Ocsigen

Reactive client-server
Web applications

with Js_of_ocaml and Eliom

Vincent Balat — Hugo Heuzard
OUPS meetup — 9 December 2014
The project

created in 2004, more than 200 000 l.o.c., LGPL

**People:**

Vincent Balat, Jérôme Vouillon,  
Pierre Chambart, Grégoire Henry, Benedikt Becker, Benjamin Canou, Boris Yakobowski,  
Jérémye Dimino, Gabriel Radanne, Hugo Heuzard,  
Romain Calascibetta, Charly Chevalier, Enguerrand Decorne, Jacques-Pascal Deplaix, Grégoire Lionnet,  
Raphaël Proust, Stéphane Glondu, Jérôme Maloberti, Gabriel Kerneis, Arnaud Parant, Christophe  
Lecointe, Denis Berthod, Gabriel Cardoso, Piero Furiesi, Jaap Boender, Baptiste Strazzulla, Thorsten  
Ohl, Gabriel Scherer, Séverine Maingaud, Simon Castellan, Jean-Henri Granarolo, Archibald Pontier,  
Nataliya Guts, Cécile Herbelin, Charles Oran, Jérôme Velleine, Pierre Clairambault ...

---

université PARIS DIDEROT  
cnrs  
ANR  
IRILL  
Inria
Some companies and open source projects:
BeSport, NYU CGSB Genomics Core, Pumgrana, Facebook, Cumulus, Life.tl, Ashima Arts, Metaweb/Freebase, Hypios, Ocamlcore, Ocamlpro, Nleyten...
Some of our projects

- **eliom**: Write client/server Web applications in very few lines of OCaml code.
- **js_of_ocaml**: An OCaml to Javascript compiler.
- **lwt**: A cooperative threading library for OCaml.
- **server**: A full-featured and extensible Web server.

> And many other projects ...
Recap of last episode

1. Running OCaml programs in a Browser (Ocsigen Js_of_ocaml)
Recap of last episode

1. Running OCaml programs in a Browser
2. Interact with the DOM and JS libraries (Syntax `value##method(a, b), etc.`)
Recap of last episode

1. Running OCaml programs in a Browser
2. Interact with the DOM and JS libraries
3. Static typing of HTML
Recap of last episode

1. Running OCaml programs in a Browser
2. Interact with the DOM and JS libraries
3. Static typing of HTML
4. Calling typed services
Recap of last episode

1. Running OCaml programs in a Browser
2. Interact with the DOM and JS libraries
3. Static typing of HTML
4. Calling typed services
5. Server side: Writing services in OCaml
Recap of last episode

1. Running OCaml programs in a Browser
2. Interact with the DOM and JS libraries
3. Static typing of HTML
4. Calling typed services
5. Server side: Writing services in OCaml
6. Client-server applications: sections
   Syntax:

```
{server{ ... 
}}
{client{ ... 
}}
{shared{ ... 
}}
```
Recap of last episode

1. Running OCaml programs in a Browser
2. Interact with the DOM and JS libraries
3. Static typing of HTML
4. Calling typed services
5. Server side: Writing services in OCaml
6. Client-server applications: sections
7. Client-server communication: server values in client code, RPCs
Recap of last episode

1. Running OCaml programs in a Browser
2. Interact with the DOM and JS libraries
3. Static typing of HTML
4. Calling typed services
5. Server side: Writing services in OCaml
6. Client-server applications: sections
7. Client-server communication: server values in client code, RPCs
8. Client values: defining client values from server side

let cv = {{ f x }} in
A demo of some features of Eliom, presented as a tutorial. With some upcoming features:

→ Eliom base app
→ Reactive DOM
→ Client-server reactive programming
→ Client-server cache of data
→ Notifications

The application: a very basic forum
http://ocsigen.org/tuto/manual/tutoreact
Eliom Base App

High level libraries for Eliom:

- **User management** (registration, activation links, lost password, change password, upload picture, group of users ...)
- **Notifications** to users (new messages, etc)
- **Tips** for new users
- **Dates** and time zones
- ...

Status: beta, not fully documented

+ a **template** for quickly building applications with users:

```
eliom-distillery -name tutoreact -template eba.pgocaml
```
I will not show the following functions:

```ocaml
module Db : sig
  val get_messages : unit -> int list Lwt.t
  val get_message : int -> string Lwt.t
  val add_message : string -> int Lwt.t
end
```

implemented on server side.

Use the DB you prefer. PGOCaml is a good choice.
let display_messages userid_o =
  lwt messages = Db.get_messages () in
  lwt l = Lwt_list.map_s
    (fun id -> lwt msg = Db.get_message id in
     Lwt.return (li [pcdata msg]))
    messages
  in
  Lwt.return (ul l)
Adding new messages

Add an input in the page:

```ocaml
let inp = Raw.input ~a:[a_input_type `Text] () in
```
Adding new messages

Add an input in the page:

```ocaml
let inp = Raw.input ~a:[a_input_type `Text] () in
```

Make function `Db.add_message` accessible from client side:

→ server functions (RPC)

```ocaml
let add_message_rpc =
  server_function Json.t<string> Db.add_message
```
Adding new messages

Add an input in the page:

```ocaml
let inp = Raw.input ~a:[a_input_type `Text] () in
```

Make function `Db.add_message` accessible from client side:

→ server functions (RPC)

```ocaml
let add_message_rpc =
    server_function Json.t<string>
        (Eba_session.connected_rpc
            (fun userid value -> Db.add_message value))
```

`Eba_session.connected_rpc` wraps a function of type `'a -> 'b Lwt.t` into a function of type `int64 -> 'a -> 'b Lwt.t`, where the `int64` is the user id, and fails if user is not connected.
Adding new messages

Add an input in the page:

```ocaml
let inp = Raw.input ~a:[a_input_type `Text] () in
```

Make function `Db.add_message` accessible from client side:

```
let add_message_rpc =
  server_function Json.t<string>
  (Eba_session.connected_rpc
   (fun userid value -> Db.add_message value))
```

Bind the input:

```
let _ = {unit{
      let open Lwt_js_events in
      let inp = To_dom.of_input %inp in
      async (fun () -> changes inp (fun _ _ ->
        let value = Js.to_string (inp##value) in
        inp##value <- Js.string "";
        %add_message_message_rpc value))
    }}

```
Client server cache

We recommend to store the values used by the client program in a local database. New module `Eliom_cscache` implements this in a way that is compatible with client-server (and reactive) programming. On server side, the cache is implemented “with scope request”, avoiding to retrieving several times the same data from the DB during a request.

```ocaml
let cache = E_cscache.create ()
```

To retrieve some data, call `Eliom_cscache.find cache get_data key` from client or server side.
We recommend to store the values used by the client program in a local database. New module `Eliom_cscache` implements this in a way that is compatible with client-server (and reactive) programming. On server side, the cache is implemented “with scope request”, avoiding to retrieving several times the same data from the DB during a request.

```ocaml
let cache = E_cscache.create ()
```

To retrieve some data, call `Eliom_cscache.find cache get_data key` from client or server side.

Function `get_cache` must be implemented on both sides:

```ocaml
let get_data = Db.get_message
let get_data_rpc =
    server_function Json.t<int>
    (Eba_session.Opt.connected_rpc
     (fun userid_o id -> get_data id))

{client{
    let get_data id = %get_data_rpc id
}}
```
Reactive programming (using module React, by Daniel Bünzli)

```ocaml
let x, set_x = React.S.create 1
let pr_x = React.S.map print_int x
let () = List.iter set_x [2; 2; 3]
```
Reactive programming (using module React, by Daniel Bünzli)

```ocaml
let x, set_x = React.S.create 1
let pr_x = React.S.map print_int x
let () = List.iter set_x [2; 2; 3]
```

Reactive DOM (before Eliom 4.1)

```ocaml
R.ul (React.S.map f r1)
```
Reactive programming (using module React, by Daniel Bünzli)

```ocaml
let x, set_x = React.S.create 1
let pr_x = React.S.map print_int x
let () = List.iter set_x [2; 2; 3]
```

Reactive DOM (before Eliom 4.1)

```ocaml
R.ul (React.S.map f rl)
```

Incremental changes (using module ReactiveData, by Hugo Heuzard)

```ocaml
R.ul (ReactiveData.RList.map f rd)
...
ReactiveData.RList.cons a (snd rd)
```

Reactive programming makes the update of an interface much easier.
Client-server reactive DOM

New modules SharedReactiveData and SharedReact, to make possible to generate reactive pages from server (or client) sides.

```ocaml
{shared{
  let display_message id =
    lwt msg = E_cscache.find %cache get_data id in
    Lwt.return (li [pcdata msg])
}}

let display_messages () =
  lwt messages = Db.get_messages () in
  let rmessages = SharedReactiveData.RList.make messages in
  lwt content = SharedReactiveData.RList.Lwt.map_p
    {shared{ display_message }}
    (fst rmessages)
  in
  Lwt.return (R.ul content)
}
Notify clients of changes

Define a notification module for each type of data:

```ocaml
module Msg_notif = Eba_notif.Make(struct
  type key = ... (* The type of the data identifiers *)
  type notification = ... (* The type of notifications *)
end)
```

Reacting to notifications:

```ocaml
React.
E.
map
(handle_notif %
rmessages %(
Msg_notif.
client_ev()))
```

Where `handle_notif` is defined by:

```ocaml
let handle_notif rmessages (_, msgid) =
  SharedReactiveData.
RList.
cons
msgid
(snd
rmessages)
```

To declare that I'm a client listening on data `i`:

```ocaml
Msg_notif.
listen i;
```

To notify clients listening on data `i`:

```ocaml
Msg_notif.
notify i (fun userid -> Lwt.
return (Some userid));
```

In this example, `i` is `()`, type `key` is `unit`, and type `notification` is `int`.
Notify clients of changes

Define a notification module for each type of data:

```ocaml
module Msg_notif = Eba_notif.Make(struct
    type key = ... (* The type of the data identifiers *)
    type notification = ... (* The type of notifications *)
end)
```

Reacting to notifications:

```ocaml
React.E.map (handle_notif %rmessages) %(Msg_notif.client_ev ()
```

Where `handle_notif` is defined by:

```ocaml
let handle_notif rmessages (_, msgid) =
    SharedReactiveData.RList.cons msgid (snd rmessages)
```
Notify clients of changes

Define a notification module for each type of data:

```plaintext
module Msg_notif = Eba_notif.Make(struct
type key = ... (* The type of the data identifiers *)
type notification = ... (* The type of notifications *)
end)
```

Reacting to notifications:

```plaintext
React.E.map (handle_notif %rmessages) %(Msg_notif.client_ev ())
```

Where `handle_notif` is defined by:

```plaintext
let handle_notif rmessages (_, msgid) =
  SharedReactiveData.RList.cons msgid (snd rmessages)
```

To declare that I’m a client listening on data `i`:

```plaintext
Msg_notif.listen i;
```

To notify clients listening on data `i`:

```plaintext
Msg_notif.notify i (fun userid -> Lwt.return (Some id));
```

In this example, `i` is `()`, type `key` is `unit`, and type `notification` is `int`. 
Full code

```ocaml
module Msg_notif = Eba_notif.Make(struct
  type key = unit
  type notification = int
end)

let add_message_rpc = server_function Json.t
  <
    Eba_session.connected_rpc (fun userid value ->
      lwt id = Db.add_message value in
      Msg_notif.notify () (fun userid -> Lwt.return (Some id));
      Lwt.return ())

let get_data_rpc = server_function Json.t
  <
    Eba_session.Opt.connected_rpc (fun userid_o id -> get_data id)

let cache = (int, string) E_cscache.t = E_cscache.create ()

let display_message id =
  lwt msg = E_cscache.find %cache get_data id in
  Lwt.return (li [pcdata msg])

let display_messages () =
  Msg_notif.listen ();
  lwt messages = Db.get_messages () in
  let rmessages = SharedReactiveData.RList.make messages in
  ignore {unit{
    ignore (React.E.map (handle_notif_message_list %rmessages)
      %Msg_notif.client_ev ());
    }};
  lwt content = SharedReactiveData.RList.Lwt.map_p
    {shared{display_message}}
    (fst rmessages)
  in
  Lwt.return (R.ul content)

let display_userid_o =
  lwt messages = display_messages () in
  let l = match userid_o with
    | None -> []
    | _ ->
      let inp = Raw.input ~a:[a_input_type `Text] () in
      let = (unit)
      lwt events in
      let inp = To_dom.of_input %inp in
      async (fun {} -> changes inp (fun {} ->
        let value = Js.to_string (inp##value) in
        inp##value <- Js.string "";
        add_message_rpc value
      ))
    in
    [inp]
  in
  Lwt.return (messages::l)
```

→ 75 lines of code
Conclusion

Client-server reactive programming

→ Pages generated either from client or server side (same code)
  
  *(good for indexing by search engines, etc.)*

→ Very fast page changes (using client-server cache)

→ No need to program the updates of the page at all!

→ Very concise

Eliom base app

→ Send customized notifications to client easily

→ Template for applications with users (quick MVP)

→ ...

ocsigen
Experience:
Rewriting a Web app. from “traditional” Eliom to Client-server reactive Eliom.
Lines of code: 10053 → 4521
Experience:
Rewriting a Web app. from “traditional” Eliom to Client-server reactive Eliom.
Lines of code: 10053 → 4521

Status:
Modules `Eliom_csreact` and `Eliom_cscache` are still experimental (but usable).
If you want to try: Eliom’s Github repository, sharedreact branch.
Ocsigen is dedicated not only to this kind of programs:

→ Client only (using Js_of_ocaml and Eliom)
→ Server only (Web site)
→ HTML5 mobile apps with Phonegap
→ Static Web site (using Server)
→ Embedded Web server (using Server, Eliom, etc) (soon)
→ ...

Conclusion