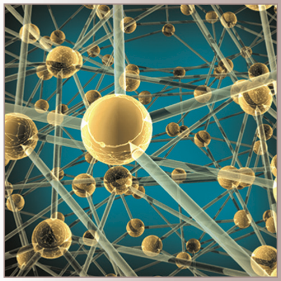


Is the Browser the Side for Templating?

Browser-side templating (BST) is a valid alternative for Web development, even when it comes to building accessible applications. BST processes templates in the browser using a JavaScript-coded engine, providing significant performance improvements and making model–view separation a reality. However, it also has significant drawbacks. BST's dependence on JavaScript affects its accessibility and hides the delivered pages' content from search engines, hampering Web visibility. The authors confront this dilemma and propose a technique that lets BST be accessible and semantically crawlable, while preserving its advantages.

Francisco J. García-Izquierdo
Universidad de La Rioja

Raúl Izquierdo
Universidad de Oviedo

Gone are the days when, to create a dynamic Web application, developers had to print HTML code directly to the browser. In recent years, development frameworks have enriched the catalogue of tools at Web developers' disposal, reducing those tedious code-writing tasks. Such frameworks come in several flavors and are suited for different programming languages, but most use template systems in the context of a model–view–controller (MVC) architecture for the Web. Unsurprisingly, template systems are now the de facto standard for Web application development.[1] Although most such systems naturally place templates at the server side, here we look at the feasibility of changing the side where templates reside.

Accordingly, we analyze and evaluate the benefits of *browser-side templating* (BST) systems, which move the processing engine to the Web browser. BST brings clear benefits, mainly related to the high degree of model–view separation it provides. However, BST isn't a good choice by itself, because its reliance on JavaScript poses significant drawbacks related to a lack of accessibility and a reduction in the semantic payload of the pages delivered to the browser.

We propose a technique that retains BST's strengths and overcomes its weaknesses, letting us conclude that BST constitutes a valid alternative to developing Web applications, even when accessible applications are needed. We use our own system, Yeast-Templates (http://yeasttemplates.org), as an example to present the technique, but other BST systems can also adopt it.

Beyond Server-Side Templating

Although the first work on BST dates back to 2003,[2] Ajax seems to have monopolized BST use in such a way that some authors refer to it as an Ajax pattern.[3] Most existing BST systems, such as Embedded JavaScript (EJS;

1089-7801/12/$31.00 © 2012 IEEE

http://embeddedjs.com) or Jemplate (http://jemplate.net), focus on making Ajax manipulations easier, providing a template definition language and a processing engine for dynamically generating HTML in the browser. For us, this relationship to Ajax is an interesting but secondary use for BST. Looking beyond Ajax, we consider only BST systems whose design allows for building complete Web applications — for instance, Yeast-Templates, JSON-Template (http://code.google.com/p/json-template/), or JS-Templates (http://code.google.com/p/trimpath/wiki/JavaScriptTemplates).

BST deserves attention because it's the most effective way to separate the model from the view in Web applications. 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 approach is based on using a different and private model in the application's view and business-logic layers. On one hand, we have the view's model, developed by graphic designers, which holds the data necessary for rendering the view; on the other hand, we have the application's model, which corresponds to the classical model in MVC. Each layer can use only its own model. In particular, the view can't access the application's model. A component called the *transformer* adapts both models, taking data from the application's model and reformatting it as the view's model mandates. The double-model approach is the basis of the MVC+mT architecture — that is, the classical MVC pattern plus the view's model and the transformer.[4]

Making each part of the application dependent on its own model isolates and protects each side of the application from changes in the other, which is the essence of separation. The transformer — which programmers can easily develop — is 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 at model sharing impossible. BST naturally fits this schema by using JavaScript variables to implement the view's model and, via the transformers, plugging in any other technology to support the application's model at the server side.

A Deeper Look at BST

BST templates are practically identical to any other server-side template. A hosting HTML document embeds data placeholders and processing instructions that the BST engine interprets. The engine then replaces the placeholders with actual data values. Figure 1 shows a Yeast template in which the HTML embeds the following BST code (other BST samples are available in the Web appendix at http://doi.ieeecomputersociety.org/10.1109/MIC.2011.81):

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

This code wires bindings to view-model data declared in script blocks by the template designer. Initially, this model is populated with test values that let the designer test the template:

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

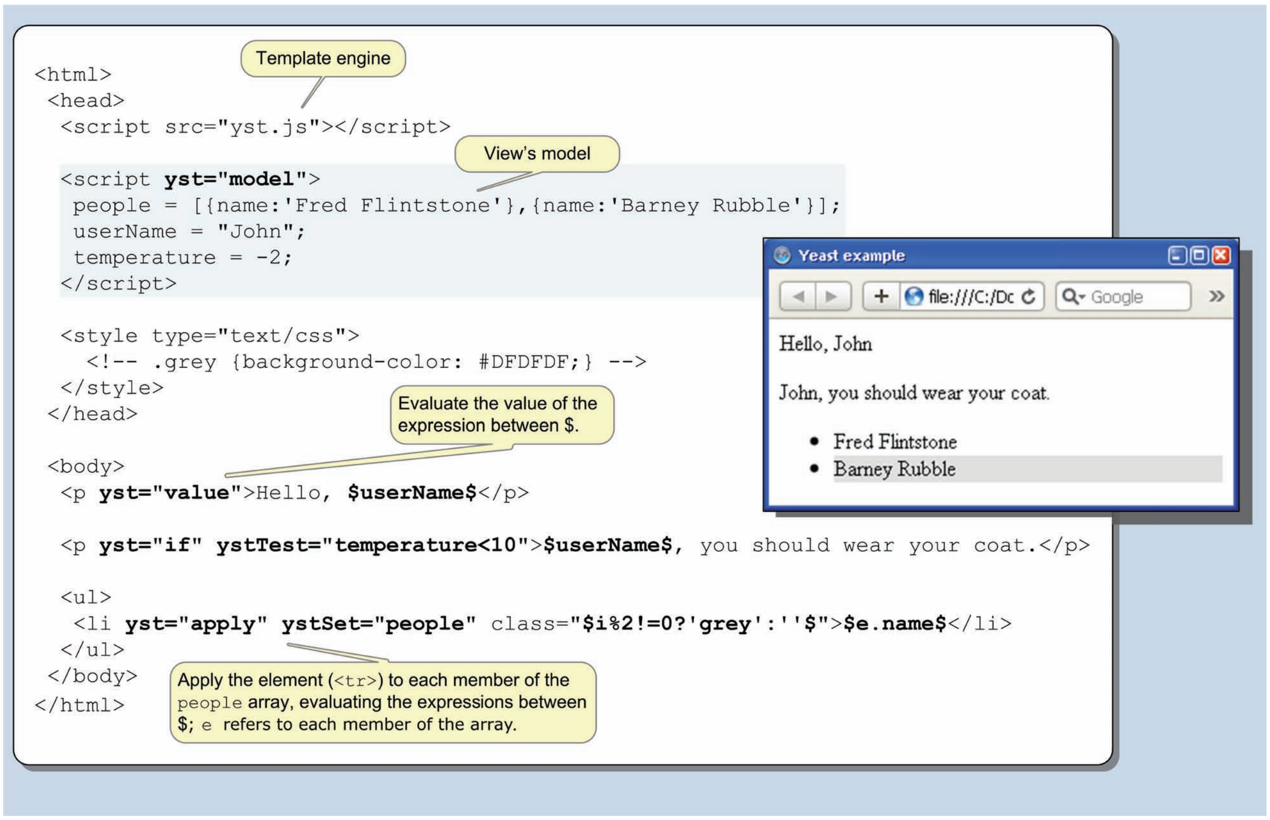
When the engine processes the template, it produces the following output:

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

Integrating the template into the final Web application is extremely simple. The application must only 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 transformer first transforms the application data into the JavaScript format that the view's model requires. Continuing the example, the application's model consists of an array of `Person` objects with a `name` field, so the transformer could be (using some kind of pseudocode)

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

Some BST systems, such as Yeast-Templates, provide server-side APIs and utilities that facilitate



```
<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>
```

Template engine

View's model

Evaluate the value of the expression between $.

Apply the element (`<tr>`) to each member of the `people` array, evaluating the expressions between $; `e` refers to each member of the array.

Hello, John

John, you should wear your coat.

- Fred Flintstone
- Barney Rubble

Figure 1. Yeast template example showing the view's model. Yeast processing instructions are specified by a set of nonstandard 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; `yst` has eight possible values covering evaluations (`value`), conditionals (`if`), iterations (`apply`), Ajax, and subtemplates. Yeast expressions are JavaScript expressions enclosed between `$$` symbols. More details are available at http://yeasttemplates.org/doc.html.

the development of transformers for object-oriented or relational-database-based models.

Analyzing BST

That BST exploits browser processing power is only one of its interesting properties. Let's look in-depth at a few more here.

Designer–Programmer Independence

Due to its double-model conformance, BST provides Web applications with effective model–view separation. With separation comes encapsulation, clarity, and reusability, 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 myriad interaction issues between teams,[5] and reducing their communication needs. This is especially true using BST because the double-model approach lets designers lead the graphic interface design, something that seems logical but is rarely enforced by existing development frameworks. By simply creating a set of templates, the designers — the experts in aesthetics — impose the view's model, leaving the programmers — the programming experts — responsible for providing and maintaining the transformers, which adapt the application's model to the view's model. The templates' visual aspect might evolve as the project does, but the view's model doesn't need to change. Moreover, the application's model can be endlessly refactored, but thanks to the transformers, the changes never propagate to the view. The view and the application don't touch each other. They can be developed and maintained separately by different development teams that always work on what they're good at.

Rapid Prototyping

Not only can the development teams work separately, but they can also work in ways they're used to. Existing frameworks don't help properly manage designer–programmer interaction in Web development projects because they force people to play roles that aren't theirs. Two options exist for integrating an application's look and feel into the project. In the first, designers develop pure HTML prototypes, after which programmers transform them into templates. However, we don't believe this method is effective. In addition to the waste of skilled programming manpower, a high risk exists of desynchronization between the templates and the original prototypes. We advocate instead for the second option: designers make the templates independently. The problem with this option is that designers are compelled to use development tools with which they're unfamiliar. For example, to test their designs, they depend on a connection to the server where the application resides.

BST simplifies template development, letting designers 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 to evaluate their designs. Consequently, BST encourages the agile development of the view prototypes, which the development team can integrate unmodified into the final deliverable, promoting them to the definitive application views.

Performance Improvements

Contrary to appearances, the performance benefits BST provides don't come from eliminating 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). We can't consider the response-size reduction to be a relevant advantage either, because although the BST template is usually smaller than the expanded HTML, occasionally it's larger due to the BST boilerplate.

The definitive performance benefit comes down to BST's browser-side caching possibilities,[2,3] which complement and are orthogonal to other caching alternatives.[6] Most BST systems, such as EJS or FlyingTemplate,[7] let the template declare its body (HTML with BST code) using string variables that the designer can move to separate JavaScript files that are cached in the browser in the first template load. Thus, the template designer — or even the server automatically — can replace the original template with a much more compact document containing the engine, the view's model data, and a single script tag that loads the file with the original template body and triggers its processing. For instance, the following could be the skeleton of a cacheable version of the Yeast template Figure 1 shows:

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

The content of the `templ_body.js` file is

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

This is the approach FlyingTemplate uses. The "Yeast-Templates Benchmarking" sidebar shows how using a BST system such as Yeast can improve server performance compared to traditional technologies such as JSP by up to 98 percent, or 138 percent if combined with browser-side caching techniques. Others report similar results (see the "Related Work in Browser-Side Templating" sidebar).

BST Drawbacks

BST's drawbacks are related to the fact that the final rendered template's dynamic content isn't part of the initial page content but is instead generated during template JavaScript processing. Although in the present Ajax-based world, we can't imagine the Web without scripting, using JavaScript in webpages still causes controversy, especially when we consider security[8] or accessibility issues. Even though the

Yeast-Templates Benchmarking

To benchmark Yeast-Templates, we considered a dynamic webpage simulating the timeline of a Twitter user (related material and more details about these tests is at 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: Java Server Pages (JSP), Yeast, browser-side cacheable Yeast, and Accessible-Yeast (see the "Providing Accessibility to BST" section in the main text). We ran experiments using five identical Intel Core 2 Duo E8400 3-GHz computers with 2 Gbytes RAM, connected through a 100-Mbit per second 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.

Using Yeast greatly reduces the amount of transferred data. For 10 tweets, the JSP size was 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 percent, and even 138 percent if combined with browser-side caching techniques (see Figure A). The throughput decreases drastically when, to support accessibility, the server processes the Yeast templates because of the heavy computation that the timeline template requires. We expect these results to be similar for other browser-side templating (BST) systems, given that the burden of the request process is on the transformation from the application's model to the view's model, which is similar for every BST.

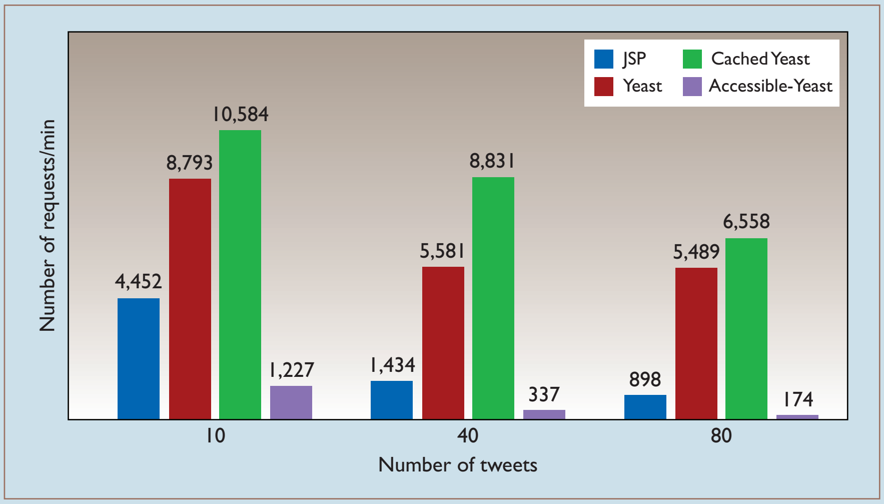


Figure A. Server throughput. For the specified experiment, and compared to JSP, Yeast templates improve the server throughput by at least 98 percent, and even 138 percent if combined with browser-side caching techniques.

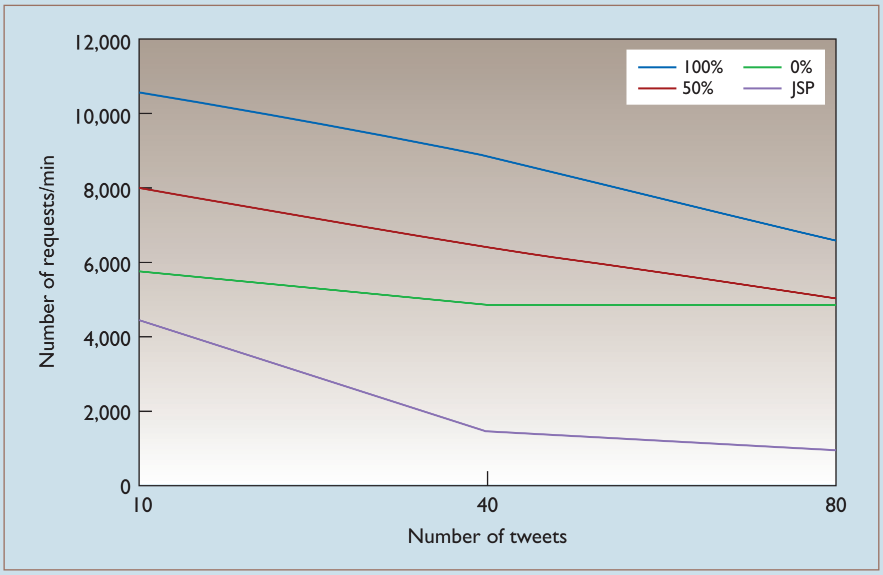


Figure B. Throughput for various cache hit ratios (100, 50, and 0 percent). Although the server throughput decreases for lower cache hit ratios, it's always greater than the throughput of the JSP case.

We ran our tests assuming a template-body cache hit of 100 percent, equivalent to downloading only the template's skeleton. This is true only for frequently used templates but, because of the need to separately download the template's body and skeleton, browser caching techniques can be counterproductive for less-used templates. Figure B 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 is always greater than JSP's throughput.

Related Work in Browser-Side Templating

The coverage of browser-side templating (BST) in the literature is only partial. To our knowledge, no work has covered in depth all of the BST aspects that we discuss in the main text. Although the first reference dates from 2003,[1] more recent references analyze performance improvements resulting from the fact that the browser not only processes the templates but can also cache them. Michiaki Tatsubori and Toyotaro Suzumura propose FlyingTemplate,[2] reporting server throughput increments ranging from 59 percent to 104 percent. Similarly, Edward Benson and his colleagues present the Sync Kit toolkit,[3] which even caches the view's model in the browser using the HTML 5 Web SQL Database. Nevertheless, neither paper mentions the accessibility issues we discuss in the main text.

Michael Rabinovich and his colleagues provide the only previous work we know of that considers a case in which the client has JavaScript disabled, proposing server processing as a solution.[1] Our proposal is more general, given that it doesn't 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 the server about the client's JavaScript capabilities.

None of the aforementioned works analyze the separation between the model and the view in Web applications, let alone consider BST's contribution to it. Others have studied model–view decoupling: Terence Parr formally analyzes this concern, proposing a set of rules for achieving strict model–view separation,[4] and Sergei Kojarski and David Lorenz differentiate between intra-crosscutting (tangled code) and inter-crosscutting (code scattering).[5]

References

1. M. Rabinovich et al., "Moving Edge-Side Includes to the Real Edge: The Clients," *Proc. 4th Usenix Symp. Internet Technologies and Systems* (USITS 03), Usenix Assoc., 2003, p. 12.
2. M. Tatsubori and T. Suzumura, "HTML Templates that Fly: A Template Engine Approach to Automated Offloading from Server to Client," *Proc. 18th Int'l Conf. World Wide Web* (WWW 09), ACM Press, 2009, pp. 951–960.
3. E. Benson et al., "Sync Kit: A Persistent Client-Side Database Caching Toolkit for Data Intensive Websites," *Proc. 19th Int'l Conf. World Wide Web* (WWW 10), ACM Press, 2010, pp. 121–130.
4. T.J. Parr, "Enforcing Strict Model-View Separation in Template Engines," *Proc. 13th Int'l Conf. World Wide Web* (WWW 04), ACM Press, 2004, pp. 224–233.
5. S. Kojarski and D.H. Lorenz, "Domain-Driven Web Development with WebJinn," *Companion of the 18th Ann. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA 03), ACM Press, 2003, pp. 53–65.

latest available statistics show that 95 percent of browsers have JavaScript enabled (http://w3schools.com), and that 89.52 percent of the 10,000 top websites use JavaScript (http://trends.builtwith.com/docinfo/Javascript), certain types of devices have limited JavaScript support (phones, assistive technologies, search bots, and so on). BST's unavoidable JavaScript-based processing model thus has negative consequences on Web accessibility[9] and page semantics.

In the former case, the intensive use of JavaScript that characterizes BST doesn't 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 isn't an enhancement but an essential ingredient. We can't expect any kind of graceful degradation using a BST template in a browser without JavaScript. The page simply crashes, constituting a priority 1 WCAG violation. Although WCAG 2.0[10] and specifications such as the Web Accessibility Initiative – Accessible Rich Internet Applications (WAI-ARIA)[11] suite are more open-minded than WCAG 1.0 in reference to using technologies such as JavaScript, criteria imposed by version 1.0 will prevail in the middle term, given that governmental regulations inspired mainly by this version can't be adapted immediately.

While accessibility is an important issue, an even more devastating argument is that BST isn't semantically crawlable. When search robots (which don't process JavaScript) reach BST documents, they don't find the entire semantic payload that the page must show to users. They see only a mixture of HTML plus BST code that they don't know how to interpret, let alone index. The dynamic content data – that is, the view's model – are confined inside script blocks, which Web crawlers usually ignore. This is a relevant factor with direct economic impact. Consider that much of an online company's success depends on its potential customers finding its products using a search engine. Using BST would negatively affect such products' visibility.

Unfortunately, both downsides are insurmountable barriers that hinder BST systems' adoption for building complete Web applications and apparently relegate them for use only in Ajax. We can't do anything to turn a BST page into an accessible and semantically crawlable document, but we want to benefit from BST's undeniable upsides, which speed up development and improve performance. Our BST system, Yeast-Templates, solves this seemingly intractable dilemma.

Providing Accessibility to BST

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

This approach boils down to detecting the client's JavaScript processing capacity and, if it's disabled, providing a version of the page without BST. In fact, this is in the spirit of WCAG 1.0 checkpoint 6.3.[12] But Yeast-Templates goes a step further. First, it automates the generation of a page's alternative accessible version, avoiding desynchronization during maintenance. Second, it avoids the requirement for an explicit link to the accessible version.

Although the idea seems simple, implementing it is tricky. We must extend our server-side BST infrastructure, the Yeast-Server API, with a component that encapsulates a server-side JavaScript engine responsible for processing the Yeast templates. We 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 for processing the templates' document object model (DOM), unlike browsers, which do include this capability. Consequently, we can't process DOM-manipulation-based BST systems on the server. To avoid DOM manipulation, we needed to develop an alternative compiler/processor for Yeast-Templates.

Yeipee decomposes Yeast templates into a set of fragments that might 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; Yeipee adds HTML fragments directly to the response.

By removing the BST code and returning only HTML, Yeipee processors also remove all the obstacles to accessibility. However, this processing mode is perceptibly slower than the original (see the "Yeast-Templates Benchmarking" sidebar) and incompatible with the caching strategies that characterize BST. We're losing the performance benefits that BST provides.

Fortunately, Yeast-Server needn't always resort to Yeipee processing. We devised a progressive enhancement mechanism for using Yeipee processors transparently and only when strictly necessary. By default, Yeast-Server processes templates using Yeipee. But the processed template returned to the client now embeds a little script that detects whether JavaScript is enabled. If so, the browser attaches a parameter (`yst.yeipee=OFF`) in every request to disable server Yeipee processing. The script tries to set up a session cookie to carry that parameter, but if cookies are disabled, the script registers `onClick` and `onSubmit` event handlers to the page links and forms, respectively. When Yeast-Server detects that a certain request includes that parameter with that value, it changes its processing mode, omitting Yeipee processing and returning Yeast-Templates content. If the user agent isn't JavaScript enabled, then the enhancement script won't execute, the parameter won't be included, and Yeipee will process the request to the server. 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.

We want to emphasize that our strategy delivers accessible content by default, and its application is transparent to users and development teams, designers, and application programmers. The overall result is that, without being penalized with any extra tasks, developers can take advantage of the BST development philosophy, which gives them independence and agility in development, while applications continue to benefit from BST performance improvements, relinquishing them only when it's unavoidable. BST, together with our approach for accessibility support, constitutes a true option for use as a core technology for template systems, even when accessible applications are needed.

References

1. T.J. Parr, "Enforcing Strict Model-View Separation in Template Engines," *Proc. 13th Int'l Conf. World Wide Web* (WWW 04), ACM Press, 2004, pp. 224–233.
2. M. Rabinovich et al., "Moving Edge-Side Includes to the Real Edge: The Clients," *Proc. 4th Usenix Symp. Internet Technologies and Systems* (USITS 03), Usenix Assoc., 2003, p. 12.
3. M. Mahemoff, *Ajax Design Patterns*, O'Reilly Media, 2006.
4. F. García-Izquierdo, R. Izquierdo, and A. Juan Fuente, "A Double-Model Approach to Achieve Effective Model-View Separation in Template-Based Web Applications," *Web Eng.*, LNCS 4607, Springer, 2007, pp. 442–456.
5. H. Böttger, A. Møller, and M.I. Schwartzbach, "Contracts for Cooperation between Web Service Programmers and HTML Designers," *J. Web Eng.*, vol. 5, no. 1, 2003, pp. 68–89.
6. J. Ravi, Z. Yu, and W. Shi, "A Survey on Dynamic Web Content Generation and Delivery Techniques," *J. Network and Computer Applications*, vol. 32, no. 5, 2009, pp. 943–960.
7. M. Tatsubori and T. Suzumura, "HTML Templates that Fly: A Template Engine Approach to Automated Offloading from Server to Client," *Proc. 18th Int'l Conf. World Wide Web* (WWW 09), ACM Press, 2009, pp. 951–960.
8. C. Yue and H. Wang, "Characterizing Insecure JavaScript Practices on the Web," *Proc. 18th Int'l Conf. World Wide Web* (WWW 09), ACM Press, 2009, p. 961.
9. W. Kern, "Web 2.0: End of Accessibility? Analysis of Most Common Problems with Web 2.0-Based Applications Regarding Web Accessibility," *Int'l J. Public Information Systems*, vol. 4, no. 2, 2008, pp. 131–154.
10. M. Ribera et al., "Web Content Accessibility Guidelines 2.0: A Further Step Towards Accessible Digital Information," *Program: Electronic Library and Information Systems*, vol. 43, no. 4, 2009, pp. 392–406.
11. B. Gibson, "Enabling an Accessible Web 2.0," *Proc. Int'l Cross-Disciplinary Conf. Web Accessibility* (W4A 07), ACM Press, 2007, pp. 1–6.
12. *Web Content Accessibility Guidelines 1.0,* World Wide Web Consortium (W3C) recommendation, May, 1999; www.w3.org/TR/WCAG10/.

Francisco J. García-Izquierdo is an assistant professor at the Universidad de La Rioja, Spain. His research interests include Web engineering, Web accessibility, modeling theories, and computer science teaching. García-Izquierdo 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.