The Flask Mega-Tutorial Part XX: Some JavaScript Magic (/post/the-flask-mega-tutorial-part-xx-some-javascript-magic)

April 17 2018

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

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)
- 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) (this article)

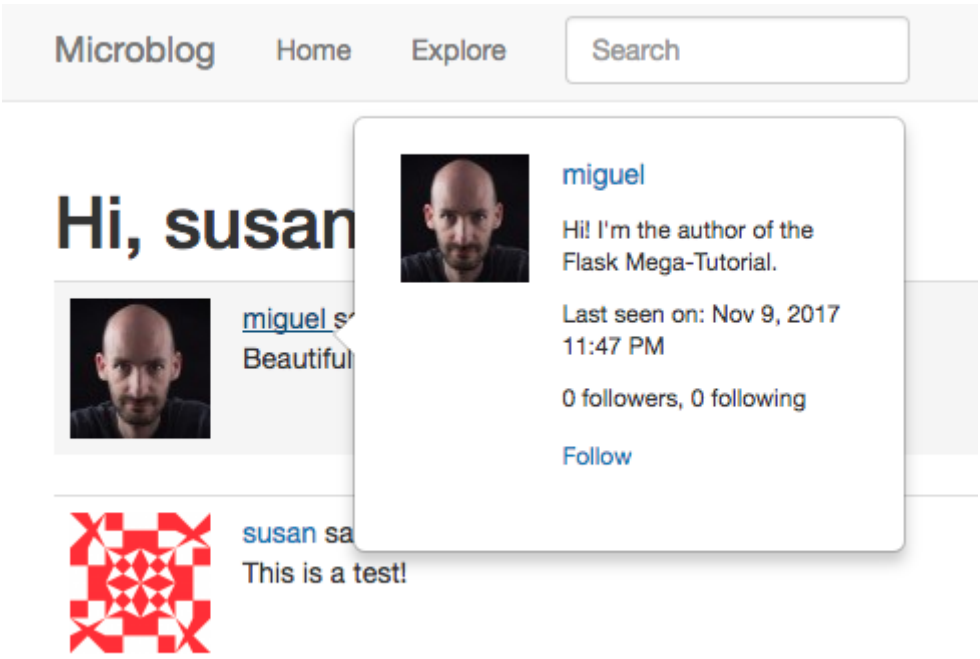
- Chapter 21: User Notifications (/post/the-flask-mega-tutorial-part-xxi-user-notifications)
- Chapter 22: Background Jobs (/post/the-flask-mega-tutorial-part-xxii-background-jobs)
- Chapter 23: Application Programming Interfaces (APIs) (/post/the-flask-mega-tutorial-part-xxiii-application-programming-interfaces-apis)

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).

Nowadays it is impossible to build a web application that doesn't use at least a little 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 (/post/the-flask-mega-tutorial-part-xiv-ajax) 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 (https://github.com/miguelgrinberg/microblog/tree/v0.20), Zip (https://github.com/miguelgrinberg/microblog/archive/v0.20.zip), Diff (https://github.com/miguelgrinberg/microblog/compare/v0.19...v0.20).

Server-side Support

Before we delve into the client-side, let's get the little bit of 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 = User.query.filter_by(username=username).first_or_404()
    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.

```
<table class="table">
    <tr>
        <td width="64" style="border: 0px;"><img src="{{ user.avatar(64) }}"></td>
        <td style="border: 0px;">
            <p><a href="{{ url_for('main.user', username=user.username) }}">{{ user.username }}</a>
            <small>
                {% if user.about_me %}<p>{{ user.about_me }}</p>{% endif %}
                {% if user.last_seen %}
                <p>{{ _('Last seen on') }}: {{ moment(user.last_seen).format('lll') }}</p>
                {% endif %}
                <p>
                    {{ _('%(count)d followers', count=user.followers.count()) }},
                    {{ _('%(count)d following', count=user.followed.count()) }
                }</p>
                {% if user != current_user %}
                    {% if not current_user.is_following(user) %}
                    <p>
                        <form action="{{ url_for('main.follow', username=user.username) }}" method="
                            {{ form.hidden_tag() }}
                            {{ form.submit(value=_('Follow'), class_='btn btn-default btn-sm') }}
                        </form>
                    </p>
                    {% else %}
                    <p>
                        <form action="{{ url_for('main.unfollow', username=user.username) }}" metho
                            {{ form.hidden_tag() }}
                            {{ form.submit(value=_('Unfollow'), class_='btn btn-default btm-sm') }}
                        </form>
                    </p>
                    {% endif %}
                {% endif %}
            </small>
        </td>
    </tr>
</table>
```

The JavaScript code that I will write 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 then display. When the user moves the mouse away the popup will be removed. Sounds simple, right?

If you want to have an idea of how the popup will look, you can now run the application, go to any user's profile page and then append */popup* to the URL in the address bar to see a full-screen version of the popup content.

Introduction to the Bootstrap Popover Component

In Chapter 11 (/post/the-flask-mega-tutorial-part-xi-facelift) 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 (https://getbootstrap.com/docs/3.3/javascript/#popovers), 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.

The HTML portion to do a popover is really simple, you just need to define the element that is going to trigger the popover to appear. In my case, this is going to the clickable username that appears in each blog post. The *app/templates/_post.html* sub-template has the username already defined:

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

Now according to the popover documentation, I need to invoke the `popover()` JavaScript function on each of the links like the one above that appear on the page, and this will initialize the popup. The initialization call accepts a number of options that configure the popup, including options that pass the content that you want displayed in the popup, what method to use to trigger the popup to appear or disappear (a click, hovering over the element, etc.), if the content is plain text or HTML, and a few more options that you can see in the documentation page. Unfortunately, after reading this information 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 from JavaScript after the page is rendered, so that I can then initialize them as popovers.
- The popover examples in the Bootstrap documentation all provide the content of the popover as a `data-content` attribute added to the target HTML element, so when the hover event is triggered, all Bootstrap needs to do is display the popup. That is

really inconvenient for me, because I want to make an Ajax call to the server to get the content, and only when the server's response is received I want the popup to appear.
- 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.

Executing a Function On Page Load

It is clear that I'm going to need to run some JavaScript code as soon as each page loads. The function that I'm going to run will search for all the links to usernames in the page, and configure those with a popover component from Bootstrap.

The jQuery JavaScript library is loaded as a dependency of Bootstrap, so I'm going to take advantage of it. When using jQuery, you can register a function to run when the page is loaded by wrapping it inside a `$( ... )`. I can add this 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() {
        // write start up code here
    });
</script>
```

As you see, I have added my start up function inside the `<script>` element in which I defined the `translate()` function in Chapter 14 (/post/the-flask-mega-tutorial-part-xiv-ajax).

Finding DOM Elements with Selectors

My first problem is to create a JavaScript function that finds all the user links in the page. This function is going to run when the page finishes loading, and when complete, will configure the hovering and popup behavior for all of them. For now I'm going to concentrate in finding the links.

If you recall from Chapter 14 (/post/the-flask-mega-tutorial-part-xiv-ajax), 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 using the jQuery, the expression `$('#post123')` was used in JavaScript to locate this element in the DOM. The `$()` function is extremely powerful and has a fairly sophisticated query language to search for DOM elements that is based on CSS Selectors (https://api.jquery.com/category/selectors/).

The selector that I used for the translation feature was designed to find one specific element that had a unique identifier set as an `id` attribute. Another option to identify elements is by using the `class` attribute, which can be assigned to multiple elements in the page. For example, I could mark all the user links with a `class="user_popup"`, and then I could get the list of links from JavaScript with `$('.user_popup')` (in CSS selectors, the `#` prefix searches by ID, while the `.` prefix searches by class). The return value in this case would be a collection of all the elements that have the class.

Popovers and the DOM

By playing with the popover examples on the Bootstrap documentation and inspecting the DOM in the browser's debugger, I determined that Bootstrap creates the popover component as a sibling of the target element in the DOM. As I mentioned above, that affects the behavior of the hover event, which will trigger a "mouse out" as soon as the user moves the mouse away from the `<a>` link and into the popup itself.

A trick that I can use to extend the hover event to include the popover, is to make the popover a child of the target element, that way the hover event is inherited. Looking through the popover options in the documentation, this can be done by passing a parent element in the `container` option.

Making the popover a child of the hovered element would work well for buttons, or general `<div>` or `<span>` based elements, but in my case, the target for the popover is going to be an `<a>` element that displays the clickable link of a username. The problem with making the popover a child of a `<a>` element is that the popover will then acquire the link behavior of the `<a>` parent. The end result would be something like this:

```
        <a href="..." class="user_popup">
            username
            <div> ... popover elements here ... </div>
        </a>
```

To avoid the popover being inside the `<a>` element, I'm going to use is another trick. I'm going to wrap the `<a>` element inside a `<span>` element, and then associate the hover event and the popover with the `<span>`. The resulting structure would be:

```
        <span class="user_popup">
            <a href="...">
                username
            </a>
            <div> ... popover elements here ... </div>
        </span>
```

The `<div>` and `<span>` elements are invisible, so they are great elements to use to help organize and structure your DOM. The `<div>` element is a *block element*, sort of like a paragraph in the HTML document, while the `<span>` element is an *inline element*, which would compare to a word. For this case I decided to go with the `<span>` element, since the `<a>` element that I'm wrapping is also an inline element.

So I'm going to go ahead and refactor my *app/templates/_post.html* sub-template to include the `<span>` element:

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

If you are wondering where the popover HTML elements are, the good news is that I don't have to worry about that. When I get to call the `popover()` initialization function on the `<span>` elements I just created, the Bootstrap framework will dynamically insert the popup component for me.

Hover Events

As I mentioned above, the hover behavior used by the popover component from Bootstrap is not flexible enough to accommodate my needs, but if you look at the documentation for the `trigger` option, "hover' is just one of the possible values. The one that caught my eye was the "manual" mode, in which the popover can be displayed or removed manually by making JavaScript calls. This mode will give me the freedom to implement the hover logic myself, so I'm going to use that option and implement my own hover event handlers that work the way I need them to.

So my next step is to attach a "hover" event to all the links in the page. Using jQuery, a hover event can be attached to any HTML element by calling `element.hover(handlerIn, handlerOut)`. If this function is called on a collection of elements, jQuery conveniently attaches the event to all of them. The two arguments are two functions, which are invoked when the user moves the mouse pointer into and out of the target element respectively.

app/templates/base.html: Hover event.

```
    $(function() {
        $('.user_popup').hover(
            function(event) {
                // mouse in event handler
                var elem = $(event.currentTarget);
            },
            function(event) {
                // mouse out event handler
                var elem = $(event.currentTarget);
            }
        )
    });
```

The `event` argument is the event object, which contains useful information. In this case, I'm extracting the element that was the target of the event using the `event.currentTarget`.

The browser dispatches the hover event immediately after the mouse enters the affected element. In the case of a popup, you want to activate only after waiting a short period of time where the mouse stays on the element, so that when the mouse pointer briefly passes over the element but doesn't stay on it there is no popups flashing. Since the event does not come with support for a delay, this is another thing that I'm going to need to implement myself. So I'm going to add a one second timer to the "mouse in" event handler:

app/templates/base.html: Hover delay.

```
$(function() {
    var timer = null;
    $('.user_popup').hover(
        function(event) {
            // mouse in event handler
            var elem = $(event.currentTarget);
            timer = setTimeout(function() {
                timer = null;
                // popup logic goes here
            }, 1000);
        },
        function(event) {
            // mouse out event handler
            var elem = $(event.currentTarget);
            if (timer) {
                clearTimeout(timer);
                timer = null;
            }
        }
    )
});
```

The `setTimeout()` function is available in the browser environment. It takes two arguments, a function and a time in milliseconds. The effect of `setTimeout()` is that the function is invoked after the given delay. So I added a function that for now is empty, which will be invoked one second after the hover event is dispatched. Thanks to closures in the JavaScript language, this function can access variables that were defined in an outer scope, such as `elem`.

I'm storing the timer object in a `timer` variable that I have defined outside of the `hover()` call, to make the timer object accessible also to the "mouse out" handler. The reason I need this is, once again, to make for a good user experience. If the user moves the mouse pointer into one of these user links and stays on it for, say, half a second before moving it away, I do not want that timer to still go off and invoke the function that will display the popup. So my mouse out event handler checks if there is an active timer object, and if there is one, it cancels it.

Ajax Requests

Ajax requests are not a new topic, as I have introduced this topic back in Chapter 14 (/post/the-flask-mega-tutorial-part-xiv-ajax) as part of the live language translation feature. When using jQuery, the `$.ajax()` function sends an asynchronous request to the server.

The request that I'm going to send to the server will 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 popup.

My immediate problem regarding this request is to know what is the value of `username` that I need to include in the URL. The mouse in event handler function is generic, it's going to run for all the user links that are found in the page, so the function needs to determine the username from its context.

The `elem` variable contains the target element from the hover event, which is the `<span>` element that wraps the `<a>` element, but I'm using the `$()` jQuery function which returns a list of matching results, so this is going to be a one item list. To extract the username, I can navigate the DOM starting from the first and only item in `elem`, moving to the first child, which is the `<a>` element, and then extracting the text from it, which is the username that I need to use in my URL. With jQuery's DOM traversal functions, this is actually easy:

```
elem.first().text().trim()
```

The `first()` function applied to a list of DOM nodes returns the first one. The `text()` function returns the text contents of a node and all of its children combined, without including HTML tags. This function doesn't do any trimming of the text, so for example, if you have the `<a>` in one line, the text in the following line, and the `</a>` in another line, `text()` will filter the `<a>` and `</a>` but will return all the 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() {
        var timer = null;
        var xhr = null;
        $('.user_popup').hover(
            function(event) {
                // mouse in event handler
                var elem = $(event.currentTarget);
                timer = setTimeout(function() {
                    timer = null;
                    xhr = $.ajax(
                        '/user/' + elem.first().text().trim() + '/popup').done(
                            function(data) {
                                xhr = null
                                // create and display popup here
                            }
                        );
                }, 1000);
            },
            function(event) {
                // mouse out event handler
                var elem = $(event.currentTarget);
                if (timer) {
                    clearTimeout(timer);
                    timer = null;
                }
                else if (xhr) {
                    xhr.abort();
                    xhr = null;
                }
                else {
                    // destroy popup here
                }
            }
        )
    });
```

Here I defined a new variable in the outer scope, `xhr`. This variable is going to hold the asynchronous request object, which I initialize from a call to `$.ajax()`. 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.

The `$.ajax()` call returns a promise, which is this special JavaScript object that represents the asynchronous operation. I can attach a completion callback by adding `.done(function)`, so then my callback function will be invoked once the request is complete. The callback function will receive the response as an argument, which you can see I named `data` in the code above. This is going to be the HTML content that I'm going to put in the popover.

But before we get to the popover, there is one more detail related to giving the user a good experience that needs to be taken care of. Recall that I added logic in the "mouse out" event handler function to cancel the one second timeout if the user moved the mouse

pointer out of the `<span>`. The same idea needs to be applied to the asynchronous request, so I have added a second clause to abort my `xhr` request object if it exists.

Popover Creation and Destruction

So finally I can create my popover component, using the `data` argument that was passed to me in the Ajax callback function:

app/templates/base.html: Display popover.

```
                                function(data) {
                                    xhr = null;
                                    elem.popover({
                                        trigger: 'manual',
                                        html: true,
                                        animation: false,
                                        container: elem,
                                        content: data
                                    }).popover('show');
                                    flask_moment_render_all();
                                }
```

The actual creation of the popup is very simple, the `popover()` function from Bootstrap does all the work required to set it up. Options for the popover are given as an argument. I have configured this popover with the "manual" trigger mode, HTML content, no fade animation (so that it appears and disappears more quickly), and I have set the parent to be the `<span>` element itself, so that the hover behavior extends to the popover by inheritance. Finally, I'm passing the `data` argument to the Ajax callback as the `content` argument.

The return of the `popover()` call is the newly created popover component, which for a strange reason, had another method also called `popover()` that is used to display it. So I had to add a second `popover('show')` call to make the popup appear on the page.

The content of the popup includes the "last seen" date, which is generated through the Flask-Moment plugin as covered in Chapter 12 (/post/the-flask-mega-tutorial-part-xii-dates-and-times). As documented (https://github.com/miguelgrinberg/Flask-Moment#ajax-support) by the extension, when new Flask-Moment elements are added via Ajax, the `flask_moment_render_all()` function needs to be called to appropriately render those elements.

What remains now is to deal with the removal of the popup on the mouse out event handler. This handler already has the logic to abort the popover operation if it is interrupted by the user moving the mouse out of the target element. If none of those

conditions apply, then that means that the popover is currently displayed and the user is leaving the target area, so in that case, a `popover('destroy')` call to the target element does the proper removal and cleanup.

app/templates/base.html: Destroy popover.

```
            function(event) {
                // mouse out event handler
                var elem = $(event.currentTarget);
                if (timer) {
                    clearTimeout(timer);
                    timer = null;
                }
                else if (xhr) {
                    xhr.abort();
                    xhr = null;
                }
                else {
                    elem.popover('destroy');
                }
            }
```