PSGI and AJAX for single-page applications

PSGI Ajax JavaScript

Prev Next

In the "old school" web applications and in the previous PSGI examplesevery click on a button caused the the browser to load a whole new page. In more modern web applications the browser only loads a single HTML page that has a lot of JavaScript in it. Then on every click some JavaScript code runs. It sends an AJAX request to the server in the background. Asynchronously. When the reply arrives, another JavaScript function runs that updates the site.

As there is effectively only only one page which is constantly changing, these kind of applications are called Single-page application.

In this example we'll see a very simple application, the good old "echo", but implemented with some low-level tools. We are going to use PSGI on the server side and plain JavaScript on the front-end.

We save the following code in the `echo_ajax.psgi` file and run it using `plackup echo_ajax.psgi`. Visiting `http://localhost:5000/` will show an entry box and a button. If we type in something and click on the button the, text will be displayed under the horizontal line after being sent to the server and back. The URL itself will not change and the page won't be reloaded.

Behind the scenes, when we load the main page, it also loads a few lines of JavaScript. When we click the button a JavaScript function runs, takes the value from the entry box and sends a request to the server. The request is exactly the same as the URL was in the previous case. For example if we typed in "hello" the JavaScript will send a request for `http://localhost:5000/echo?field=hello`. It won't be displayed as this happens in the background. It also happens asynchronously, which means the calling function returns immediately, and when the response arrives another function will be called.

In response to the above request the server will return a string containing a JSON data structure. In case you have not encountered it, JSON is basically the name of all the data structures in JavaScript. As if there was a single word for "hash or array" in perl. I guess "Complex data structure" might be a good candidate. The point is, that we can easily generate such string from our arrays and hashes in Perl, and in JavaScript it is just a native data structure.

The back-end

The back-end code, the part that runs on the server, the part that is written in Perl, uses the same routing as we have already seen in routing and PSGI though there is one substantial difference. The `serve_echo` function instead of returning and HTML, it will return a data structure in JSON format.

Instead of creating HTML, the function fills the variable `$data` with a hash reference. `{ txt => 'You said: ' . $request->param('field') };` or if no value was supplied then `{ txt => 'You did not say anything.' };`.

Then the function returns the 3 element array reference as earlier, but the content type is now set to be `application/json`, after all we are returning JSON and not HTML, and the string returned is the JSON string created by the `to_json` function imported from the JSONmodule.

The `serve_root` function remained the same, just the content it returns changed.

```perl
#!/usr/bin/perl
use strict;
use warnings;

use Plack::Request;
use JSON qw(to_json);

my %ROUTING = (
    '/'      => \&serve_root,
    '/echo'  => \&serve_echo,
);


my $app = sub {
    my $env = shift;

    my $request = Plack::Request->new($env);
    my $route = $ROUTING{$request->path_info};
    if ($route) {
        return $route->($env);
    }
    return [
        '404',
        [ 'Content-Type' => 'text/html' ],
        [ '404 Not Found' ],
    ];
};

sub serve_root {
    my $html = get_html();
    return [
        '200',
        [ 'Content-Type' => 'text/html' ],
        [ $html ],
    ];
}

sub serve_echo {
    my $env = shift;

    my $request = Plack::Request->new($env);
    my $data;
    if ($request->param('field')) {
        $data = { txt => 'You said: ' . $request->param('field') };
    } else {
        $data = { txt => 'You did not say anything.' };
    }
    return [
        '200',
        [ 'Content-Type' => 'application/json' ],
        [ to_json $data ],
    ];
}


sub get_html {
    return q{
      <input id="field">
      <button id="echo">Echo</button>
      <hr>
      <div id="response"></div>

      <script>
      function show_response(resp) {
          document.getElementById('response').innerHTML = resp['txt'];
      }

      function send_text() {
         ajax_get_json('/echo?field=' + document.getElementById('field').valu
      }

      function ajax_get_json(url, on_success) {
          var xmlhttp = new XMLHttpRequest();
          xmlhttp.onreadystatechange = function() {
              if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {
                  console.log('responseText:' + xmlhttp.responseText);
                  on_success(JSON.parse(xmlhttp.responseText));
              }
          }
          xmlhttp.open("GET", url, true);
          xmlhttp.send();
      }

      document.getElementById('echo').addEventListener('click', send_text);
      </script>

    }
}
```

The front-end

The `get_html` function will return the HTML. In a real application this would probably read it from an external template file, but for a small example this works as well. Returned content consists of some HTML and some JavaScript. If we compare it to the HTML that was returned in the previous example then these are the changes:

In the HTML we removed the `form` tag as we don't really need that functionality any more. We kept the input field, but instead of having a `name` attribute, it now has and `id` attribute. It will make it easier for the JavaScript code to find this element and extract its value. The button was also changed form an `input` element of type `button` to a `button` element. We also added an empty `div` element with an `id="response"`. That's where we are going to write the response we get from the server. That's all the HTML.

We also embedded the JavaScript code in the HTML page using the `script` tags. In a real application we would probably put all the JavaScript in a separate file, but again for such a small example we can get away with the sloppiness.

There are 3 JavaScript function, defined. The `show_response` and the `send_text` are relevant to our example. The `ajax_get_json` is just a generic function that could be reused in any other example. It gets two parameters. The first is a URL, the second is a function. It sends a GET request to the URL and makes sure that when the response arrives the function that was the second parameter will be called. We won't go into more details. It is a reasonable, albeit not full solution.

Once the page gets loaded in the browser the last line of the `script` section will be executed:

```
document.getElementById('echo').addEventListener('click', send_text);
```

This will locate the HTML element that has `id="echo"` (which is the `button`) and attaches a call-back to the `click` event of that element. In other words, when the user clicks on the button, the function `send_text` will be called.

If we look at the `send_text` function, it calls the `ajax_get_json` mentioned earlier with a URL and with `show_response` as the second parameter. The first parameter, the URL, is created by the concatenation of `/echo?field=` and the value of the element with `id="field"` using this expression: `document.getElementById('field').value` Thats' the whole "sending the request" part.

When the response arrives from the server, the `show_response`function is called (this was arranged by the `ajax_get_json` helper function) and the JSON data structure sent by the server will be passed as a parameter `resp`.

On the server side we arranged a hash to be sent with a key 'text'. In the client side, in the `show_response` function we can access the value of this key by using `resp['txt']`.

The expression `document.getElementById('response').innerHTML = resp['txt'];` will locate the element with `id="response"` and fill its content with the text that was send from the server. That's all we want.

Conclusion

This was a small example and some parts were missing, but it can provide you the basic understanding how AJAX works, and how can it be used with PSGI as the back-end.