IS THE BROWSER THE SIDE FOR TEMPLATING?

Francisco J. García-Izquierdo
Dpto. de Matemáticas y Computación, Universidad de La Rioja, Logroño, Spain
francisco.garcia@unirioja.es

Raúl Izquierdo
OOTLab: Laboratory of Object Oriented Technology, Universidad de Oviedo, Oviedo, Spain
raul@uniovi.es

Keywords: (J.8.n) Software engineering in Internet Applications, (J.8.s) Web site development tools, Web templates, (J.8.q) Web browsers, (K.4.2.d) Handicapped persons/special needs.

Abstract: This article examines the feasibility of Browser Side Templating (BST) as valid alternative for Web development, even when it comes to building accessible applications. With BST, templates are processed in the browser using a JavaScript coded engine, thus providing significant performance improvements and making the model-view separation a reality. However BST also has significant drawbacks. The BST dependence on JavaScript affects the accessibility and hides the content of the delivered pages to search engines, hampering Web visibility. Our paper confronts this dilemma and as its main contribution, proposes a technique that allows BST to be accessible supported and semantically crawlable, while preserving all its advantages.

1 INTRODUCTION

Gone are the days when, to develop a dynamic Web application, developers had to print the HTML code directly out to the browser. Over recent years development frameworks have enriched the catalogue of tools at Web developers' disposal, relaxing those tedious code-writing tasks. Frameworks come in several flavors and are suited for different programming languages, but most of them use template systems in the context of a *Model View Controller* (MVC) architecture for the Web. Not surprisingly, as shown by several papers that have analyzed them from diverse perspectives [1], template systems have become the de facto standard for Web application development. Despite most of them naturally placing templates at the server-side, just as the title of this paper suggests, here we are looking at the feasibility of changing the side where templates reside.

Accordingly, here we analyze and evaluate the benefits of *browser-side templating (BST) systems*. In these, the processing engine is moved to the Web browser. BST brings clear benefits, mainly related to the high degree of model-view separation it provides. However, BST is not such a good choice by itself, since its reliance on JavaScript poses significant drawbacks related to the lack of accessibility and the reduction of the semantic payload of the pages delivered to the browser.

The novelty of this paper lies on the proposition of a technique that retains all BST strengths, and overcomes the mentioned weaknesses, allowing us to conclude that BST does constitute a valid alternative to develop Web applications, even when accessible applications are the case. Though we use our own system, Yeast-Templates (http://yeasttemplates.org/), as an example to present the technique, it can be ported to other BST systems.

2 WHY BST? BEYOND SERVER-SIDE TEMPLATING

Although the first papers on BST date back to 2003 [2], AJAX seems to have recently monopolized the use of BST, in such a way that some authors refer to it as an AJAX pattern [3]. Most existing BST sys-

tems (e.g. EJS http://embeddedjs.com/ or Jemplate http://jemplate.net/) focus on making AJAX manipulations easier, providing a template definition language and a processing engine for the dynamic generation of HTML in the browser. But for us, this relationship to AJAX is only an interesting but secondary use of BST. In fact, in this paper, we only consider BST systems that, looking beyond AJAX, were designed to build complete Web applications. Yeast-Templates, JSON-Template (http://code.google.com/p/json-template/) or JS-Templates (http://code.google.com/p/trimpath/wiki/JavaScriptTemplates) are examples of such BST systems

In our opinion, BST deserves attention because it is the most effective way to separate the model from the view in Web applications. Although it doesn't appear so, BST has a lot to do with this concern. We reached this conclusion in a previous work in which we proposed the *double-model approach* [4], an architectural modification of the MVC pattern. This

The double-model approach is based on the use of a different and private model in both the view and the business logic layers of the application. On one hand, the *view's model*, developed by the graphic designers, which holds the data necessary for the view rendering; and on the other hand, the *application's model*, which corresponds to the classical model in MVC. Each layer can only use its model. In particular, no access to the application's model is allowed from the view, this requirement being the key for the separation achievement. A new component called *Transformer* adapts both models taking data from the application's model and re-formatting it as the view's model mandates. The double model approach is associated to the *MVC+mT architecture*: application's-**M**odel, **V**iew, **C**ontroller, view's-**m**odel and **T**ransformer [4].

Making each part of the application dependent on its own model results in the protection of both sides of the application from changes in the other, which is the essence of the separation. Each part can live isolated, being the Transformer, which can be easily developed by the programmers, the only dependent part.

We soon realized that the best way to implement the double model approach was to use a different technology in both the application server and the browser. This would make any attempt of model sharing impossible. BST naturally fits this schema, by using JavaScript variables to implement the view's model and, by means of the transformers, plugging it with any other technology supporting the application's model at the server side.

3 A DEEPER LOOK AT BST

BST templates are practically identical to any other server side templates. A hosting HTML document embeds data placeholders and processing instructions that are interpreted by the BST engine, which replaces the placeholders by the actual data values. Figure 1 shows a Yeast template in which the HTML embeds the following BST code (Table 1 shows other BST samples):

```
<ul><li yst="apply" ystSet="people">
$e.name$</li></ul>
```

This code wires bindings to *view's model* data declared in script blocks by the template designer. Initially this model is populated with test values that allow the template to be tested:

```
<script yst="model">
var people = [{name:'Fred Flintstone'},
              {name:'Barney Rubble'}];
...
</script>
```

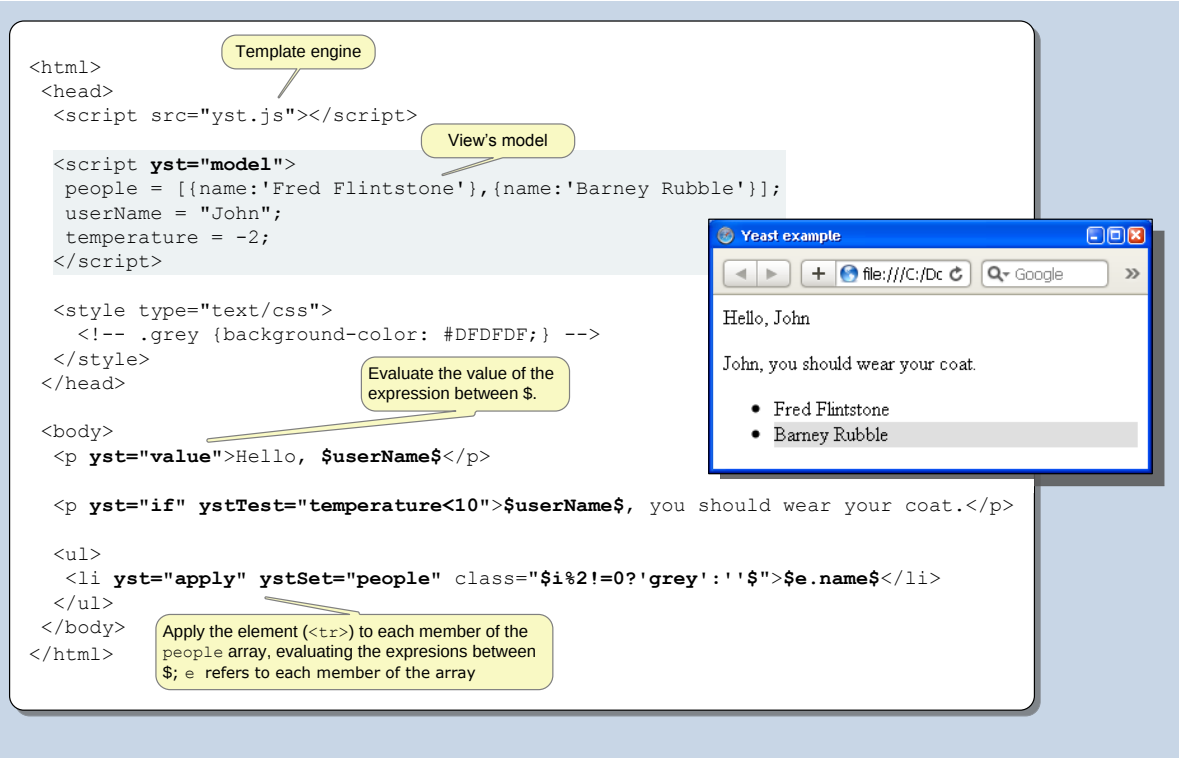
When the template is processed, the following output is produced:

```
<ul><li>Fred Flintstone</li><li>Barney
Rubble</li></ul>
```

To integrate the template into the final Web application is extremely simple. The only action that the application must perform is to replace the test values of the *view's model* in the template with actual data taken from the *application's model*. To do so, the application data are first transformed into the suitable JavaScript format imposed by the view's model. This task is performed by the *Transformer*, which, continuing the example, and assuming that the application's model consists of an array of `Person` objects having a `name` field, could be (using some kind of pseudo-code):

```
out("var people = [");
for (i=0 to persons.length)
  out("{name:'"+persons[i].name+"'}");
out("];");
```

A Yeast Template Example




```
<html>
 <head>
  <script src="yst.js"></script>

  <script yst="model">
   people = [{name:'Fred Flintstone'},{name:'Barney Rubble'}];
   userName = "John";
   temperature = -2;
  </script>

  <style type="text/css">
    <!-- .grey {background-color: #DFDFDF;} -->
  </style>
 </head>

 <body>
  <p yst="value">Hello, $userName$</p>

  <p yst="if" ystTest="temperature<10">$userName$, you should wear your coat.</p>

  <ul>
   <li yst="apply" ystSet="people" class="$i%2!=0?'grey':''$">$e.name$</li>
  </ul>
 </body>
</html>
```

Figure 1. Yeast template example showing its view's model. Yeast processing instructions are specified by a set of non-standard HTML attributes inserted in the HTML elements (*Yeast elements*). The most important Yeast attribute is `yst`, which specifies the type of processing that the element that carries it must undergo. There are eight possible values for `yst`, covering evaluations (`value`), conditionals (`if`), iterations (`apply`), AJAX and sub-templates. *Yeast expressions* are JavaScript expressions enclosed between a couple of `$$` symbols. The example is explanatory enough, but you can consult http://yeasttemplates.org/Doc.html for more details.

4 ANALYZING BST

That BST exploits the browser processing power is only one of the interesting properties characterizing BST. This section analyzes them in depth.

4.1 Designers - programmers independence

Due to its double-model conformance, BST provides Web applications with effective model-view separation. Separation brings encapsulation, clarity and reusability with it, but, in our opinion, the definitive and pragmatic benefit it drives is the effective division of labor between development teams, programmers and designers. Separation simplifies the collaboration workflow solving a lot of interaction issues between the teams [5], and reducing their communication needs. This is especially true using BST, because the double-model approach allows designers to lead the graphic interface design, something that seems very logical but it is rarely enforced. By simply creating a set of templates, the designers, the experts in aesthetics, impose the view's model, being the programmers, the programming experts, responsible for providing and maintaining the *Transformers* which adapt the application's model to it. The visual aspect of the templates may evolve while the project does, but the view's model does not need to change. Moreover, the application's model can be endlessly refactored, but thanks to the transformers, in no case the changes are propagated to the view. The view and the ap-

View's model	

	<script language="javascript">
	  var data = {
	  customer: "John",
	  products: [{ name: "iPhone", price: 499},
	  { name: "Galaxy S", price: 500}]
	  };
	</script>

EJS	
View	Hello <%= customer %> <% for(var i=0; i<products.length; i++) {%> <%= products[i].name %> = <%= products[i].price %> <% } %>
Processing	target_div.innerHTML = new EJS({url: 'view.ejs'}).render(data);

Mustache	
View	var template = " {{customer}} " + " {{#products}} {{name}} = {{price}} {{/items}} "
Processing	target_div.innerHTML = Mustache.to_html(template, data);

Mjt.Template	
View	Hello $data.customer <li mjt.for="product in data.products"> $product.name + " = " + $product.price
Processing	<body onload="mjt.run()" style="display:none"> <!-- MJT code Here --> </body>

Table 1. Examples of several BST systems (the code is equivalent and all share the same view's model)

plication do not touch each other. They can be developed and maintained separately by different development teams that always work in what they are good at.

4.2 Rapid prototyping

Not only can the development teams work separately, but they can also work as they are used to doing it. In our opinion, existing frameworks do not help to properly manage designer-programmer interaction in Web development projects, as they force them to play roles that are not theirs. Clearly there are two options for integrating the application look&feel in the project. First, the designers just develop pure-HTML prototypes, and then the programmers transform them into templates. In our opinion this is not the right way. Besides the waste of skilled programming manpower, there is a high risk of desynchronization between the templates and the original prototypes. We definitely advocate for the second option: designers should make the templates by themselves. The problem now is that designers are compelled to use development tools with which they are unfamiliar. For example, in order to test their designs, they depend on a connection to the server where the application resides.

BST simplifies the template development, allowing designers to get by on the tools they commonly use: HTML and JavaScript. The template engine is in the browser, the designers' realm, so they can even work disconnected from the applications server

Yeast Templates bechmarking

In benchmarking Yeast-Templates we considered a dynamic web page simulating the timeline of a Twitter user (related material and more details about these tests can be found in http://yeasttemplates.org/bench/). The page size varies depending on the number of displayed tweets. We considered 10, 40 and 80 tweets, comparing four implementations: JSP, Yeast, browser-side cacheable Yeast and Accessible-Yeast (see Section 5). Experiments were run using five identical Intel(R) Core(TM)2 Duo E8400 3 GHz (2 Gb. RAM) computers, connected through a 100 Mb. LAN. We used Apache JMeter 2.4 to measure the server (Tomcat 6.0.26) throughput, configuring three slaves and one master. Each slave invoked 20 simultaneous requests.

The transferred data is greatly reduced when Yeast is used. E.g., for 10 tweets JSP size is 37.371 bytes, whereas Yeast takes 10.383 bytes (3.476 when cached). Yeast improves the throughput with respect to the JSP implementation by at least 98%, and even 138% if combined with browser-side cache techniques (Figure 2). The throughput decreases drastically when Yeast templates are processed in the server due to the heavy computation that the timeline template requires. Results are expected to be similar for other BST systems, since the burden of the request process is on the transformation from the application model to the view's model, which is similar for every BST.

The previous tests were run assuming a template-body cache hit of 100%, equivalent to downloading only the skeleton of the template (see Section 4.3). This is only true for frequently used templates but, due to the need to separately download the body and the skeleton of the template, browser caching techniques can be counterproductive for less-used templates. Figure 3 depicts the effect of the template-body cache hit ratio on the server throughput compared to the JSP implementation. As expected, the throughput decreases for lower cache hit ratios, but in this experiment it is always greater than that of JSP.

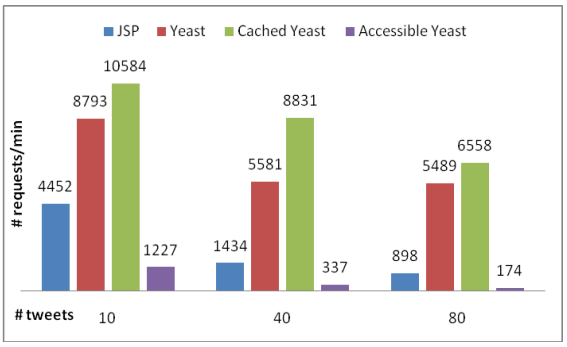


Figure 2 Server throughput

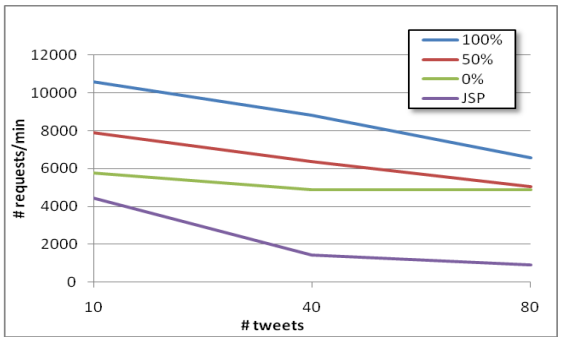


Figure 3 Throughput for various cache hit ratios

to evaluate their designs. Consequently, BST encourages the agile development of the view prototypes, which can be integrated unmodified into the final deliverable being promoted to the definitive application views.

4.3 Performance improvements

Contrary to what might appear, the performance benefits provided by BST are not based on the elimination of the server-side processing. After all, the server must transform the application's model data into the view's model format, which can be a considerable task, even though the resulting format is more compact (only JavaScript values versus HTML code). The response size reduction cannot be considered a relevant advantage either, because although usually the size of the BST template may be less than the expanded HTML, examples can be found in which the size is larger due to the BST boilerplate.

The definitive performance benefit is down to the browser-side caching possibilities [2, 6], which are orthogonal to and complement other caching alternatives [7]. Most BST systems, e.g. EJS, FlyingTemplates [6], allow the storage of the template body (HTML with BST code) in String variables that can be moved to separate JavaScript files, which can be cached in the browser in the first template load. Thus, the original template can be replaced by a much more compact document containing the engine, the view's model data and, as body, a single script tag that loads the file with the original template body and triggers its processing. E.g. the following could be the skeleton of a cacheable version of the Yeast template in Figure 1:

```
<html>
 <head>
  <script src="yst.js"></script>
```

```
  <script yst="model">
   people = ...
  </script>
 </head>
 <body>
  <script src="templ_body.js"></script>
 </body>
</html>
```

Where the content of the `templ_body.js` file is the following:

```
body = '<p yst="value">Hello,$userName$
</p>...';
document.write(body);
```

This is the approach used in FlyingTemplates [6]. The benchmarking section shows how the use of a BST system as Yeast can improve server performance compared to traditional technologies such as JSP by up to 100%, or 120% if combined with browser-side caching techniques. Similar results are reported by other papers (see the related work section.

4.4 BST drawbacks

BST drawbacks are related to the fact that the dynamic content of the final rendered template is not part of the initial page content, but is generated with the template JavaScript processing. Though in the present AJAX-world it´s difficult to imagine the Web without scripting, the use of JavaScript in Web pages still causes controversy, above all when security [8] or accessibility aspects are considered. Despite the fact that the latest available statistics show that 95% of browsers have the use of JavaScript enabled (w3schools.com), and that 89.52% of the 10000 top websites use JavaScript (http://trends.builtwith.com-/docinfo/Javascript) there exist other types of devices with limited JavaScript support (phones, assistive technologies, search bots…). The unavoidable JavaScript based processing model of BST has negative consequences on the Web accessibility [9] and on the page semantics.

4.4.1 BST is not accessible

The intensive use of JavaScript that characterize BST does not conform to the Web Content Accessibility Guidelines (WCAG), breaking the accessibility principle of "making resources accessible to all users, regardless of the technical, physical or mental restrictions on the client side" [9]. In BST, JavaScript is not an enhancement but an essential ingredient. We cannot expect any kind of graceful degradation using a BST template in a browser without JavaScript. The page simply crashes, this constituting a priority 1 violation of WCAG. Though WCAG 2.0 [10] and specifications such as WAI-ARIA [11] are more "open-minded" than WCAG 1.0 in reference to the use of technologies like JavaScript, criteria imposed by the version 1.0 will prevail in the middle-term, since governmental regulations, inspired mainly on WCAG 1.0, cannot be adapted immediately.

4.4.2 BST is not semantically crawlable

If accessibility is an important issue, this is an even more devastating argument. When search robots, which do not process JavaScript, reach BST documents they do not find the entire semantic payload that the page must show to users. They only see a mixture of HTML plus BST code that they don't know how to interpret, let alone index. The dynamic content data, the view's model, are confined inside script blocks, usually ignored by Web crawlers. This is a relevant factor, with direct economic impact. Consider that much of the success of a company that sells on Internet *depends on its potential customers finding its products using a search engine*. The visibility of the products will be negatively affected by the use of BST

Unfortunately, both downsides are an insurmountable barrier that hinders the adoption of BST systems to build complete Web applications and apparently relegates them to be used in AJAX. We cannot do anything to turn a BST page into an accessible and semantically crawlable document. But we want to benefit from BST undeniable upsides that speed up development and improve performance. Our BST system, Yeast-Templates, solves this seemingly intractable dilemma.

5 PROVIDING ACCESIBILITY TO BST

The only way for a BST system to provide accessible content is to send the template processing back to the server. Consequently, BST code is no longer an impediment, simply because it is removed by the server. Nevertheless, Yeast-Templates provides a smart mode of work that makes that the server processes the templates by default, but transparently, it allows JavaScript-enabled clients to take advantage of BST.

Related work on BST

In our opinion, the coverage of BST in the literature is only partial. To our knowledge, no work has covered in depth all BST aspects treated in this paper. Though the first reference dates from 2003 [1], BST has not been forgotten by the literature. Recent references analyze the performance improvements resulting from the fact that the browser not only processes the templates, but it can also cache them. Tatsubori and Suzumur [2] propose Flying-Template, reporting server throughput increments ranging from 59% to 104%. Similarly, Benson et al. [3] present Sync Kit toolkit, which even caches the view's model in the browser using the HTML5 Web SQL Database. Nevertheless, both papers forget to mention the accessibility issues discussed here.

Rabinovich et al. [1] is the only reference we know that considers the case of the client having JavaScript disabled, proposing the server processing as a solution. Our proposal is more general since it does not need an explicit link to the alternative version and detects the client capabilities without resorting to the User-agent header, which doesn't accurately inform about the client JavaScript capabilities.

None of the aforementioned papers dealt with the analysis of the separation between the model and the view in Web applications, let alone with the contribution of BST to it. Model-view decoupling has been studied by Parr [4] or Kojarski and Lorenz [5]. Parr formally analyzes this concern, proposing a set of rules for the achievement of strict model-view separation. Kojarski and Lorenz differentiate between intra-crosscutting (tangled code) and inter-crosscutting (code scattering).

References

[1] Rabinovich, M., Xiao, Z., Douglis, F., Kalmanek, C.: Moving edge-side includes to the real edge: the clients. In: USITS'03: Proc. of the 4th Conf. on USENIX Symposium on Internet Technologies and Systems, Berkeley, CA, USA, USENIX Association (2003) 12–12

[2] Tatsubori, M., Suzumura, T.: HTML templates that fly: a template engine approach to automated offloading from server to client. In: WWW '09: Proc. of the 18th Int. Conf. on World Wide Web, New York, NY, USA, ACM (2009)

[3] Benson, E., Marcus, A., Karger, D., Madden, S.: Sync kit: a persistent client-side database caching toolkit for data intensive websites. In: WWW '10: Proc. of the 19th Int. conf. on World Wide Web, New York, NY, USA, ACM (2010) 121–130

[4] Parr, T.J.: Enforcing strict model-view separation in template engines. In: WWW '04: Proc. of the 13th Int. conf. on World Wide Web, New York, NY, USA, ACM (2004) 224–233

[5] Kojarski, S., Lorenz, D.H.: Domain driven web development with WebJinn. In: OOPSLA '03: Companion of the 18th Annual ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications, New York, NY, USA, ACM (2003) 53–65

The question boils down to detecting the client's JavaScript processing capacity and, if it is disabled, providing a version of the page without BST. In fact, this is the spirit of the WCAG 1.0 checkpoint 6.3. But Yeast-Templates goes a step further. Firstly, it automates the generation of the alternative accessible version of the page, avoiding its desynchronization during the maintenance. And secondly, it avoids the need of an explicit link to the accessible version.

Although the idea seems simple, it's tricky to implement. Our server-side BST infrastructure, Yeast-Server API, must be extended with a component that encapsulates a server-side JavaScript engine responsible for the server processing of the Yeast-Templates. We have implemented this component, named *Yeipee*, using the Mozilla Rhino library, the only option available for Java, the platform on which Yeast-Server runs. Other JavaScript engines, such as Google's V8 or Mozilla's SpiderMonkey are available for other BST systems. The first obstacle in this development was that these engines don't provide the infrastructure to process the DOM of the templates, unlike browsers, which do include this facility. Consequently, DOM manipulation based BST systems cannot be processed on the server. In our case, to avoid the DOM manipulation, it was necessary to develop an alternative compiler/processor for Yeast-Templates.

Using Yeipee, Yeast templates are decomposed into a set of fragments that may contain either raw HTML or Yeast-Templates code. When a certain template is needed, the Yeipee's Rhino processor loads and evaluates the actual view's model data for the response. Then Yeipee iterates over the template fragments; if the fragment contains Yeast-Templates code, the Rhino processor processes it with the previously evaluated model data, adding the result to the overall resultant document; HTML fragments are directly added to the response.

By removing the BST code, and returning only HTML, Yeipee processors also remove all the obstacles to accessibility. But, in turn, this processing

mode is perceptibly slower than the original (see the benchmarking section) and incompatible with the caching strategies that characterize BST. We are losing the performance benefits that BST provides.

Fortunately, Yeast-Server needn't always resort to the Yeipee processing. We have devised a progressive enhancement mechanism in order to use Yeipee processors transparently and only when it is strictly necessary. By default, Yeast-Server processes templates using Yeipee. But the processed template returned to the client now embeds a little script used to detect if JavaScript is enabled. If this is the case, in every request the browser attaches a parameter (`yst.yeipee=OFF`) to disable the server Yeipee processing. The script tries to set up a session cookie to carry that parameter, but if cookies are disabled, `onClick` and `onSubmit` event handlers are registered respectively to the page links and forms. When Yeast-Server detects that a certain request includes that parameter with that value it changes its processing mode, omitting the Yeipee processing and returning Yeast-Templates content. If the user-agent isn't JavaScript enabled then the aforementioned enhancement script will not be executed, the parameter won't be included, and the request to the server will be processed with Yeipee. Note that this is the case for Web crawlers which, when accessing a Yeast-Templates page, will retrieve a server-side processed version of the page with the complete semantic content.

To conclude, we want to emphasize that this strategy delivers accessible content by default and its application is transparent to the user and the development teams, designers and application programmers. The overall result is that, without being penalized with any extra task, developers keep taking advantage of the BST development philosophy, which gives them independence and agility in the development, and applications continue to benefit from BST performance improvements, relinquishing to them only when it is unavoidable.

4 CONCLUSIONS

AJAX applications have traditionally used BST as a tool for easily generating dynamic HTML fragments with which to update page sections upon the receipt of fresh data from the server. However, throughout the paper, we have looked beyond AJAX, analyzing the characteristics of BST to determine whether this technology is applicable in general-purpose template systems. We have found compelling advantages that would recommend the adoption of BST: the double-model conformance that speed up the development process, and the provided performance improvements, above all due to the caching possibilities of this technology. But, we have also found a fundamental drawback related to the inaccessibility and the loss of semantics in the delivered pages. The main contribution of this paper is the proposition of a smart technique to solve this problem without having to relinquish the advantages of BST. Therefore, our final conclusion is that BST constitutes a true option to be used as core technology for template systems, even when accessible applications must be developed.

REFERENCES

[1] Parr, T.J.: Enforcing strict model-view separation in template engines. In: WWW '04: Proceedings of the 13th International Conference on World Wide Web, New York, NY, USA, ACM (2004) 224–233

[2] Rabinovich, M., Xiao, Z., Douglis, F., Kalmanek, C.: Moving edge-side includes to the real edge: the clients. In: USITS'03: Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems, Berkeley, CA, USA, USENIX Association (2003) 12–12

[3] Mahemoff, M.: Ajax Design Patterns. O'Reilly Media (2006)

[4] García-Izquierdo, F., Izquierdo, R., Juan Fuente, A.: A double-model approach to achieve effective model-view separation in template based web applications. In: Web Engineering. Volume 4607 of Lecture Notes in Computer Science. Springer Verlag (2007) 442–456

[5] Böttger, H., Møller, A., Schwartzbach, M.I.: Contracts for cooperation between web service programmers and HTML designers. Journal of Web Engineering **5** (2003)

[6] Tatsubori, M., Suzumura, T.: HTML templates that fly: a template engine approach to automated offloading from server to client. In: WWW '09: Proceedings of the 18th International Conference on World Wide Web, New York, NY, USA, ACM (2009)

[7] Ravi, J., Yu, Z., Shi, W.: A survey on dynamic web content generation and delivery techniques. Journal of Network and Computer Applications **32**(5) (2009) 943 – 960 Next Generation Content Networks.

[8] Yue, C., Wang, H.: Characterizing insecure JavaScript practices on the web. In: 18th International World Wide Web Conference. (April 2009) 961–961

[9] Kern, W.: Web 2.0 - End of accessibility? – Analysis of most common problems with Web 2.0 based applications regarding Web accessibility. International Journal of Public Information Systems **2008:2** (2008) 131–154

[10] Ribera, M., Porras, M., Boldu, M., Termens, M., Sule, A., Paris, P.: Web content accessibility guidelines 2.0: A further step towards accessible digital information. Program: electronic library and information systems **43**(4) (2009) 392–406
[11] Gibson, B.: Enabling an accessible Web 2.0. In: W4A '07: Proceedings of the 2007 International Cross-Disciplinary Conference on Web Accessibility (W4A), New York, NY, USA, ACM (2007) 1–6

Short author biographies

Francisco J. García-Izquierdo is an assistant professor at the Universidad de La Rioja, Spain. Currently his research interests include web engineering, web accessibility, modeling theories, and computer science teaching. He has a PhD in computer science from the University of Zaragoza. Contact him at francisco.garcia@unirioja.es.

Raúl Izquierdo is an assistant professor at the Universidad de Oviedo, Spain. His research interests include web engineering, interaction design, web usability and compiler construction. Izquierdo has a PhD in computer science from the University of Oviedo. Contact him at raul@uniovi.es.

Complete contact information

Francisco J. García-Izquierdo

Mailing address:
Edificio Vives
C/ Luis de Ulloa, s.n.
Universidad de La Rioja.
26004-Logroño. Spain.

Phone/Fax: (+34) 941 299 260 / (+34) 941 299 460
Email: francisco.garcia@unirioja.es

Raúl Izquierdo

Mailing address:
Facultad de Ciencias.
C\ Calvo Sotelo s/n.
Despacho 238
Universidad de Oviedo.
33007-Oviedo. Spain

Phone/Fax: (+34) 985 103 172/ (+34) 985 103 175
Email: raul@uniovi.es