Client-server Web applications with Ocsigen[*]

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

Pierre Chambart
CNRS, PPS, UMR 7126
Univ Paris Diderot
Sorbonne Paris Cité
Paris, France
pierre.chambart@
crans.org

Grégoire Henry
CNRS, PPS, UMR 7126
Univ Paris Diderot
Sorbonne Paris Cité
Paris, France
gregoire.henry@
pps.univ-paris-diderot.fr

ABSTRACT

The Ocsigen framework offers a new way to develop sophisticated client-server Web applications. It makes it possible to write as a single program both the server and client sides of a Web application, thus simplifying a lot communications and data transfers, and avoiding code duplications. It also proposes a wide set of high level concepts to program traditional Web interactions in a very concise way while mixing them seamlessly with client side features. The use of a powerful type system improves a lot the reliability of programs, reducing debugging time, and making the code easier to maintain.

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

Continuation-based Web programming, AJAX, Javascript

1. INTRODUCTION

Recent Web browsers have become powerful virtual machines in which more and more programs are executed. Desktop programs are currently being superseded very quickly by ever more sophisticated cloud applications.

Today, most Web developers are using distinct languages for programming traditional Web interaction (e.g. forms, links and bookmarks) and client side features. This creates an impedance mismatch that often requires explicit data conversions and code duplications, all of which being error prone. For instance, data validation should be performed typically both on the client for early feed-back and on the server for safety. This approach also makes it difficult to combine in a flexible and natural way client and server features. As a result, client features remain often restricted to small one page applications with no persistence when links are followed.

Even though there are good libraries that attempt to address these problems, we believe that rethinking programming tools could speed up the rise of applications taking full advantage of the new possibilities offered by modern browsers.

A few programming frameworks now propose to use the same language both client and server side [4, 7, 2]. On the client, they must rely on what browsers provide, that is, basically Javascript. The good news is that there now exists good and fast Javascript engines, available on servers as well [5].

The Ocsigen project [1] takes the alternative approach of using Javascript as the target language of a compiler. Indeed, we want to use advanced language features not available in Javascript. The goal is to abstract away technical details, proposing high level technical concepts corresponding closely to the very needs of Web developers. This makes it possible to express sophisticated behaviors in very few lines.

One of the most powerful features of Ocsigen is the way Web interaction, such as the effect of following links, is programmed through a unique concept of "services", and remains automatically compatible with client side features: the client side program persists while the user browses the Web site, and traditional features like bookmarks, forms, sessions or "back button" automatically remain available, in a transparent way.

Finally Ocsigen is making extensive use of the powerful type system of the underlying language, OCaml, to detect at compile time many programming errors. For example it makes it impossible to generate invalid HTML. All this reduces a lot debugging and testing time!

We start in section 2 by showing an example of application written with Ocsigen. Section 3 shows the basics for writing a client-server application as a single program, whereas the way to program traditional Web interaction is sketched in section 4. Then, section 5 shows how we benefits from static typing.

[*]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

2. EXAMPLE

Figure 2 shows the full source code of a client-server drawing application (see also a screen-shot in figure1, and a working version at URL `http://ocsigen.org/graffiti`). This program allows the user to draw on the screen, choose the color of the brush, and see in real time what other users are drawing.

We won't go into the full details of the code, which is basically plain OCaml. For more explanations, there is a tutorial showing step by step how to write this program at URL `http://ocsigen.org/tutorial`.

The first thing to notice is that the code is short. And this is due not to the use of specialized libraries that would do everything for us (we are using a library only for the color picker) but rather to the expressiveness of concepts.

As you can see, the code is divided into several parts with special bracket to distinguish between client code, server code, and shared code. It mainly defines one *service*, corresponding to a URL and returning an HTML5 page, together with some code to be executed once the page is loaded.

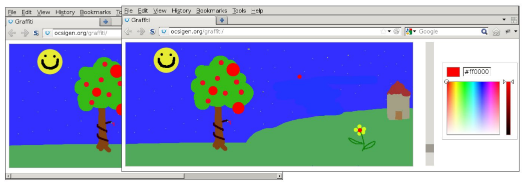

Figure 1: The Graffiti program with two windows

3. CLIENT SERVER WEB APPLICATIONS

3.1 General principles

Writing client and server code together brings many advantages beside the use a single language. It allows a seamless communication of data between both parts of the program. Server side variables can be used from the client (see line 37); their values are sent along the page. This also ensures that client and server codes remain consistent. For instance, in this example, if the color of the brush were removed from the messages sent on the bus (on line 33), the compiler would complain if the remaining of the code were not updated accordingly, and it would point to the location of the error.

Interfaces are built in the usual declarative way, and sent together with the code corresponding to dynamic parts. Programs are written in way very similar to desktop applications. For example a very simple interface allows to use user variables server side as if there was a single user program.

3.2 The Js_of_ocaml compiler

The client side code is translated from OCaml to Javascript by the Js_of_ocaml compiler. A syntax extension makes it possible to directly invoke Javascript methods (line 23), set object properties (line 19) or create objects (line 22). This makes it easy to bind Javascript libraries (like Google Closure in our example [3]). Appropriate types can be used to ensure that objects are not misused. The compiler performs dead code elimination. Hence, the programmer does not have to restrict himself to minimal libraries.

3.3 Persistent client side programs

Ocsigen allows to combine the best of traditional and persistent applications. It is possible to write persistent applications in a classical declarative way. Only the reactive parts involves using DOM interfaces. This way, users with Javascript disabled and search engine crawlers can still browse the site and pages have real bookmarkable URLs.

When loading a page, links and forms are modified such that clicking links appears to work as usual, but does not in fact interrupt the Javascript program. This feature makes it possible to write very easily applications that need to keep a state during the navigation, such as an audio library with a music player, or persistent widgets (ex. a chat).

Having these features with other frameworks usually requires code duplication to implement a Javascript-less version of the site.

4. SERVICE BASED PROGRAMMING

Programming traditional Web interaction is done through the notion of *service*. A service is a kind of function invoked in reaction to an HTTP request, and that usually returns a Web page. Main services are identified by the URL they are attached to. Ocsigen proposes many other ways to identify a service: based on special parameters, on the HTTP method, on the names of parameters, on the current session, etc.

As an example, suppose you want to implement a login box on each page of your Web site. If a user logs in from a page, you want him to remain on this page after connection. With Ocsigen, you just need to create an extra service, not attached to a precise URL (that is: available from any URL), and that will perform the action of checking the password and registering session data. This only takes a few lines of code.

Another very convenient feature is the ability to dynamically create new services, for example services customized to one user and that depend on previous interactions with him. This simplifies a lot programming sequences of pages, each depending on the previous ones (like buying a plane ticket). This is usually known as *continuation based Web programming* [6].

5. STATIC TYPING

Simply put, static typing is the principle of verifying at compile time that values are correctly used. It can provides some basics properties like ensuring that variables are declared and initialized. But, when the type system is powerful enough, it can also protect against more complex programming errors. With a powerful type inference mechanism, type safety can be combined with the lightweightness and flexibility of dynamic languages.

In Ocsigen we leverage the power of OCaml type system to enforce many properties at compile time. We can for instance ensure that Javascript APIs are correctly used, that database accesses are safe, that links go to existing services with the right parameters. With the whole application written as a single code base, we can moreover ensure that communications between client and server can't go wrong.

One of the most impressive use of static typing in Ocsigen is for checking at compile time the validity of function generating HTML. This makes it impossible for an Ocsigen program to generate a page that does not respect the W3C recommendations!

```
 1 {shared{                     (* Code shared by client and server, defining constants and the type of messages *)
 2   let width, height = 700, 400
 3   type messages = (string * (int * int) * (int * int)) deriving (Json)
 4 }}
 5
 6 let b = Eliom_bus.create ~name:"graff" Json.t<messages>                       (* The bus used for communication *)
 7
 8 {client{
 9   let draw ctx (color, (x1, y1), (x2, y2)) =                   (* Client side function for drawing on a canvas *)
10     ctx##strokeStyle <- (string color);
11     ctx##beginPath(); ctx##moveTo(float x1, float y1); ctx##lineTo(float x2, float y2); ctx##stroke()
12 }}
13
14 let main_service = My_appl.register_service ~path:[""] ~get_params:Eliom_parameters.unit
15   (fun () () ->                                          (* The main service, at URL "/", with no parameter *)
16     onload                          (* piece of code to be executed in the browser after loading the page *)
17       {{ let canvas = Dom_html.createCanvas Dom_html.document in                                (* the canvas *)
18          let ctx = canvas##getContext (Dom_html._2d_) in
19          canvas##width <- width; canvas##height <- height;
20          Dom.appendChild Dom_html.document##body canvas;
21
22          let pSmall = jsnew Goog.Ui.hsvPalette (null, null, some (string "goog-hsv-palette-sm")) in
23          pSmall##render(some Dom_html.document##body);                                    (* the color palette *)
24
25          let x, y = ref 0, ref 0 in                                                    (* computing coordinates *)
26          let set_coord ev =
27            let x0, y0 = Dom_html.elementClientPosition canvas in
28            x := ev##clientX - x0; y := ev##clientY - y0 in
29          let compute_line ev =
30            let oldx = !x and oldy = !y in
31            set_coord ev;
32            let color = to_string (pSmall##getColor()) in
33            (color, (oldx, oldy), (!x, !y))
34          in
35          let line ev =
36            let v = compute_line ev in
37            let _ = Eliom_bus.write %b v in                                              (* writing on the bus *)
38            draw ctx v
39          in
40          Lwt_stream.iter (draw ctx) (Eliom_bus.stream %b);                     (* Reacting to events from bus *)
41          run (mousedowns canvas                                         (* For each mousedown on the canvas *)
42                       (arr (fun ev -> set_coord ev; line ev)                                    (* draw a dot *)
43                        >>> first [mousemoves Dom_html.document (arr line);
44                                   mouseup Dom_html.document >>> (arr line)])) ();
45        }};                                                           (* Then for each mousemove draw a line, *)
46                                                   (* and if mouseup draw a line and start again catching mousedowns *)
47
48     return << <html> <head> <title>Graffiti</title>
49                             <link rel=stylesheet href="./css/style.css"/>
50                             <script src="./oclosure.js"></script> </head>
51                      <body> <h1>Graffiti</h1></body> </html> >>
```

Figure 2: Full source code of the Graffiti program

Typing is useful to ensure code safety, but it also helps a lot in maintaining and refactoring code. When some parts of the code base is modified, the compiler can show the programmer where he forgot to propagate the changes.

6. CONCLUSION

The Ocsigen software is now mature and full featured. It has a growing community of users and is released under an open source licence (LPGL). Ocsigen provides the most advanced features of Web programming frameworks (like dynamic services or unified client-server programming) but also many unique features: service identification mechanism, static checking of HTML, persistence of the client program, etc. These unique features come to a large extent from the use of one of the most advanced programming language (OCaml), known to be very fast, expressive, concise and safe, with many robust libraries.

Many further features have not been presented in this paper: an advanced session mechanism, a very concise way to implement event handlers (see lines 41-44), or a flexible interface for server push events (see lines 6, 37, 40). Ocsigen also guarantee the absence of code injection on the client, the server, nor in SQL queries, and provides mechanisms to protect angaisnt XSS and data leaks. In our experience, the safety provided by static typing helps a lot in reducing development time.

Acknowledgements

Many acknowledgements are due to Jérôme Vouillon, Raphaël Proust, Benjamin Canou and Emmanuel Chailloux.

7. REFERENCES

[1] 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*, pages 311–316, Edinburgh, Scotland, 2009. ACM.

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

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

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

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

[6] C. Queinnec. Continuations and web servers. *Higher-Order and Symbolic Computation*, 17(4):277–295, Dec. 2004.

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