Client-server Web applications widgets*

Vincent Balat
Univ Paris Diderot – Sorbonne Paris Cité – PPS, UMR 7126 CNRS, Inria – Paris, France
vincent.balat@ univ-paris-diderot.fr

ABSTRACT

The evolution of the Web from a content platform into an application platform has raised many new issues for developers. One of the most significant is that we are now developing distributed applications, in the specific context of the underlying Web technologies. In particular, one should be able to compute some parts of the page either on server or client sides, depending on the needs of developers, and preferably in the same language, with the same functions. This paper deals with the particular problem of user interface generation in this client-server setting. Many widget libraries for browsers are fully written in JavaScript and do not allow to generate the interface on server side, making more difficult the indexing of pages by search engines. We propose a solution that makes possible to generate widgets either on client side or on server side in a very flexible way. It is implemented in the Ocsigen framework.

Categories and Subject Descriptors

D.3.3 [**Software**]: Programming languages—*Language Constructs and Features*; H.5.3 [**Information Systems**]: Information interfaces and presentation—*Group and Organization Interfaces* Web-based interaction

General Terms

Languages, Reliability

Keywords

JavaScript, Web applications, mobile applications, GUI

1. INTRODUCTION

The time of regular desktop applications is gone. It is now rare to design a modern application without thinking about a storage in the cloud, and a mobile version, with shared data. The time is also gone of static Web sites or Web sites as basic front-ends to databases. Users now expect a high level of interaction with modern ergonomics and user-friendliness. The frontier between applications and sites is narrowing. It's even more true for mobile applications for which the use of Web technologies (HTML5) is the simplest way to ensure portability.

This convergence is a challenge for Web framework developers. Most of these modern Web applications are still written with old technologies that were not designed for this. A lot of libraries have been designed to make this work easier, but most of the time they do not give a complete solution for the whole problem of distributed Web-based programs. Some JavaScript libraries tend to mimic traditional desktop graphical user interface libraries, without taking into account the specificities of the Web, nor the evolution in graphic design and ergonomics.

Taking the problem as a whole first requires to use the same language for implementing both client and server parts of the application. This makes a lot more than just removing the need for developers to master several languages. It also makes disappear the impedance mismatch between the two sides. No need to use conversion functions to transmit data. You can also share libraries and use the exact same functions on both sides without having to implement them twice.

Very few frameworks give the ability to use the same language on both sides. The main difficulty is that it is very difficult to make a new language or virtual machine available in all browsers, and relying on a plugin is not a good solution for inter-operability. One solution is to use JavaScript on server side, as NodeJS is doing [11]. Another solution is to make possible to execute programs written in your language through JavaScript, either by implementing and interpreter in JavaScript (or a byte-code interpreter [4]), or by compiling to JavaScript [9, 6, 14, 4, 15].

The Ocsigen framework [2, 1] experimented the two last solutions and is now mainly relying on a compiler to JavaScript [15]. We chose the OCaml language because we believe that the kind of complex programs we want to implement deserves a modern expressive language with sophisticated typing features, in order to help the developer to reduce the number of bugs. Ocsigen is using an extension of OCaml providing a way to distinguish between parts of the code to be executed on one side or another, or even to share some parts between both sides.

This opens a new range of possibilities for the developers. Whereas the side where to execute some computation was

*Work partially supported by the French national research agency (ANR), PWD project, grant ANR-09-EMER-009-01, and performed at the IRILL center for Free Software Research and Innovation in Paris, France

heavily constrained (if not imposed) by the languages or libraries, the model we advocate gives the freedom of choice for the developer. For example he is now free to generate some parts of the page on server side or on client side. This is really useful for example when you want your Web site to be indexed by search engines (which requires to generate HTML on server side), but also want to add dynamically new content on browser side, for example from data entered by a user.

But more than one same language for both sides, the model implemented in Ocsigen provides a way to design the full Web application as a single program, with some parts being executed on server side, others on client side. Thus the developer does not need anymore to think about defining a way to communicate between the two parts. As it is the same program, you can just use the same variable. For example Ocsigen makes possible to give a name to some HTML element on server side and use this name on client side to refer the element without having to rely on manually inserted HTML identifiers and the `getElementById` function.

This way of programming Web applications is a step forward in the convergence between Web sites and applications, because it makes possible to keep the best of the two worlds: the program is still a Web site, and can benefit from the traditional Web interaction features, like URLs, forms, bookmarks, back button, multi-tabs. But it also behaves like a desktop application in term of features, responsiveness and ergonomics. Ocsigen makes possible for example to have a program running in your browser which is persistent across page changes through links or forms.

The first industrial users of our framework witness that the freedom gained by the developers has a great impact on the development time. But these new possibilities require to rethink the way interface libraries are written, in order to make them flexible enough and not to impose unnatural limitations. It is for example noticeable that a lot of existing JavaScript user interface libraries have not been designed in a way that make them usable in this context.

In next section, we will briefly describe the two-level language Ocsigen is using for programming client-server Web applications. Then we will describe the problem that shows up when designing client-server widgets, and the solution we propose, with some examples.

2. A TWO-LEVEL PROGRAMMING LANGUAGE FOR WEB APPLICATIONS

2.1 Client and server sections, injections

Ocsigen's syntax is the usual OCaml syntax, with some extra constructs to distinguish between client, server, and shared (duplicated on both sides) parts.

The three main syntactical constructs are `{server{ ... }}` `{client{ ... }}` and `{shared{ ... }}` used for enclosing respectively server side, client side, and shared code (that we want on both sides).

It is possible to use server side values in client side code, by prefixing their name with a `%`. We call this an *injection*. Injections are send together with the page and correspond to the server side values at the moment when the piece of code containing then `%` is executed on server side. A typical use of this is when you want to define a communication bus between client side processes. It is defined on server side and all clients can listen or write on the bus just by using the same variable name.

During compilation, server and client parts are extracted into two separated program, one being compiled to native code to be executed on the server, and the other one being compiled to JavaScript. Shared parts are duplicated. Some type information are automatically added in both parts to ensure that values types match on both sides.

2.2 Client values

The additional double brackets syntax `{{ ... }}` makes possible to define client side values inside server parts. For example, the following code defines a client side value that is referred on server side through name `v`.

```
1 {server{
2   let v = {{ "some␣string" }}
3 }}
```

Here, the value `v` has type `string client_value` on both sides. This type is abstract on server side and it is impossible to access the value. Besides, knowing the actual client side value may require some computation that will be performed on client side next time something is send to the client (either a page or some OCaml value).

The only way to use client values is to use the `%` syntax on client side, for example: `{{ %v }}`. On client side, type `'a client_value` is just `'a` (where `'a` is any type). Here `%v` has type `string`.

It is possible to define client values of any types, even functions, and they can require any kind of computation, even with side effects. Incidentally, it is used very often just to ask the client to execute some piece of code.

3. CLIENT-SERVER USER INTERFACES

3.1 The problem of client server widgets

Most user interface libraries, like Google Closure [8], GWT [9] or Enyo [7], define their widgets programmatically. That is, you call a JavaScript function that will create the DOM elements to be inserted in the page. This makes impossible to generate the widgets before sending the page. It is acceptable when your program is mostly client side, or if you use these widgets only for limited parts, but not if you want the content to be indexed by search engines.

Some other libraries depart from the traditional widget style by separating the creation of the element from the computational part (for example binding events). This makes possible to generate the page fully on server side, and change the behaviour of DOM elements on client side after the page is loaded. In that extend they are closer to the needs of Web developers. This is the case for example of JQuery [10].

3.2 User interfaces generated on server side

The syntax we described in previous section makes possible to define server side generated widgets in a straightforward manner. While generating the HTML element, just use a client value to ask the client side program to do some side effects on it after loading the page (for example bind some mouse event or call JQuery methods).

Here is an example of a page using client server widgets:

```
1 {server{
2   let my_box () =
3     let a = div [pcdata "Hello"] in
```

```
4       let b = div [pcdata "Click me"] in
5       let jqa = {{ JQuery.jQuery %a }} in
6       let _ = {{ lwt ev = click %b in
7                  %jqa##slideDown() }} in
8       div [a; b]
9   }}
```

This example defines a server side function `my_box` that returns a `<div>` element, containing two other `<div>` elements, called `a` and `b`. Together with this page, we ask the client side program to execute two pieces of client side code, defining two client values. The first one is computed by calling the JQuery function `jQuery` (also called `$` in JQuery). The result is a client value, referred through name `jqa` on server side.

The second one asynchronously waits for a mouse click on element `b` and then calls JQuery method `slideDown` on `jqa`[1].

It is convenient to use OCaml objects for widgets. The instanciation of the object is done through a client value section. It is possible to save the object as a new property of the DOM element.

3.3 Client-server user interfaces

In order to make possible to create widgets either on server or client sides, we extended the client value syntax to shared sections. While compiling the client version of shared sections, client value syntax `{{ ... }}` is just ignored, whereas it is kept on server side.

While doing that, one must be careful about two things:

- The computational part of a widget may require to wait for the element to be inserted in the page before being executed. This can be done easily by using the Ocsigen equivalent of callbacks (using the Lwt library).

- In the client version of shared sections, the `%` syntax may refer either to a server value defined before the shared section or a client value defined in the shared section (in which case the `%` must be ignored). Distinguishing between the two cases is left to the programmer for now. We introduced another syntax `%%` for this. But we hope to be able to detect the case automatically in the future.

4. CONCLUSION

Ocsigen's client server syntax makes very easy to write sophisticated Web applications in very few lines of code. It removes the impedance mismatch between server and client code, makes unnecessary to convert data before sending it to the other side, and removes in most cases the need for calls to `getElementById`. But one on the most interesting features, which is described in this paper, is the ability to define client-server widgets. First you can define your widgets on server side and ask the client side program to initialize it. Secondly, it is even possible to define a widget that will be usable either on server or client side.

The use of the OCaml language provides a lot of very expressive features like sum types, pattern matching, objects, or type inference to make programming very enjoyable and concise. An advanced use by Ocsigen of its powerful typing system helps the programmer to remove a very large set of bugs at compile time (for example a program that may generate a page that does not conform to the recommendations of the W3C is rejected!).

Very few other frameworks are proposing to write client-server Web applications as a single program. One noticeable exception is the Hop language [14], derived from Scheme, and also the Links research prototype [5] or the Dart [6] language. One can also cite JavaScript frameworks like NodeJS [11] or Opa [12]. None of them focus on the server side as we do, especially in the goal of achieving the full convergence between Web sites and applications. Hop has the closest client server syntax, but without client-server widgets.

This paper describes mainly the last features of Ocsigen, that have been introduced in version 3 (December 2012). The goal of the project is not only to write a research prototype but to make a complete framework, usable for real Web applications. Despite its young age, it is already used in industry. The most advanced project is the Besport [3] social network. But more and more projects are choosing Ocsigen (like Pumgrana [13] or XPrime [16]). Ocsigen is also used for many traditional Web sites without advanced client side features (which corresponds to the features of previous versions of the framework).

[1] The `lwt` syntax means that the `click` function is a thread that waits until a mouse click happens. Then it returns a DOM event called `ev`. The `##` syntax is used to call JavaScript methods from OCaml.

5. REFERENCES

[1] V. Balat, P. Chambart, and G. Henry. Client-server Web applications with Ocsigen. In *WWW2012 dev track proceedings*.

[2] V. Balat, J. Vouillon, and B. Yakobowski. Experience report: ocsigen, a web programming framework. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*

[3] Besport. `http://www.besport.com/`.

[4] B. Canou, E. Chailloux, and J. Vouillon. How to Run your Favorite Language in Web Browsers. In *WWW2012 dev track proceedings*.

[5] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *In 5th International Symposium on Formal Methods for Components and Objects (FMCO)*. Springer-Verlag, 2006.

[6] Dart. `http://www.dartlang.org/`.

[7] Enyo. `http://enyojs.com/`.

[8] Google closure library. `http://code.google.com/intl/en/closure/library/`.

[9] Google web toolkit. `http://code.google.com/webtoolkit/`.

[10] Jquery. `http://jquery.com/`.

[11] Nodejs. `http://nodejs.org/`.

[12] Opa. `http://www.opalang.com/`.

[13] Pumgrana. `http://www.pumgrana.com/`.

[14] M. Serrano, E. Gallesio, and F. Loitsch. Hop, a language for programming the web 2.0. In *Dynamic Languages Symposium*, Oct. 2006.

[15] J. Vouillon and V. Balat. From bytecode to javascript: the js_of_ocaml compiler. in journal *Software: Practice and Experience*, 2013.

[16] Xprime. `http://www.ocsigenlabs.com/xprime/`.