How to use Mashic: Step by Step

November 1, 2015

In this step-by-step tutorial, we explain how to use *Mashic* to automatically sandbox third-party code in client-side Web applications. To illustrate the process, we make a simple Web application which uses *Google Maps*.

Requirements: To run *Mashic*, you need: (1) linux; (2) Make; and (3) *bigloo 3.5a*, which can be found in `http://www-sop.inria.fr/members/Manuel.Serrano/bigloo/`.

Installing Mashic: Installing *Mashic* amounts to downloading its sources (available at `http://web.ist.utl.pt/~ist24690/mashic`) and running Make.

Compiling your Application

We illustrate the use of Mashic with a simple Web application that uses Google Maps (GM) for drawing the Bermuda's triangle. Let us start by examining the original HTML file:

```
1  <html>
2    <head>
3      <script src="http://maps.google.com/maps/api/js?sensor=false
           " type="text/javascript"></script>
4      <script src="input.js"  type="text/javascript"></script>
5    </head>
6    <body onload="initialize();">
7      <div id="map_canvas" style="width: 550px; height: 500px;
           border: 3px gray solid;"></div>
8    </body>
9  </html>
```

The first *script* node in the header element includes the GM *gadget*, while the second one includes the integrator code. When the body of the page is loaded, the initialise method of the integrator is called. The *div* element will be used by the GM gadget for drawing the map.

The integrator code interacts with the GM gadget through the variable `google` (which is defined in the GM script). For instance, the code:

```
map_div = document.getElementById("map_canvas");
map = new google.maps.Map(map_div, my_options);
```

tells the gadget to create a map with the coordinates described in the object bound to `my_options` inside the *div* element with id `"map_canvas"`.

Mashic compiles JavaScript files, it does not compile HTML files. It is the job of the developer to restructure the original HTML file in order make use of the compiled integrator code. In a nutshell, the developer needs to follow the steps given below.

1. **Pre-compilation Step:** Before compilation, the developer must distinguish interactions with DOM elements that are to be kept in the integrator side from interactions with DOM elements that will be sandboxed inside the *iframe*.

2. **Compilation Step:** Compile the integrator file specifying which are the variables that serve to communicate with the gadget.

3. **Post-compilation Step:** Split the original HTML file into two distinct ones: one for running the integrator code (corresponding to the original Web page) and a second one for running the gadget.

Pre-compilation step Before compiling, the developer needs to define a new variable `iframe` that is assigned to the document node: `iframe = document`. Then, the variable `document` must be replaced with the variable `iframe` wherever it is used to interact with a part of the document that is going to be sandboxed inside an *iframe*. For instance, in our running example, the assignment `map_div = document.getElementById("map_canvas")` needs to be replaced by `map_div = iframe.getElementById("map_canvas")` because the element node with id `"map_canvas"` is going to be sandboxed inside an *iframe*.

Compilation step When compiling the integrator code, the developer needs to specify: (1) the name of the input file, (2) the name of the output file (to be generated), and (3) the variables that are used by the integrator to interact with the gadget. Note that variables used to interact with the gadget may have a different names in the *gadget* side and in the *integrator* side. More concretely, the syntax for running Mashic is the following:

$$\$: \textit{js2post -i } \texttt{INPUT} \textit{ -o } \texttt{OUTPUT} \textit{ -a } \texttt{IVAR}_1 \ \texttt{GVAR}_1 \cdots \textit{ -a } \texttt{IVAR}_n \ \texttt{GVAR}_n$$

For instance, to compile the running example, the developer needs to run:

$$\$: \textit{js2post -i input.js -o output.js -a google google -a iframe document}$$

Observe that the variable `iframe` on the integrator side corresponds to the variable `document` on the gadget side.

Post-compilation step The post-compilation step corresponds to restructuring the original HTML file into two separate ones: one for the execution of the integrator and another for the execution of the gadget. The gadget HTML file is included inside the integrator HTML file using an *iframe* as follows:

```
<html>
  <head
    <script src="output.js"  type="text/javascript"></script>
    <script src= "proxies.js" type="text/javascript"></script>
  </head>
  <body onload="initialize();">
      <iframe src="iframe.html" id="widget"></iframe>
  </body>
</html>
```

Observe that the new integrator HTML file does not include the GM gadget, nor the original input file. Instead, it includes the compiled input file and the file `proxies.js` containing the proxy libraries for communication with the iframe. Furthermore, the *div* element where the map is to be drawn is replaced with the *iframe* containing the GM gadget. The HTML file loaded in the *iframe* is the following:

```
<html>
  <head
        <script src="http://maps.google.com/maps/api/js?sensor=
            false" type="text/javascript"></script>
        <script src= "listener.js" type="text/javascript"></
            script>
  </head>
  <body>
      <div id="map_canvas" style="width: 550px; height: 500px;
          border: 3px gray solid;"></div>
  </body>
</html>
```

This HTML file contains the code of the gadget (which is left unchanged), a listener library for communication with the integrator, and the *div* element in which the map is to be drawn.