set, then just return, since the contents of the popover were already updated.

One last detail that needs to be taken care of is the dates and times that are included in the HTML contents that are inserted in the popover components. These timestamps are generated with Flask-Moment's `moment()` function in the Jinja template. Flask-Moment adds JavaScript code to the page that renders all the timestamps when the page is loaded. When new timestamps are added dynamically after the page has loaded, the `flask_moment_render_all()` function needs to be called manually render them, so I inserted a call to this function after updating the HTML content.

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

Hacker News Reddit Twitter LinkedIn

Facebook E-Mail

8 comments

#1 **Kirill Egupov** said a year ago

Hello, Miguel!
I implemented feature from this chapter but some strange thing happened. When I hover user link first time I see when user was last time but when I hover link second time this information not displayed.

#2 Miguel Grinberg said a year ago

@Kirill: did you add the call to `flask_moment_render_all()`? That call needs to be made every time you update the page with flask-moment elements.



#3 **Chris** said 10 months ago

Yo, Miguel!

Just popping in to say how absolutely thrilled I am that someone like you is willing to put out free ressources on the internet to help others learn and grow. And on top of that you update the ressources frequently and answer questions people might have.

People like you are what really drives evolution in a modern society.

If i someday contribute even a fraction of what you have to the generation that comes ofter me, I will consider my life well spend.

Well, that is all.

Thank you so much for all the amazing ressources.

Stay awesome.

br
Chris

#4 **Ryan G** said 8 months ago

In the code above, you have a Jinja code, {% set user_link %}, but I don't see that referenced anywhere else. what is this for? My popups don't want when this is included. If I remove it, the popups work.

#5 Miguel Grinberg said 8 months ago

@Ryan: my guess is that you are skipping chapters. The `user_link` was added in Chapter 13 and it is definitely referenced, in that same template a few lines down. Please compare your code against the working code if you need to find your mistake. Use the GitHub or download links at the top of the article to get the code.

#6 **Hal** said 5 months ago

Good

#7 **Siddharth** said 4 months ago

If anyone else faced the issue that comment #1 from Kirill Egupov talked about, here is how I solved it (and Miguel can correct me in case he finds a mistake in my solution):

I simply added another event listener to each element (popups[i]) like this--->

```
el.addEventListener('shown.bs.popover', () => {
flask_moment_render_all();
});
```

Where const el = popups[i]

Note: The event here is 'shown.bs.popover', which is triggered after the popover has been fully displayed and inserted into the DOM (shown) —unlike 'show.bs.popover', which fires before the popover is shown.

#8 Miguel Grinberg said 4 months ago

@Siddharth: Ah okay, now I understand. The issue is that Bootstrap caches the contents of the popup before they have been transformed by Flask-Moment, so each additional appearance of the popup also needs the call to `flask_moment_render_all()`. Your solution is okay, but I would remove the call to this function in the `show` handler, since this is done in `shown`.

«« « » »»

## Leave a Comment

Name

Email

Comment

Captcha

I'm not a robot

reCAPTCHA
Privacy - Terms

Submit

## The Flask Mega-Tutorial

**New 2024 Edition!**


If you would you like to support my work on my Flask Mega-Tutorial series on this blog and as a reward have access to the complete tutorial nicely structured as a book and/or a set of videos, you can now order it from my Courses site or from Amazon.

Click here to get the Book!
Click here to get the Video Course!

## 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 *3*
- 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 *10*
- Microdot *1*
- Microservices *2*
- Movie Reviews *5*
- Personal *3*
- Photography *7*
- Product Reviews *2*
- Programming *193*
- 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*