AJAX with jQuery

jQuery is a small JavaScript library commonly used to simplify working with the DOM and JavaScript in general. It is the perfect tool to make web applications more dynamic by exchanging JSON between server and client.

JSON itself is a very lightweight transport format, very similar to how Python primitives (numbers, strings, dicts and lists) look like which is widely supported and very easy to parse. It became popular a few years ago and quickly replaced XML as transport format in web applications.

Loading jQuery

In order to use jQuery, you have to download it first and place it in the static folder of your application and then ensure it's loaded. Ideally you have a layout template that is used for all pages where you just have to add a script statement to the bottom of your `<body>` to load jQuery:

```
<script type=text/javascript src="{{
  url_for('static', filename='jquery.js') }}"></script>
```

Another method is using Google's AJAX Libraries API to load jQuery:

```
<script src="//ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js
<script>window.jQuery || document.write('<script src="{{
  url_for('static', filename='jquery.js') }}">\x3C/script>')</script>
```

In this case you have to put jQuery into your static folder as a fallback, but it will first try to load it directly from Google. This has the advantage that your website will probably load faster for users if they went to at least one other website before using the same jQuery version from Google because it will already be in the browser cache.

Where is My Site?

Do you know where your application is? If you are developing the answer is quite simple: it's on localhost port something and directly on the root of that server. But what if you later decide to move your application to a different location? For example to `http://example.com/myapp`? On the server side this never was a problem because we were using the handy **`url_for()`** function that could answer that question for us, but if we

are using jQuery we should not hardcode the path to the application but make that dynamic, so how can we do that?

A simple method would be to add a script tag to our page that sets a global variable to the prefix to the root of the application. Something like this:

```
<script type=text/javascript>
  $SCRIPT_ROOT = {{ request.script_root|tojson|safe }};
</script>
```

The `|safe` is necessary in Flask before 0.10 so that Jinja does not escape the JSON encoded string with HTML rules. Usually this would be necessary, but we are inside a `script` block here where different rules apply.

Information for Pros:

In HTML the `script` tag is declared `CDATA` which means that entities will not be parsed. Everything until `</script>` is handled as script. This also means that there must never be any `</` between the script tags. `|tojson` is kind enough to do the right thing here and escape slashes for you (`{{ "</script>"|tojson|safe }}` is rendered as `"<\/script>"`).

In Flask 0.10 it goes a step further and escapes all HTML tags with unicode escapes. This makes it possible for Flask to automatically mark the result as HTML safe.

JSON View Functions

Now let's create a server side function that accepts two URL arguments of numbers which should be added together and then sent back to the application in a JSON object. This is a really ridiculous example and is something you usually would do on the client side alone, but a simple example that shows how you would use jQuery and Flask nonetheless:

```
from flask import Flask, jsonify, render_template, request
app = Flask(__name__)

@app.route('/_add_numbers')
def add_numbers():
    a = request.args.get('a', 0, type=int)
    b = request.args.get('b', 0, type=int)
    return jsonify(result=a + b)

@app.route('/')
```

```
def index():
    return render_template('index.html')
```

As you can see I also added an *index* method here that renders a template. This template will load jQuery as above and have a little form where we can add two numbers and a link to trigger the function on the server side.

Note that we are using the **`get()`** method here which will never fail. If the key is missing a default value (here `0`) is returned. Furthermore it can convert values to a specific type (like in our case *int*). This is especially handy for code that is triggered by a script (APIs, JavaScript etc.) because you don't need special error reporting in that case.

The HTML

Your index.html template either has to extend a `layout.html` template with jQuery loaded and the *$SCRIPT_ROOT* variable set, or do that on the top. Here's the HTML code needed for our little application (`index.html`). Notice that we also drop the script directly into the HTML here. It is usually a better idea to have that in a separate script file:

```
<script type=text/javascript>
  $(function() {
    $('a#calculate').bind('click', function() {
      $.getJSON($SCRIPT_ROOT + '/_add_numbers', {
        a: $('input[name="a"]').val(),
        b: $('input[name="b"]').val()
      }, function(data) {
        $("#result").text(data.result);
      });
      return false;
    });
  });
</script>
<h1>jQuery Example</h1>
<p><input type=text size=5 name=a> +
   <input type=text size=5 name=b> =
   <span id=result>?</span>
<p><a href=# id=calculate>calculate server side</a>
```

I won't go into detail here about how jQuery works, just a very quick explanation of the little bit of code above:

1. `$(function() { ... })` specifies code that should run once the browser is done loading the basic parts of the page.
2. `$('selector')` selects an element and lets you operate on it.

3. `element.bind('event', func)` specifies a function that should run when the user clicked on the element. If that function returns *false*, the default behavior will not kick in (in this case, navigate to the `#` URL).
4. `$.getJSON(url, data, func)` sends a `GET` request to *url* and will send the contents of the *data* object as query parameters. Once the data arrived, it will call the given function with the return value as argument. Note that we can use the *$SCRIPT_ROOT* variable here that we set earlier.

Check out the example source for a full application demonstrating the code on this page, as well as the same thing using `XMLHttpRequest` and `fetch`.