Eliom
Programmer's guide

Vincent Balat
Laboratoire *Preuves, Programmes et Systèmes*
CNRS - Université Paris Diderot
Case 7014, 75205 Paris Cedex 13, France
`Vincent.Balat @ pps.jussieu.fr`
`http://ocsigen.org`

Eliom is an extension for the *Ocsigen* Web server that allows dynamic webpages generation. It uses very new concepts making programming very different from all other Web programming tools. It allows to write a complex Web site in very few lines of code.

Ocsigen is a research project aimed at developing new programming techniques for the Web. Its goal is to offer an alternative to Apache/PHP, based on cutting-edge technologies from research in programming languages. It contains a full featured Web server and a programming framework, called Eliom, that provides a new way to create dynamic Web sites. With Eliom, you program in a concise and modular way, with a strong type system which helps you to produce valid XHTML. Eliom handles sessions, URLs, and page parameters automatically.

Today, most web sites are written using old technologies, such as CGI scripts or (untyped) scripting languages embedded in html. This situation, was acceptable at the time when the web was mainly static. But today, more and more sites are highly dynamic, and web developers experience difficulties every day for creating and maintaining large pieces of code written in such languages. Inspired by theoretical research in computer science and logic, there are now new languages allowing to program in a cleaner way, drastically reducing the time of the debugging process. But up to now these technologies have not been fully exploited in the domain of web programming.

In particular, some recent research papers have shown that functional programming works very well for this domain (use of continuations or closures, continuation passing style ...). Accordingly, Eliom considers each URL as a function taking arguments and producing a web page. Clicking on a link or a form triggers execution of this function.

Another significant issue addressed by Eliom is the generation of valid xhtml pages. The web relies on recommendations defined by the W3 consortium, and the only means to ensure universal availability of information is to respect these standards. Today, very few web sites are fully standard compliant. Eliom uses an advanced type system that guarantees that your web site will be valid (or very close).

Warning: This manual assumes you know the Objective Caml language.

Table of contents

1. The basics: main services, parameters, forms, cooperative programming

Base principles

Unlike many other Web programming techniques (CGI, PHP, ...), with Eliom, you don't write one file for each URL, but a caml module (cmo or cma) for the whole Web site.

The **Eliom_services** module allows to create new entry points to your Web site, called *services*. In general, services are attached to an URL and generate a Web page. They are represented by OCaml values, on which you must register a function that will generate a page. There are several ways to create pages for Eliom. This tutorial is mainly using **Eliom_predefmod.Xhtml**, a module allowing to register xhtml pages statically typed using OCaml's polymorphic variants. The **XHTML.M** module defines functions to construct xhtml pages using that type system. As the **Eliom_predefmod.Xhtml** redefines some functions of **XHTML.M**, open the modules in this order:

```
open Lwt
open XHTML.M
open Eliom_services
open Eliom_parameters
open Eliom_sessions
open Eliom_predefmod.Xhtml
```

Lwt (lightweight threads) is the cooperative thread library used by Ocsigen (**see later**).

Here is an example showing how to create a new service and register a page created with XHTML.M. Use the function **Eliom_predefmod.Xhtml.register_new_service**:

```
let coucou =
  register_new_service
    ~path:["coucou"]
    ~get_params:unit
    (fun _ () () ->
      return
        (html
           (head (title (pcdata "")) [])
           (body [h1 [pcdata "Hallo!"]])))
```

The same, written with fully qualified values:

```
let coucou =
  Eliom_predefmod.Xhtml.register_new_service
    ~path:["coucou"]
    ~get_params:Eliom_parameters.unit
    (fun _ () () ->
      Lwt.return
        (XHTML.M.html
          (XHTML.M.head (XHTML.M.title (XHTML.M.pcdata "")) [])
          (XHTML.M.body [XHTML.M.h1 [XHTML.M.pcdata "Hallo!"]])))
```

As you can see, **return** is a function from **Lwt**. Use it like this for instants, and **see later** for more advanced use.

Now you can compile your file (here `tutorial.ml`) by typing :

```
ocamlc -I /path_to/ocsigen/ -I /path_to/lwt/ -c tutorial.ml
```

If you use findlib, you can also use the following command line:

```
ocamlfind ocamlc -thread -package lwt,ocsigen -c tutorial.ml
```

Replace */path_to*`/ocsigen/` by the directory where Ocsigen libraries are installed (that contains `eliom.cma`, `staticmod.cmo`, etc.), usually something like `/usr/lib/ocaml/3.09.3/ocsigen` or `/usr/local/lib/ocaml/3.09.3/ocsigen` or `/opt/godi/lib/ocaml/site-lib/ocsigen`.

Add the following lines to Ocsigen's config file (`/etc/ocsigen/ocsigen.conf` most of the time):

```
<host>
 <site path="examples">
  <eliom module="/path_to/tutoeliom.cmo" />
 </site>
</host>
```

Note that if your module has a findlib `META` file, it is also possible to do:

```
<host>
 <site path="examples">
  <eliom findlib-package="package-name" />
 </site>
</host>
```

Then run ocsigen. You should see your page at url `http://`*your_server*`/examples/coucou`.

NB: See the default config file to see how to set the port on which your server is running, the user who runs it, the path of the log files, etc.

Here is a sample **Makefile** for your modules:

```
# Write here all the findlib packages you need, for example:
# PACKAGES= ,extlib,netstring

# Write here all your .ml files, in dependency order (default: all)
FILES=$(wildcard *.ml)

CAMLC = ocamlfind ocamlc -g $(LIB)
CAMLOPT = ocamlfind ocamlopt $(LIB)
CAMLDOC = ocamlfind ocamldoc $(LIB)
CAMLDEP = ocamlfind ocamldep
OCSIGENREP = `ocamlfind query ocsigen`
#OCSIGENREP = ../ocsigen/lib
LIB = -package lwt,ocsigen$(PACKAGES) -I $(OCSIGENREP)
# If you use the syntax extension:
# PP = -pp "camlp4o $(OCSIGENREP)/xhtmlsyntax.cma"
# otherwise
PP =

OBJS = $(FILES:.ml=.cmo)
```

```
CMA = site.cma

all: depend $(CMA) install

$(CMA): $(OBJS)
        $(CAMLC) -a -o $(CMA) $(OBJS)

install:
        chmod a+r $(CMA)

.SUFFIXES:
.SUFFIXES: .ml .mli .cmo .cmi .cmx

.PHONY: doc

.ml.cmo:
        $(CAMLC) $(PP) -c $<

.mli.cmi:
        $(CAMLC) -c $<
.ml.cmx:
        $(CAMLOPT) $(PP) -c $<

doc:
#       $(CAMLDOC) -d doc -html db.mli

clean:
        -rm -f *.cm[ioxa] *~ $(NAME)

depend:
        $(CAMLDEP) $(PP) $(LIB) $(FILES:.ml=.mli) $(FILES) > .depend

FORCE:

-include .depend
```

Static typing of XHTML with XHTML.M

Typing of xhtml with **`XHTML.M`** and **`Eliom_predefmod.Xhtml`** is very strict and compels you to respect xhtml 1.1 standard (with some limitations). For example if you write:

```
(html
   (head (title (pcdata "")) [])
   (body [pcdata "Hallo"]))
```

You will get the following error message:

```
This expression has type ([> `PCDATA ] as 'a) XHTML.M.elt
but is here used with type
([< XHTML.M.block ] as 'b) XHTML.M.elt
Type 'a is not compatible with type
'b =
  [< `Address | `Blockquote | `Del | `Div | `Dl | `Fieldset
   | `Form | `H1 | `H2 | `H3 | `H4 | `H5 | `H6 | `Hr | `Ins
   | `Noscript | `Ol | `P | `Pre | `Script | `Table | `Ul ]
```

`'b` is the type of block tags (only tags allowed inside `<body>`), but PCDATA (i.e. raw text) is not a block tag.

In XHTML, some tags cannot be empty. For example `<table>` must contain at least one row. To enforce this, the **`XHTML.M.table`** function takes two parameters: the first one is the first row, the second one is a list containing all the other rows. (same thing for `<tr>` `<form>` `<dl>` `<ol>` `<ul>` `<dd>` `<select>` ...) This forces the user to handle the empty list case specially and thus make the output conform to the DTD.

A more detailed introduction to `XHTML.M` is available **`on the Web site`**. Take a quick look at it before continuing this tutorial.

Alternate syntax

If you prefer using a syntax closer to html, you can write:

```
let coucou1 =
  Eliom_predefmod.Xhtml.register_new_service
    ~path:["coucou1"]
    ~get_params:Eliom_parameters.unit
    (fun _ () () ->
      return
        << <html>
             <head><title></title></head>
             <body><h1>Coucou</h1></body>
           </html> >>)
```

To compile this syntax, you need a camlp4 syntax extension:

```
ocamlc -I /path_to/ocsigen/
 -pp "camlp4o /path_to/ocsigen/xhtmlsyntax.cma -loc loc"
 -c tutorial.ml
```

(Replace */path_to*`/ocsigen/` by the directory where ocsigen is installed).

As the syntax extension is using the same typing system as XHTML.M, You can mix the two syntaxes (**see later**).

Warning: The two syntaxes are not equivalent for typing. Using the syntax extension will do less checking. For example the following code is accepted but not valid regarding xhtml's dtd (because `<head>` must contain a title):

```
<< <html>
     <head></head>
     <body><h1>plop</h1></body>
   </html> >>
```

We recommend you to use the functions from **`XHTML.M`**, as you will (almost) always get valid xhtml. Use the syntax extension for example to enclose already created pieces of html, and check your pages validity with the **W3C validator**.

Eliom and OcamlDuce

If OCamlDuce is installed on your system, it is now possible to use it instead of XHTML.M and Eliom_parameters.Xhtml to typecheck your pages. You will get a stronger type checking and more flexibility (easier to use other XML types, to parse incoming XML data, etc.).

To use it, make sure that you have Eliom compiled with OCamlDuce support. Then dynlink `ocamlduce.cma` and `eliomduce.cma` from the configuration file (after `eliom.cma`). Then use **`Eliom_duce.Xhtml`** instead of **`Eliom_predefmod.Xhtml`** to register your pages.

Here is an example:

```
open Lwt

let s =
  Eliom_duce.Xhtml.register_new_service
    ~path:[""]
    ~get_params:unit
    (fun sp () () ->
      return
        {{ <html>
             [<head> [<title> ""]
              <body> [<h1> "This page has been \
type checked by OCamlDuce"]] }})
```

Eliom_predefmod.HtmlText

If you want to register untyped (text) pages, use the functions from **`Eliom_predefmod.HtmlText`**, for example **`Eliom_predefmod.Text.register_new_service`**:

```
let coucoutext =
  Eliom_predefmod.HtmlText.register_new_service
    ~path:["coucoutext"]
    ~get_params:Eliom_parameters.unit
    (fun sp () () ->
      return
        ("<html>n'importe quoi "^
         (Eliom_predefmod.HtmlText.a coucou sp "clic" ())^
         "</html>"))
```

More examples

Services registered with `register_new_service` are available for all users. We call them *public services*.

Page generation may have side-effects:

```
let count =
  let next =
    let c = ref 0 in
      (fun () -> c := !c + 1; !c)
  in
  register_new_service
    ~path:["count"]
    ~get_params:unit
    (fun _ () () ->
```

```
      return
        (html
         (head (title (pcdata "counter")) [])
         (body [p [pcdata (string_of_int (next ()))]])))
```

As usual in OCaml, you can forget labels when the application is total:

```
let hello =
  register_new_service
    ["dir";"hello"]  (* the url dir/hello *)
    unit
    (fun _ () () ->
      return
        (html
         (head (title (pcdata "Hello")) [])
         (body [h1 [pcdata "Hello"]])))
```

The last example shows how to define the default page for a directory. (Note that `["rep";""]` means the default page of the directory `rep/`)

```
let default = register_new_service ["rep";""] unit
  (fun _ () () ->
    return
     (html
      (head (title (pcdata "")) [])
      (body [p [pcdata "default page. rep is redirected to \
rep/"]])))
```

Remarks on paths

`["foo";"bar"]` corresponds to the URL `foo/bar`.
`["dir";""]` corresponds to the URL `dir/` (that is: the default page of the directory `dir`).
The empty list `[]` is equivalent to `[""]`.

Warning: You cannot create a service on path `["foo"]` (URL `foo`, without slash at the end) and another on path `["foo";"bar"]` (URL `foo/bar`) because `foo` can not be both a directory and a file. Be also careful not to use a string as a directory with Eliom, if it is a file for Staticmod (and vice versa).

Warning: `["foo";"bar"]` is not equivalent to `["foo/bar"]`. In the latter, the "/" will be encoded in the URL.

Parameters

Typed parameters

The parameter labeled `~get_params` indicates the type of GET parameters for the page (that is, parameters present in the URL). **`unit`** means that the page does not take any GET parameter.

Functions implementing services are called *service handlers*. They take three parameters. The first one has type **`Eliom_sessions.server_params`** and corresponds to server parameters (user-agent, ip, current-url, etc. - see later in that section for examples of use),

the second one is for GET parameters (that is, parameters in the URL) and the third one for POST parameters (parameters in the body of the HTTP request).

Here is an example of a service with GET parameters:

```
let writeparams _ (i1, (i2, s1)) () =
  return
   (html
    (head (title (pcdata "")) [])
    (body [p [pcdata "You sent: ";
              strong [pcdata (string_of_int i1)];
              pcdata ", ";
              strong [pcdata (string_of_int i2)];
              pcdata " and ";
              strong [pcdata s1]]]))

let coucou_params = register_new_service
    ~path:["coucou"]
    ~get_params:(int "i" ** (int "ii" ** string "s"))
    writeparams
```

Note that the URLs of `coucou` and `coucou_params` differ only by parameters. Url **`http://your_server/examples/coucou`** will run the first one, **`http://your_server/examples/coucou?i=42&ii=17&s=krokodile`** will run the second one.
If `i` is not an integer, the server will display an error-message (try to change the value in the URL).
Here, `int`, `string` and `**` are functions defined in the **`Eliom_parameters`** module.
Warning: The infix function `( ** )` is to be used to construct *pairs* (not tuples).

The following examples shows how to create a service with "suffix" service (taking the end of the URL as a parameter, as wikis do very often) and how to get server information:

```
let uasuffix =
  register_new_service
    ~path:["uasuffix"]
    ~get_params:(suffix (int "year" ** int "month"))
    (fun sp (year, month) () ->
      return
       (html
        (head (title (pcdata "")) [])
        (body
           [p [pcdata "The suffix of the url is ";
               strong [pcdata ((string_of_int year)^"/"
                               ^(string_of_int month))];
               pcdata ", your user-agent is ";
               strong [pcdata (Eliom_sessions.get_user_agent sp)];
               pcdata ", your IP is ";
               strong [pcdata (Eliom_sessions.get_remote_ip sp)]]])))
```

This service will answer to URLs like `http://.../uasuffix/2000/11`.

Suffix parameters have names, because we can create forms towards these services. `uasuffix/2000/11` is equivalent to `uasuffix/?year=2000&month=11`.

`suffix_prod` allows to take both a suffix and other parameters.
`all_suffix` allows to take the end of the suffix as a `string list`.

```
let isuffix =
  register_new_service
    ~path:["isuffix"]
    ~get_params:(suffix_prod
      (int "suff" ** all_suffix "endsuff")
      (int "i"))
    (fun sp ((suff, endsuff), i) () ->
      return
       (html
        (head (title (pcdata "")) [])
        (body
           [p [pcdata "The suffix of the url is ";
               strong [pcdata (string_of_int suff)];
               pcdata " followed by ";
               strong [pcdata
                 (Ocsigen_extensions.string_of_url_path endsuff)];
               pcdata " and i is equal to ";
               strong [pcdata (string_of_int i)]]])))
```

The following example shows how to use your own types :

```
type mysum = A | B
let mysum_of_string = function
  | "A" -> A
  | "B" -> B
  | _ -> raise (Failure "mysum_of_string")
let string_of_mysum = function
  | A -> "A"
  | B -> "B"

let mytype =
  Eliom_predefmod.Xhtml.register_new_service
    ~path:["mytype"]
    ~get_params:
      (Eliom_parameters.user_type
        mysum_of_string string_of_mysum "valeur")
    (fun _ x () ->
      let v = string_of_mysum x in
      return
        (html
         (head (title (pcdata "")) [])
         (body [p [pcdata
           (v^" is valid. Now try with another value.")]])))
```

Untyped parameters

If you want a service that answers to requests with any parameters, use the **`Eliom_parameters.any`** value. The service will get an association list of strings. Example:

```
let raw_serv = register_new_service
    ~path:["any"]
    ~get_params:Eliom_parameters.any
  (fun _ l () ->
    let ll =
      List.map
        (fun (a,s) -> << <strong>($str:a$, $str:s$)</strong> >>) l
```

```
    in
    return
     << <html>
          <head><title></title></head>
          <body>
          <p>
            You sent:
            $list:ll$
          </p>
          </body>
        </html> >>)
```

Catching errors

You can catch parameters typing errors and write your own error messages using the optional parameter `error_handler`. Example:

```
let catch = register_new_service
    ~path:["catch"]
    ~get_params:(int "i")
    ~error_handler:(fun sp l ->
      return
        (html
         (head (title (pcdata "")) [])
         (body [p [pcdata ("i is not an integer.")]])))
    (fun _ i () ->
      let v = string_of_int i in
      return
        (html
           (head (title (pcdata "")) [])
           (body [p [pcdata ("i is an integer: "^v)]])))
```

`error_handler` takes as parameters the usual `sp`, and a list of pairs `(n,ex)`, where `n` is the name of the wrong parameter, and `ex` is the exception that has been raised while parsing its value.

Links

To create a link (<a>), use the **`Eliom_predefmod.Xhtml.a`** function (or `Eliom_duce.Xhtml.a`, etc), as in these examples:

```
let links = register_new_service ["rep";"links"] unit
 (fun sp () () ->
   return
    (html
     (head (title (pcdata "Links")) [])
     (body
       [p
        [Eliom_predefmod.Xhtml.a coucou sp
          [pcdata "coucou"] (); br ();
         Eliom_predefmod.Xhtml.a hello sp
           [pcdata "hello"] (); br ();
```

```
          Eliom_predefmod.Xhtml.a default sp
            [pcdata "default page of the dir"] (); br ();
          Eliom_predefmod.Xhtml.a uasuffix sp
            [pcdata "uasuffix"] (2007,06); br ();
          Eliom_predefmod.Xhtml.a coucou_params sp
            [pcdata "coucou_params"] (42,(22,"ciao")); br ();
          Eliom_predefmod.Xhtml.a raw_serv sp
            [pcdata "raw_serv"]
            [("sun","yellow");("sea","blue and pink")]; br ();
          Eliom_predefmod.Xhtml.a
            (new_external_service
               ~prefix:"http://fr.wikipedia.org"
               ~path:["wiki";""]
               ~get_params:(suffix (all_suffix "suff"))
               ~post_params:unit ())
            sp
            [pcdata "OCaml on wikipedia"]
            ["OCaml"]; br ();
          XHTML.M.a
            ~a:[a_href (uri_of_string
              "http://en.wikipedia.org/wiki/OCaml")]
            [pcdata "OCaml on wikipedia"]
        ]])))
```

If you open **`Eliom_predefmod.Xhtml`** after **`XHTML.M`**, **`Eliom_predefmod.Xhtml.a`** will mask **`XHTML.M.a`**. Thus you can avoid to write fully qualified values most of the time.

`Eliom_predefmod.Xhtml.a` takes as first parameter the service you want to link to. Note that to create a (relative) link we need to know the current URL. That's why the function `a` takes `sp` as second parameter.

The third parameter is the text of the link. The last parameter is for GET parameters you want to put in the link. The type of this parameter and the name of GET parameters depend on the service you link to.

The links to Wikipedia shows how to define an external service (here it uses a suffix URL). For an external service without parameters, you can use the low level function **`XHTML.M.a`**, if you don't want to create an external service explicitely. Note that the path must be a list of strings. Do not write `["foo/bar"]`, but `["foo";"bar"]`, otherwise, the "/" will be encoded in the URL.

If you want to create (mutually or not) recursive pages, create the service using **`Eliom_services.new_service`** first, then register it in the table using (for example) **`Eliom_predefmod.Xhtml.register`**:

```
let linkrec = Eliom_services.new_service ["linkrec"] unit ()

let _ = Eliom_predefmod.Xhtml.register linkrec
    (fun sp () () ->
      return
       (html
        (head (title (pcdata "")) [])
        (body [p [a linkrec sp [pcdata "click"] ()]])))
```

The server will fail on startup if there are any unregistered services.

Forms

Forms towards services

The function **`Eliom_predefmod.Xhtml.get_form`** allows to create a form that uses the GET method (parameters in the URL). It works like **`Eliom_predefmod.Xhtml.a`** but takes a *function* that creates the form from the parameters names as parameter.

```
let create_form =
  (fun (number_name, (number2_name, string_name)) ->
    [p [pcdata "Write an int: ";
        Eliom_predefmod.Xhtml.int_input ~input_type:`Text
          ~name:number_name ();
        pcdata "Write another int: ";
        Eliom_predefmod.Xhtml.int_input ~input_type:`Text
          ~name:number2_name ();
        pcdata "Write a string: ";
        Eliom_predefmod.Xhtml.string_input ~input_type:
          `Text ~name:string_name ();
        Eliom_predefmod.Xhtml.string_input
          ~input_type:`Submit ~value:"Click" ()]])

let form = register_new_service ["form"] unit
  (fun sp () () ->
     let f = Eliom_predefmod.Xhtml.get_form
       coucou_params sp create_form in
     return
       (html
         (head (title (pcdata "")) [])
         (body [f])))
```

Note that if you want to use typed parameters, you cannot use functions like **`XHTML.M.input`** to create your forms (if you want to use parameters defined with **`Eliom_parameters.any`**, **see later**). Indeed, parameter names are typed to force them be used properly. In our example, `number_name` has type `int param_name` and must be used with `int_input` (or other widgets), whereas `string_name` has type `string param_name` and must be used with `string_input` (or other widgets). All functions for creating form widgets are detailed **here**.

For untyped forms, you may use functions from XHTML.M (or OCamlDuce's syntax, or whatever syntax you are using) or functions which name is prefixed by "`raw_`". Here is a form linking to our (untyped) service `raw_serv`.

```
let raw_form = register_new_service
    ~path:["anyform"]
    ~get_params:unit
    (fun sp () () ->
      return
        (html
           (head (title (pcdata "")) [])
           (body
              [h1 [pcdata "Any Form"];
               Eliom_predefmod.Xhtml.get_form raw_serv sp
                 (fun () ->
                   [p [pcdata "Form to raw_serv: ";
```

```
                      Eliom_predefmod.Xhtml.raw_input
                        ~input_type:`Text ~name:"plop" ();
                      Eliom_predefmod.Xhtml.raw_input
                        ~input_type:`Text ~name:"plip" ();
                      Eliom_predefmod.Xhtml.raw_input
                        ~input_type:`Text ~name:"plap" ();
                      Eliom_predefmod.Xhtml.string_input
                        ~input_type:`Submit ~value:"Click" ()]])
                ]]))
```

POST parameters

By default Web page parameters are transferred in the URL (GET parameters). A web page may also expect POST parameters (that is, parameters that are not in the URL but in the body of the HTTP request). Use this if you don't want the user to be able to bookmark the URL with parameters, for example if you want to post some data that will change the state of the server (payment, database changes, etc). When designing a Web site, think carefully about the choice between GET or POST method for each service!

When you register a service with POST parameters, you must first register a service (fallback) without these parameters (for example that will answer if the page is reloaded without the hidden parameters, or if it is bookmarked).

```
let no_post_param_service =
  register_new_service
    ~path:["post"]
    ~get_params:unit
    (fun _ () () ->
      return
        (html
         (head (title (pcdata "")) [])
         (body [h1 [pcdata
           "Version of the page without POST parameters"]])))

let my_service_with_post_params =
  register_new_post_service
    ~fallback:no_post_param_service
    ~post_params:(string "value")
    (fun _ () value ->
      return
        (html
         (head (title (pcdata "")) [])
         (body [h1 [pcdata value]])))
```

Services may take both GET and POST parameters:

```
let get_no_post_param_service =
  register_new_service
    ~path:["post2"]
    ~get_params:(int "i")
    (fun _ i () ->
      return
        (html
         (head (title (pcdata "")) [])
         (body [p [pcdata "No POST parameter, i:";
                   em [pcdata (string_of_int i)]]])))

let my_service_with_get_and_post = register_new_post_service
```

```
  ~fallback:get_no_post_param_service
  ~post_params:(string "value")
  (fun _ i value ->
    return
      (html
         (head (title (pcdata "")) [])
         (body [p [pcdata "Value: ";
                   em [pcdata value];
                   pcdata ", i: ";
                   em [pcdata (string_of_int i)]]])))
```

POST forms

To create a POST form, use the **`Eliom_predefmod.Xhtml.post_form`** function. It is similar to **`Eliom_predefmod.Xhtml.get_form`** with an additional parameter for the GET parameters you want to put in the URL (if any). Here, `form2` is a page containing a form to the service `post` (using XHTML.M's functions) and `form3` (defined using the syntax extension) contains a form to `post2`, with a GET parameter. `form4` is a form to an external page.

```
let form2 = register_new_service ["form2"] unit
  (fun sp () () ->
     let f =
       (Eliom_predefmod.Xhtml.post_form
          my_service_with_post_params sp
          (fun chaine ->
            [p [pcdata "Write a string: ";
                string_input ~input_type:`Text
                  ~name:chaine ()]]) ()) in
     return
       (html
         (head (title (pcdata "form")) [])
         (body [f])))

let form3 = register_new_service ["form3"] unit
  (fun sp () () ->
     let f  =
       (Eliom_predefmod.Xhtml.post_form
          my_service_with_get_and_post sp
          (fun chaine ->
            <:xmllist< <p> Write a string:
                    $string_input ~input_type:`Text
                      ~name:chaine ()$ </p> >>)
          222) in
     return
       << <html>
            <head><title></title></head>
            <body>$f$</body></html> >>)

let form4 = register_new_service ["form4"] unit
  (fun sp () () ->
     let f  =
       (Eliom_predefmod.Xhtml.post_form
          (new_external_service
             ~prefix:"http://www.petizomverts.com"
             ~path:["zebulon"]
             ~get_params:(int "i")
```

```
            ~post_params:(string "chaine") ()) sp
          (fun chaine ->
            <:xmllist< <p> Write a string:
                       $string_input ~input_type:`Text
                         ~name:chaine ()$ </p> >>)
          222) in
    return
      (html
       (head (title (pcdata "form")) [])
       (body [f])))
```

Threads

Remember that a Web site written with Eliom is an OCaml application. This application must be able to handle several requests at the same time, in order to prevent a single request from slowing down the whole server. To make this possible, Ocsigen is using *cooperative threads* (implemented in monadic style by Jérôme Vouillon) which make them really easy to use (see **`Lwt`** module).

Take time to read the **documentation about `Lwt`** right now if you want to understand the following of this tutorial.

As it doesn't cooperate, the following page will stop the server for 5 seconds. No one will be able to query the server during this period:

```
let looong =
  register_new_service
    ~path:["looong"]
    ~get_params:unit
    (fun sp () () ->
      Unix.sleep 5;
      return
        (html
          (head (title (pcdata "")) [])
          (body [h1 [pcdata
            "Ok now, you can read the page."]])))
```

To solve this problem, use a cooperative version of **`sleep`**:

```
let looong =
  register_new_service
    ~path:["looong"]
    ~get_params:unit
    (fun sp () () ->
      Lwt_unix.sleep 5.0 >>= fun () ->
      return
        (html
          (head (title (pcdata "")) [])
          (body [h1 [pcdata
                    "Ok now, you can read the page."]])))
```

If you want to use, say, a database library that is not written in cooperative way, but is thread safe for preemptive threads, use the `Lwt_preemptive` module to detach the computation. In the following example, we simulate the request by a call to `Unix.sleep`:

```
let looong2 =
  register_new_service
    ~path:["looong2"]
    ~get_params:unit
    (fun sp () () ->
      Lwt_preemptive.detach Unix.sleep 5 >>= fun () ->
      return
        (html
          (head (title (pcdata "")) [])
          (body [h1 [pcdata
                   "Ok now, you can read the page."]])))
```

The big picture

You now have the minimum knowledge to write basic Web sites with Eliom: page typing, service creation, parameters, forms and database acces using **`Lwt`** (and possibly **`Lwt_preemptive.detach`**). Here is a summary of all other concepts introduced by Eliom. They will allow you to easily program more complex behaviours and will be developped in the following sections of this tutorial.

Different kinds of services

Before beginning the implementation, think about the URLs you want to create as entry points to your Web site, and the services you want to provide.

Services we used, so far, are called *main services*. For instants, Eliom uses four kinds of services:

Main services
: are the main entry points of your sites. Created by `new_service` or `new_post_service`. They correspond to the public URLs of your Web site, and will last forever.

(Attached) coservices
: are services that share their location (URL) with a main service (fallback). They are distinguished from that main service using a special parameter (added automatically by Eliom). They are often created dynamically for one user (usually in the session table), depending on previous interaction during the session. In general, they disappear after a timeout letting the fallback answer afterwards. Another use of (POST) coservices is to customize one button but not the page it leads to (like the disconnect button in the example of sessions with *actions* as below).

Non-attached services
: are services that are not attached to a particular URL. A link towards a non-attached service will go to the current URL with a special parameter containing the name of the service. It is useful when you want the same link or form on several pages (for example a connection box) but you don't want to go to another URL. Non-attached (co)services are often used with *actions* (see below).

Non-attached coservices
: are coservices that are not attached to a particular URL. The difference with non-attached services is that they have no name, but a number generated automatically (different at each time).

GET or POST?

Each of these services both have a version with GET parameters and another with POST parameters.

POST and GET parameters are not equivalent, and you must be very careful if you want to use one or the other.
GET parameters are the parameters you see in the URL (for example `http://your_server/examples/coucou?i=42&ii=17&s=krokodile`).
They are created by browsers if you use forms with the GET method, or written directly in the URL.
POST parameters are sent by browsers in the body of the HTTP request. That's the only solution if you want to send files with your request.

Remember that only pages without POST parameters are bookmarkable. Use GET parameters if you want the user to be able to come back to the URL later or to write the URL manually.
Use POST parameters when you want a different behaviour between the first click and a reload of the page. Usually sending POST parameters triggers an action on server side (like a payment, or adding something in a database), and you don't want it to succeed several times if the page is reloaded or bookmarked.

Data returned by services

Services can send several types of data, using these different modules:

	Services		Coservices	
	attached	**non-attached**	**attached**	**non-attached**
Eliom_predefmod.Xhtml	Allows to register functions that generate xhtml pages statically checked using polymorphic variant types. You may use constructor functions from **XHTML.M** or a syntax extension close to the standard xhtml syntax.			
Eliom_predefmod.Xhtmlcompact	Same, but without pretty printing (does not add spaces or line breaks).			
Eliom_predefmod.Blocks	Allows to register functions that generate a portion of page (content of the body tag) using **XHTML.M** or the syntax extension. (useful for `XMLHttpRequest` requests for example).			
Eliom_duce.Xhtml	Allows to register functions that generate xhtml pages statically checked using `OCamlduce`. Typing is stricter, and you need a modified version of the OCaml compiler (OCamlduce).			
Eliom_predefmod.HtmlText	Allows to register functions that generate text html pages, without any typechecking of the content. The content type sent by the server is "text/html".			
Eliom_predefmod.CssText	Allows to register functions that generate CSS pages, without any typechecking of the content. The content type sent by the server is "text/css".			
Eliom_predefmod.Text	Allows to register functions that generate text pages, without any typechecking of the content. The services return a pair of strings. The first one is the content of the page, the second one is the content type.			
Eliom_predefmod.Actions	Allows to register actions (functions that do not generate any page). The URL is reloaded after the action.			
Eliom_predefmod.Unit	is like **Eliom_predefmod.Actions** but the URL is not reloaded after the action.			
Eliom_predefmod.Redirection	**[New in 1.1.0]** Allows to register HTTP permanent redirections. You register the URL of the page you want to redirect to. Warning: According to the RFC of the HTTP protocol, the URL must be absolute! The browser will get a 301 or 307 code in answer and redo the request to the new URL. To specify whether you want temporary (307) or permanent (301) redirections, use the `?options` parameter of registration functions. For example: `register ~options:`Permanent ...` or `register ~options:`Temporary ...`.			
Eliom_predefmod.Files	Allows to register services that send files			
Eliom_predefmod.Any	Allows to register services that can choose what they send, for example an xhtml page or a file, depending on some situation (parameter, user logged or not, page present in a cache ...).			

It is also possible to create your own modules for other types of pages.

Public and session service tables

Each of these registrations may be done in the *public* service table, or in a *session* service table, accessible only to a single user of the Web site. This allows to generate custom services for a specific user.

Eliom will try to find the page, in that order:

in the session service table,
in the public service table,
the fallback in the session table, if the coservice has expired,
the fallback in the public table, if the session has expired.

Session data tables

It is also possible to create a session data table, where you can save information about the session. Each service can look in that table to see if a session is opened or not and get the data.

Examples

The most commonly used services are:

Main services (GET or POST) in public service table for public pages,
GET attached coservices in session service table to make the browser's "back" button turn back in the past, and to allow several tabs on different versions of the same page,
Actions registered on POST non-attached services to make an effect on the server, from any page, and without changing the URL (connection/disconnection for example).

Here is a list of frequent issues and the solution Eliom provides to to solve them. Most of them will be developped in the following parts of the tutorial.

Display the result of a search (plane ticket, search engines ...)
Use a coservice (with timeout) in the session service table.

Keep information about the session (name of the user ...)
Use a session data table.

A connection or disconnection box on each page of your site
Use actions registered on non-attached services to set or remove data from a session data table.

Add something in a shopping basket
Use an action registered on a non-attached coservice, with the names of the items as parameters. The action saves the shopping basket in a session data table. Thus, the shopping basket will remain even if the user pushes the back button of his browser.

Book a ticket (in several steps)
Each step creates new (GET) coservices (with or without parameters, all attached to the service displaying the main booking page) according to the data entered by the user. These coservices are registered in the session table (with a timeout for the whole session or for each of them). Thus the user can go back to a previous state, or keep several proposals on differents tabs before choosing one.

...

Help us to complete this list by giving your examples or asking questions about other cases! Thank you!

2. Sessions, users and other common situations in Web sites

When programming dynamic Web sites, you often want to personalise the behaviour and content for one user. To do this, you need to recognise the user and save and restore its data. Eliom implements several high level features to do that:

- Session data tables,
- Session service tables, where you can save private versions of main services or new coservices,
- Coservices, that may be created dynamically with respect to previous interaction with the user.

Eliom is using cookies to recognize users. One cookie is automatically set for each user when needed and destroyed when the session is closed.

Coservices, but also *actions*, are also means to control precisely the behaviour of the site and to implement easily very common situations that require a lot of programming work with other Web programming tools. We'll have a lot at some examples in that section.

Session data

Eliom provides a way to save session data on server side and restore it at each request. This data is available during the whole duration of the session. To save session data, create a table using **`Eliom_sessions.create_volatile_table`** and save and get data from this table using **`Eliom_sessions.set_volatile_session_data`** and **`Eliom_sessions.get_volatile_session_data`**. The following example shows a site with authentification. The name of the user is asked in the login form and saved in a table to be displayed on the page instead of the login form while the user is connected. Note that the session is opened automatically when needed.

```
(*****************************************************************)
(************ Connection of users, version 1 *********************)
(*****************************************************************)



(* "my_table" will be the structure used to store
   the session data (namely the login name): *)

let my_table = Eliom_sessions.create_volatile_table ()


(* ---------------------------------------------------------- *)
(* Create services, but do not register them yet:             *)

let session_data_example =
  Eliom_services.new_service
    ~path:["sessdata"]
    ~get_params:Eliom_parameters.unit
    ()

let session_data_example_with_post_params =
  Eliom_services.new_post_service
```

```
    ~fallback:session_data_example
    ~post_params:(Eliom_parameters.string "login")
    ()

let session_data_example_close =
  Eliom_services.new_service
    ~path:["close"]
    ~get_params:Eliom_parameters.unit
    ()



(* ---------------------------------------------------------- *)
(* Handler for the "session_data_example" service:           *)

let session_data_example_handler sp _ _  =
  let sessdat = Eliom_sessions.get_volatile_session_data
    ~table:my_table ~sp () in
  return
    (html
       (head (title (pcdata "")) [])
       (body
          [
           match sessdat with
           | Eliom_sessions.Data name ->
               p [pcdata ("Hello "^name);
                  br ();
                  Eliom_predefmod.Xhtml.a
                    session_data_example_close
                    sp [pcdata "close session"] ()]
           | Eliom_sessions.Data_session_expired
           | Eliom_sessions.No_data ->
               Eliom_predefmod.Xhtml.post_form
                 session_data_example_with_post_params
                 sp
                 (fun login ->
                   [p [pcdata "login: ";
                       Eliom_predefmod.Xhtml.string_input
                         ~input_type:`Text ~name:login ()]]) ()
         ]))

(* ---------------------------------------------------------- *)
(* Handler for the "session_data_example_with_post_params"  *)
(* service with POST params:                                 *)

let session_data_example_with_post_params_handler sp _ login =
  Eliom_sessions.close_session  ~sp () >>= fun () ->
  Eliom_sessions.set_volatile_session_data
    ~table:my_table ~sp login;
  return
    (html
       (head (title (pcdata "")) [])
       (body
          [p [pcdata ("Welcome " ^ login ^
            ". You are now connected.");
              br ();
              Eliom_predefmod.Xhtml.a session_data_example sp
                [pcdata "Try again"] ()
```

```
            ]]))



(* ---------------------------------------------------------- *)
(* Handler for the "session_data_example_close" service:    *)

let session_data_example_close_handler sp () () =
  let sessdat = Eliom_sessions.get_volatile_session_data
    ~table:my_table ~sp () in
  Eliom_sessions.close_session  ~sp () >>= fun () ->
  return
    (html
       (head (title (pcdata "Disconnect")) [])
       (body [
        (match sessdat with
        | Eliom_sessions.Data_session_expired ->
          p [pcdata "Your session has expired."]
        | Eliom_sessions.No_data ->
          p [pcdata "You were not connected."]
        | Eliom_sessions.Data _ ->
          p [pcdata "You have been disconnected."]);
        p [Eliom_predefmod.Xhtml.a session_data_example sp
          [pcdata "Retry"] () ]]))


(* ---------------------------------------------------------- *)
(* Registration of main services:                           *)

let () =
  Eliom_predefmod.Xhtml.register
    session_data_example_close session_data_example_close_handler;
  Eliom_predefmod.Xhtml.register
    session_data_example session_data_example_handler;
  Eliom_predefmod.Xhtml.register
    session_data_example_with_post_params
    session_data_example_with_post_params_handler
```

To close a session, use the function **`Eliom_sessions.close_session`**. Session data will disappear when the session is closed (explicitely or by timeout). Warning: if your session data contains opened file descriptors, they won't be closed by OCaml's garbage collector. Close it yourself! (for example using `Gc.finalise`).

We will see in the following of this tutorial how to improve this example to solve the following problems:

- The use of a main service for disconnection is not a good idea for usability. You probably want to go to the same page with the login form. We will do this with a coservice.
- If you want the same login form on several pages, it is tedious work to create a coservice with POST parameters for each page. We will se how to solve this using actions and non-attached services.
- Session data are kept in memory and will be lost if you switch off the server, which is bad if you want long duration sessions. You can solve this problem by using persistent tables.

Session services

Eliom allows to replace a public service by a service valid only for one user. Use this to personalise main services for one user (or to create new coservices available only to one user, **see later**). To create a "session service", register the service in a "session service table" (valid only for one client) instead of the public table. To do that, use **`register_for_session`** (for example **`Eliom_predefmod.Xhtml.register_for_session`**).

Users are recognized automatically using a cookie. Use this for example if you want two versions of each page, one public, one for connected users.
To close a session, use **`Eliom_sessions.close_session`**. Both the session service table and the session data table for that user will disappear when the session is closed.

Note that `register_for_session` and `close_session` take `sp` as parameter (because sp contains the session table).

The following example shows how to reimplement the previous one (`session_data_example`), without using **`Eliom_sessions.set_volatile_session_data`**. Note that this version is less efficient than the other if your site has lots of pages, because it requires to register all the new services each time a user logs in. But in other cases, that feature is really useful, for example with coservices (see **later**).

We first define the main page, with a login form:

```
(************************************************************)
(************ Connection of users, version 2 ****************)
(************************************************************)


(* ---------------------------------------------------------- *)
(* Create services, but do not register them yet:             *)

let session_services_example =
  Eliom_services.new_service
    ~path:["sessionservices"]
    ~get_params:Eliom_parameters.unit
    ()

let session_services_example_with_post_params =
  Eliom_services.new_post_service
    ~fallback:session_services_example
    ~post_params:(Eliom_parameters.string "login")
    ()

let session_services_example_close =
  Eliom_services.new_service
    ~path:["close2"]
    ~get_params:Eliom_parameters.unit
    ()


(* ---------------------------------------------------------- *)
(* Handler for the "session_services_example" service:        *)
(* It displays the main page of our site, with a login form.  *)

let session_services_example_handler sp () () =
  let f =
```

```
    Eliom_predefmod.Xhtml.post_form
      session_services_example_with_post_params
      sp
      (fun login ->
        [p [pcdata "login: ";
            string_input ~input_type:`Text ~name:login ()]]) ()
  in
  return
    (html
       (head (title (pcdata "")) [])
       (body [f]))


(* ---------------------------------------------------------------- *)
(* Handler for the "session_services_example_close" service:       *)

let session_services_example_close_handler sp () () =
  Eliom_sessions.close_session  ~sp () >>= fun () ->
  return
    (html
       (head (title (pcdata "Disconnect")) [])
       (body [p [pcdata "You have been disconnected. ";
                 a session_services_example
                   sp [pcdata "Retry"] ()
                ]]))
```

When the page is called with login parameters, it runs the function `launch_session` that replaces some services already defined by new ones:

```
(* ---------------------------------------------------------------- *)
(* Handler for the "session_services_example_with_post_params"     *)
(* service:                                                        *)

let launch_session sp () login =

  (* New handler for the main page: *)
  let new_main_page sp () () =
    return
      (html
       (head (title (pcdata "")) [])
       (body [p [pcdata "Welcome ";
                 pcdata login;
                 pcdata "!"; br ();
                 a coucou sp [pcdata "coucou"] (); br ();
                 a hello sp [pcdata "hello"] (); br ();
                 a links sp [pcdata "links"] (); br ();
                 a session_services_example_close
                   sp [pcdata "close session"] ()]]))
  in

  (* If a session was opened, we close it first! *)
  Eliom_sessions.close_session  ~sp () >>= fun () ->

  (* Now we register new versions of main services in the
     session service table: *)
  Eliom_predefmod.Xhtml.register_for_session
    ~sp
    ~service:session_services_example
```

```
    (* service is any public service already registered,
       here the main page of our site *)
    new_main_page;

  Eliom_predefmod.Xhtml.register_for_session
    ~sp
    ~service:coucou
    (fun _ () () ->
      return
        (html
         (head (title (pcdata "")) [])
         (body [p [pcdata "Coucou ";
                   pcdata login;
                   pcdata "!"]])));

  Eliom_predefmod.Xhtml.register_for_session
    ~sp
    ~service:hello
    (fun _ () () ->
      return
        (html
         (head (title (pcdata "")) [])
         (body [p [pcdata "Ciao ";
                   pcdata login;
                   pcdata "!"]])));

  new_main_page sp () ()

(* ---------------------------------------------------------- *)
(* Registration of main services:                             *)

let () =
  Eliom_predefmod.Xhtml.register
    ~service:session_services_example
    session_services_example_handler;
  Eliom_predefmod.Xhtml.register
    ~service:session_services_example_close
    session_services_example_close_handler;
  Eliom_predefmod.Xhtml.register
    ~service:session_services_example_with_post_params
    launch_session
```

Warning: As in the previous example, to implement such connection and disconnection forms, you get more flexibility by using *actions* instead of xhtml services (see below for the same example with actions).

Services registered in session tables are called *session* or *private* services. Services registered in the public table are called *public*.

Coservices

A coservice is a service that uses the same URL as a main service, but generates another page. They are distinguished from main services only by a special parameter, called *state* parameter. Coservices may use GET or POST parameters.

Most of the time, GET coservices are created dynamically with respect to previous interaction with the user and are registered in the session table. They allow to give a precise semantics to the "back" button of the browser (be sure that you will go back in the past) or bookmarks, or duplication of the browser's window. (See the **calc** example below).

Use POST coservices if you want to particularize a link or form, but not the URL it points to. More precisely, POST coservices are mainly used in two situations:

> For the same purpose as GET coservices (new services corresponding to precise points of the interaction with the user) but when you don't want this service to be bookmarkable.
> To create a button that leads to a service after having performed a side-effect. For example a disconnection button that leads to the main page of the side, but with the side effect of disconnecting the user.

To create a coservice, use **Eliom_services.new_coservice** and **Eliom_services.new_post_coservice**. Like **Eliom_services.new_post_service**, they take a public service as parameter (labeled `fallback`) to be used as fallback when the user comes back without the state parameter (for example if it was a POST coservice and/or the coservice has expired).

The following example shows the difference between GET coservices (bookmarkable) and POST coservices:

```
(************************************************************)
(************** Coservices. Basic examples ******************)
(************************************************************)

(* ---------------------------------------------------------- *)
(* We create one main service and two coservices:             *)

let coservices_example =
  Eliom_services.new_service
    ~path:["coserv"]
    ~get_params:Eliom_parameters.unit
    ()

let coservices_example_post =
  Eliom_services.new_post_coservice
    ~fallback:coservices_example
    ~post_params:Eliom_parameters.unit
    ()

let coservices_example_get =
  Eliom_services.new_coservice
    ~fallback:coservices_example
    ~get_params:Eliom_parameters.unit
    ()


(* ---------------------------------------------------------- *)
(* The three of them display the same page,                   *)
(* but the coservices change the counter.                     *)

let _ =
  let c = ref 0 in
  let page sp () () =
    let l3 = Eliom_predefmod.Xhtml.post_form coservices_example_post sp
        (fun _ -> [p [Eliom_predefmod.Xhtml.string_input
```

```
                        ~input_type:`Submit
                        ~value:"incr i (post)" ()]]) ()
    in
    let l4 = Eliom_predefmod.Xhtml.get_form coservices_example_get sp
        (fun _ -> [p [Eliom_predefmod.Xhtml.string_input
                        ~input_type:`Submit
                        ~value:"incr i (get)" ()]])
    in
    return
      (html
       (head (title (pcdata "")) [])
       (body [p [pcdata "i is equal to ";
                 pcdata (string_of_int !c); br ();
                 a coservices_example sp [pcdata "reload"] (); br ();
                 a coservices_example_get sp [pcdata "incr i"] ()];
              l3;
              l4]))
in
Eliom_predefmod.Xhtml.register coservices_example page;
let f sp () () = c := !c + 1; page sp () () in
Eliom_predefmod.Xhtml.register coservices_example_post f;
Eliom_predefmod.Xhtml.register coservices_example_get f
```

Note that if the coservice does not exist (for example it has expired), the fallback is called.

In this example, coservices do not take any parameters (but the state parameter), but you can create coservices with parameters. Note that the fallback of a GET coservice cannot take parameters. Actually as coservices parameters have special names, it is possible to use a "pre-applied" service as fallback (**see later**).

Exercise: Rewrite the example of Web site with connection (`session_data_example`, with session data) using a POST coservice without parameter to make the disconnection link go back to the main page of the site instead of a "disconnection" page. It is better for ergonomics, but it would be even better to stay on the same page ... How to do that with POST coservices? A much better solution will be seen in the **section about actions and non-attached services**.

URLs

While designing a Web site, think carefully about the URLs you want to use. URLs are the entry points of your site. Think that they may be bookmarked. If you create a link, you want to go to another URL, and you want a page to be generated. That page may be the default page for the URL (the one you get when you go back to a bookmarked page), or another page, that depends on the precise link or form you used to go to that URL (link to a coservice, or page depending on post data). Sometimes, you want that clicking a link or submitting a form does something without changing the URL. You can do this using *non-attached services* (see below).

Continuations

Eliom is using the concept of *continuation*. A continuation represents the future of a program (what to do after). When a user clicks on a link or a form, he chooses the future of the computation. When he uses the "back" button of the browser, he chooses to go back to an old continuation. Continuations for Web programming have been introduced by **Christian Queinnec**, and are a big step in the understanding of Web interaction.

Some programming languages (Scheme...) allow to manipulate continuations using *control operators* (like `call/cc`). The style of programming used by Eliom is closer to *Continuation Passing Style* (CPS), and has the advantage that it does not need control operators, and fits very well Web programming.

Coservices allow to create dynamically new continuations that depend on previous interactions with users (**See the `calc` example below**). Such a behaviour is difficult to simulate with traditional Web programming.

If you want continuations dedicated to a particular user register them in the session table.

Non-attached services and coservices

Non-attached (co)services are (co)services that are not attached to an URL. When you do a link or a form towards such a service, the URL do not change. The name of the service is sent as a special parameter.

Non-attached services have a name, non-attached coservices are distinguished by a number (every times different).

As for attached (co)services, there are GET and POST versions. To create them, use **`Eliom_services.new_service'`**, **`Eliom_services.new_post_service'`**, **`Eliom_services.new_coservice'`** or **`Eliom_services.new_post_coservice'`**. POST non-attached (co)services are really useful if you want a link or form to be present on every page but you don't want the URL to change. Very often, POST (co)services are used with *actions* (**see more details and an example in the section about actions below**).

Coservices in session tables

You can register coservices in session tables to create dynamically new services dedicated to an user. Here is an example of pages that add two integers. Once the first number is sent by the user, a coservice is created and registered in the session table. This service takes the second number as parameter and displays the result of the sum with the first one. Try to duplicate the pages and/or to use the back button of your navigator to verify that it has the expected behaviour.

```
(**********************************************************)
(*************** calc: sum of two integers ****************)
(**********************************************************)


(* ------------------------------------------------------ *)
(* We create two main services on the same URL,           *)
(* one with a GET integer parameter:                      *)

let calc =
  new_service
    ~path:["calc"]
    ~get_params:unit
    ()

let calc_i =
  new_service
    ~path:["calc"]
    ~get_params:(int "i")
    ()
```

```
(* ------------------------------------------------------- *)
(* The handler for the service without parameter.          *)
(* It displays a form where you can write an integer value: *)

let calc_handler sp () () =
  let create_form intname =
    [p [pcdata "Write a number: ";
        Eliom_predefmod.Xhtml.int_input ~input_type:`Text
          ~name:intname ();
        br ();
        Eliom_predefmod.Xhtml.string_input ~input_type:`Submit
          ~value:"Send" ()]]
  in
  let f = Eliom_predefmod.Xhtml.get_form calc_i sp create_form in
  return
    (html
       (head (title (pcdata "")) [])
       (body [f]))


(* ------------------------------------------------------- *)
(* The handler for the service with parameter.             *)
(* It creates dynamically and registers a new coservice    *)
(* with one GET integer parameter.                         *)
(* This new coservice depends on the first value (i)       *)
(* entered by the user.                                    *)

let calc_i_handler sp i () =
  let create_form is =
    (fun entier ->
       [p [pcdata (is^" + ");
           int_input ~input_type:`Text ~name:entier ();
           br ();
           string_input ~input_type:`Submit ~value:"Sum" ()]])
  in
  let is = string_of_int i in
  let calc_result =
    register_new_coservice_for_session
      ~sp
      ~fallback:calc
      ~get_params:(int "j")
      (fun sp j () ->
        let js = string_of_int j in
        let ijs = string_of_int (i+j) in
        return
          (html
             (head (title (pcdata "")) [])
             (body
                [p [pcdata (is^" + "^js^" = "^ijs)]])))
  in
  let f = get_form calc_result sp (create_form is) in
  return
    (html
       (head (title (pcdata "")) [])
       (body [f]))
```

```
(* ------------------------------------------------------- *)
(* Registration of main services:                          *)

let () =
  Eliom_predefmod.Xhtml.register calc   calc_handler;
  Eliom_predefmod.Xhtml.register calc_i calc_i_handler
```

Actions

Actions are services that do not generate any page. Use them to perform an effect on the server (connection/disconnection of a user, adding something in a shopping basket, delete a message in a forum, etc.). The page you link to is redisplayed after the action. For ex, when you have the same form (or link) on several pages (for ex a connection form), instead of making a version with post params of all these pages, you can use only one action, registered on a non-attached coservice. To register actions, just use the module **`Eliom_predefmod.Actions`** instead of **`Eliom_predefmod.Xhtml`** (or **`Eliom_duce.Xhtml`**, etc.). For example **`Eliom_predefmod.Actions.register`**, **`Eliom_predefmod.Actions.register_new_service`**, **`Eliom_predefmod.Actions.register_for_session`**.

Here is one simple example. Suppose you wrote a function `remove` to remove one piece of data from a database (taking an identifier of the data). If you want to put a link on your page to call this function and redisplay the page, just create an action like this:

```
let remove_action =
  Eliom_services.register_new_post_coservice'
    ~post_params:(Eliom_parameters.int "id")
    (fun sp () id -> remove id >>= fun () -> Lwt.return [])
```

Then wherever you want to add a button to do that action (on data `id`), create a form like:

```
Eliom_predefmod.Xhtml.post_form remove_action sp
  (fun id_name ->
     Eliom_predefmod.Xhtml.int_input
       ~input_type:`Hidden ~name:id_name ~value:id ();
     Eliom_predefmod.Xhtml.string_input
       ~input_type:`Submit ~value:("remove "^string_of_int id) ())
```

Here we rewrite the example `session_data_example` using actions and non-attached services (note the POST coservice for disconnection, much better than the previous solution that was using another URL).

```
(**********************************************************)
(************ Connection of users, version 3 ***************)
(**********************************************************)


(* ------------------------------------------------------- *)
(* We create one main service and two (POST) actions       *)
(* (for connection and disconnection)                      *)

let connect_example3 =
  Eliom_services.new_service
    ~path:["action"]
    ~get_params:Eliom_parameters.unit
    ()
```

```
let connect_action =
  Eliom_services.new_post_service'
    ~name:"connect3"
    ~post_params:(Eliom_parameters.string "login")
    ()

(* As the handler is very simple, we register it now: *)
let disconnect_action =
  Eliom_predefmod.Actions.register_new_post_service'
    ~name:"disconnect3"
    ~post_params:Eliom_parameters.unit
    (fun sp () () ->
      Eliom_sessions.close_session  ~sp () >>= fun () ->
      Lwt.return [])


(* ---------------------------------------------------------- *)
(* login ang logout boxes:                                    *)

let disconnect_box sp s =
  Eliom_predefmod.Xhtml.post_form disconnect_action sp
    (fun _ -> [p [Eliom_predefmod.Xhtml.string_input
                    ~input_type:`Submit ~value:s ()]]) ()

let login_box sp =
  Eliom_predefmod.Xhtml.post_form connect_action sp
    (fun loginname ->
      [p
         (let l = [pcdata "login: ";
                   Eliom_predefmod.Xhtml.string_input
                     ~input_type:`Text ~name:loginname ()]
         in l)
     ])
    ()


(* ---------------------------------------------------------- *)
(* Handler for the "connect_example3" service (main page):    *)

let connect_example3_handler sp () () =
  let sessdat = Eliom_sessions.get_volatile_session_data
    ~table:my_table ~sp () in
  return
    (html
       (head (title (pcdata "")) [])
       (body
          (match sessdat with
          | Eliom_sessions.Data name ->
              [p [pcdata ("Hello "^name); br ()];
              disconnect_box sp "Close session"]
          | Eliom_sessions.Data_session_expired
          | Eliom_sessions.No_data -> [login_box sp]
          )))


(* ---------------------------------------------------------- *)
(* Handler for connect_action (user logs in):                 *)
```

```
let connect_action_handler sp () login =
  Eliom_sessions.close_session  ~sp () >>= fun () ->
  Eliom_sessions.set_volatile_session_data  ~table:my_table ~sp login;
  return []


(* ---------------------------------------------------------- *)
(* Registration of main services:                             *)

let () =
  Eliom_predefmod.Xhtml.register
    ~service:connect_example3 connect_example3_handler;
  Eliom_predefmod.Actions.register
    ~service:connect_action connect_action_handler
```

Note that actions return a list (here empty). **See later for more advanced use**

That version of the site with connection solves the main problems of **`sessdata`**:

> Connection and disconnection stay on the same page,
> If you want a connection/disconnection form on each page, no need to create a version with POST parameters of each service.

We'll see later **how to display an error message** if the connection goes wrong, and **how to have persistent sessions** (that stay opened even if the server is re-launched).

Details on service registration

> All services created during initialisation must be registered in the public table during the initialisation phase of your module. If not, the server will not start (with an error message in the logs). Thus, there will always be a service to answer when somebody clicks on a link or a form.
> Services may be registered in the public table after initialisation with `register` only if you add the `~sp` parameter.
> If you use that for main services, you will dynamically create new URLs! This may be dangerous as they will disappear if you stop the server. Be very careful to re-create these URLs when you relaunch the server, otherwise, some external links or bookmarks will be broken!
> The use of that feature is discouraged for coservices without timeout, as such coservices will be available only until the end of the server process (and it is not possible to re-create them with the same key).
> Do not register twice the same service in the public table, and do not replace a service by a directory (or vice versa). If this happens during the initialisation phase, the server won't start. If this happens after, it will be ignored (with a warning in the logs).
> All services (but non-attached ones) must be created in a module loaded inside a `<site>` tag of the config file (because they will be attached to a directory). Not possible for modules loaded inside `<extension>` or `<library>`.
> GET coservices (whithout POST parameters) can be registered only with a main service without GET/POST parameters as fallback. But it may be a ***preapplied*** service (see below).
> Services with POST parameters (main service or coservice) can be registered with a (main or co) service without POST parameters as fallback.
> The registration of (main) services must be completed before the end of the loading of the module. It not possible to launch a (Lwt) thread that will register a service later, as registering a service needs access to config file information (for example the directory of the site). If you do this, the server will raise **`Eliom_common.Eliom_function_forbidden_outside_site_loading`**

most of the time, but you may also get unexpected results (if the thread is executed while another site is loaded). If you use threads in the initialization phase of your module (for example if you need information from a database), use **`Lwt_unix.run`** to wait the end of the thread.

3. More details on services and page generation

You now know all Eliom's main concepts. In that part, we'll give more details on some aspects that have been seen before:

The different types of output for services
Timeouts and error handling
Persistence of sessions
Advanced forms

Static parts

Fully static pages

The `staticmod` extension allows to associate to your site a static directory where you can put all the static (non generated) parts of your web-site (for examples images ans stylesheets). See the default config file `ocsigen.conf` to learn how to do that. A predefined service can be used to make links to static files. Get it using **`(static_dir ~sp)`**. That service takes as string parameter the name of the file.
For example

```
Eliom.a
  (static_dir ~sp)
  sp
  [pcdata "download image"]
  "ocsigen8-100x30.png"
```

creates this link: **download image**

It is now also possible to handle static pages with Eliom, using `Eliom_predefmod.Files` (**see later**).

Other kinds of pages

Sending portions of pages

The **`Eliom_predefmod.Blocks`** module allows to register services that send portions of pages, of any type that may be contained directly in a `<body>` tag (blocks of xhtml DTD). It is useful to create AJAX pages (i.e. pages using the `XMLHttpRequest` Javascript object). Note that the service returns a list of blocks.

```
let divpage =
  Eliom_predefmod.Blocks.register_new_service
    ~path:["div"]
    ~get_params:unit
    (fun sp () () ->
      return
        [div [h2 [pcdata "Hallo"];
              p [pcdata "Blablablabla"] ]])
```

The **Eliom_predefmod.SubXhtml** module allows to create other modules for registering portions of pages of other types. For example, **Eliom_predefmod.Blocks** is defined by:

```
module Blocks = SubXhtml(struct
  type content = Xhtmltypes.body_content
end)
```

Redirections

The **Eliom_predefmod.Redirection** module allows to register HTTP redirections.
[New in 1.1.0. For 1.0.0, please see module Eliom_predefmod.Redirections.]
If a request is done towards such a service, the server asks the browser to retry with another URL.

Such services return a GET service without parameter at all. Example:

```
let redir1 = Eliom_predefmod.Redirection.register_new_service
    ~options:`Temporary
    ~path:["redir"]
    ~get_params:Eliom_parameters.unit
   (fun sp () () -> Lwt.return coucou)
```

If you want to give parameters to such services, use **Eliom_services.preapply** (see also **later in the tutorial**). Example:

```
let redir = Eliom_predefmod.Redirection.register_new_service
    ~options:`Temporary
    ~path:["redir"]
    ~get_params:(int "o")
   (fun sp o () ->
      Lwt.return
        (Eliom_services.preapply coucou_params (o,(22,"ee"))))
```

The `options` parameter may be either `` `Temporary `` or `` `Permanent ``.

Note that the cost of a redirection is one more query and one more answer.

Sending files

You may want to register a service that will send files. To do that, use the **Eliom_predefmod.Files** module. Example:

```
let sendfile =
  Files.register_new_service
    ~path:["sendfile"]
    ~get_params:unit
    (fun _ () () -> return "filename")
```

Other example, with suffix URL:

```
let sendfile2 =
  Files.register_new_service
    ~path:["files"]
    ~get_params:(suffix (all_suffix "filename"))
    (fun _ s () -> return ("path"^
      (Ocsigen_extensions.string_of_url_path s)))
```

The extension `Staticmod` is another way to handle static files (see the default configuration file for more information).

Registering services that decide what they want to send

You may want to register a service that will send, for instance, sometimes an xhtml page, sometimes a file, sometimes something else. To do that, use the **`Eliom_predefmod.Any`** module, together with the `send` function of the module you want to use. Example:

```
let send_any =
  Eliom_predefmod.Any.register_new_service
    ~path:["sendany"]
    ~get_params:(string "type")
   (fun sp s () ->
     if s = "valid"
     then
       Eliom_predefmod.Xhtml.send sp
         (html
            (head (title (pcdata "")) [])
            (body [p [pcdata
              "This page has been statically typechecked.
               If you change the parameter in the URL you will get \
               an unchecked text page"]]))
     else
       Eliom_predefmod.HtmlText.send sp
         "<html><body><p>It is not a valid page. \
Put type=\"valid\" in the URL \
to get a typechecked page.</p></body></html>"
   )
```

You may also use **`Eliom_predefmod.Any`** to send cookies or to choose a different charset than the default (default charset is set in configuration file) for the page you send. To do that use the optional parameters `?cookies` and `?charset` of the `send` function.

Cookies

A simplest way to set your own cookies on the client is to use functions like `Eliom_predefmod.Xhtml.Cookies.register` instead of `Eliom_predefmod.Xhtml.register`. The function you register returns a pair containing the page (as usual) and a list of cookies, of type `Eliom_services.cookie` defined by:

```
type cookie =
  | Set of string list option * float option * string * string * bool
  | Unset of string list option * string
```

[New in 1.1.0] For version 1.0.0, the type `cookie` was slightly different (no secure cookies).

The `string list option` is a the path for which you want to set/unset the cookie (relative to the main directory of your site, defined in the configuration file). `None` means for all your site.

The `float option` is a the expiration date (Unix timestamp, in seconds since the epoch). `None` means that the cookie will expire when the browser will be closed.

If the `bool` is true and the protocol is https, the server will ask the browser to send the cookie only through secure connections.

You can access the cookies sent by the browser using **`Eliom_sessions.get_cookies sp`**.

Example:

```
let cookiename = "mycookie"

let cookies = new_service ["cookies"] unit ()

let _ = Cookies.register cookies
    (fun sp () () ->
      return
       ((html
         (head (title (pcdata "")) [])
         (body [p [pcdata
           (try
             "cookie value: "^
             (Ocsigen_http_frame.Cookievalues.find
               cookiename (Eliom_sessions.get_cookies sp))
           with _ -> "<cookie not set>");
                  br ();
                  a cookies sp [pcdata "send other cookie"] ()]])),
        [Eliom_services.Set (None, None,
                             cookiename,
                             string_of_int (Random.int 100),
                             false)]))
```

Persistence of sessions

Tables of sessions (for data or services) are kept in memory, and thus will disappear if you close the server process. To solve this problem, Ocsigen allows to reload the modules of your configuration file without shutting down the server. Another solution provided by Eliom is to save session data on hard disk.

Updating sites without shutting down the server

To reload the modules of the configuration file without stoping the server, use `/etc/init.d/ocsigen reload` for most of the distributions, or do it manually using:

```
echo reload > /var/run/ocsigen_command
```

.
Only modules loaded inside `<site>` or `<library>` will be reloaded. Module loaded using `<extension>` will not.

Have a look at the logs to see if all went well during the reload. If something went wrong, old services may still be reachable.

Note that coservices created with the old modules or URLs that have not been masked by new ones will still reachable after the update.

During the reload, some information of the configuration file will not be re-read (for example port numbers, user and group, etc.).

Persistent data

Eliom allows to use more persistent data, using the module **`Ocsipersist`**. (`Ocsipersist` is needed in `eliom.cma`, thus you need to dynlink it in the configuration file before `Eliom`). There are currently two implementations of `Ocsipersist`: `ocsipersist-dbm.cma` (uses the DBM database) and `ocsipersist-sqlite.cma` (uses the SQLite database, and depends on `sqlite3.cma`).

These modules allow to:

Create persistent references (still present after restarting the server),
Create persistent association tables,
Set persistent session data (using `set_persistent_data`, see below).

Note that persistent data are serialized on hard disk using OCaml's `Marshal` module:

> **It is not possible to serialize closures or services (as we are using dynamic linking). If you ever change the type of serialised data, don't forget to delete the database file! Or if you really want to keep it, and you know what you are doing, you can use the sqlite client to manually update the table or a program to create a new sqlite or dbm table for the new type.**

Suppose for example that you use `get/set_persistent_data` (see below) to store a (int, string) tuple with the user's login credentials. At this point you stop the server, and change the code such that get/set_persistent_data now to store a (int, string, string). Now recompile and restart the server. If by any chance a client with an old cookie reconnects, you get a segfault on the server, because of the type change in the data stored in the DB backend ...

Persistent references

`Ocsipersist` allows to create persistent references. Here is an example of page with a persistent counter:

```
let mystore = Ocsipersist.open_store "eliomexamplestore2"

let count2 =
  let next =
    let cthr = Ocsipersist.make_persistent mystore "countpage" 0 in
    let mutex = Lwt_mutex.create () in
    (fun () ->
      cthr >>=
      (fun c ->
        Lwt_mutex.lock mutex >>= fun () ->
        Ocsipersist.get c >>=
        (fun oldc ->
          let newc = oldc + 1 in
          Ocsipersist.set c newc >>=
          (fun () ->
            Lwt_mutex.unlock mutex;
            return newc))))
  in
  register_new_service
    ~path:["count2"]
    ~get_params:unit
    (fun _ () () ->
      next () >>=
      (fun n ->
        return
         (html
          (head (title (pcdata "counter")) [])
          (body [p [pcdata (string_of_int n)]])))))
```

Persistent tables

`Ocsipersist` also allows to create very basic persistent tables. Use them if you don't need complex requests on your tables. Otherwise use a database such as `PostgreSQL` or `MySQL`. Here are the interface you can use:

```
type 'value table

val open_table : string -> 'value table

val find : 'value table -> string -> 'value Lwt.t

val add : 'value table -> string -> 'value -> unit Lwt.t

val remove : 'value table -> string -> unit Lwt.t
```

As you can see, all these function are cooperative.

Persistent session data

`Eliom` also implements persistent session tables. You can use them instead of memory tables if you don't need to register closures.

The following example is a new version of our site with users, with persistent connections. (`login_box`, `disconnect_box` and `disconnect_action` are the same as **before**).

```
(***************************************************************)
(************ Connection of users, version 4 ***************)
(*************** (persistent sessions) ********************)
(***************************************************************)


let my_persistent_table =
  create_persistent_table "eliom_example_table"

(* ------------------------------------------------------------ *)
(* We create one main service and two (POST) actions            *)
(* (for connection and disconnection)                           *)

let persist_session_example =
  Eliom_services.new_service
    ~path:["persist"]
    ~get_params:unit
    ()

let persist_session_connect_action =
  Eliom_services.new_post_service'
    ~name:"connect4"
    ~post_params:(string "login")
    ()

(* disconnect_action, login_box and disconnect_box have been
   defined in the section about actions *)

(* ------------------------------------------------------------- *)
(* Handler for "persist_session_example" service (main page):  *)

let persist_session_example_handler sp () () =
  Eliom_sessions.get_persistent_session_data
    ~table:my_persistent_table ~sp () >>= fun sessdat ->
```

```
  return
    (html
       (head (title (pcdata "")) [])
       (body
          (match sessdat with
          | Eliom_sessions.Data name ->
              [p [pcdata ("Hello "^name); br ()];
              disconnect_box sp "Close session"]
          | Eliom_sessions.Data_session_expired ->
              [login_box sp true persist_session_connect_action;
               p [em [pcdata "The only user is 'toto'."]]]
          | Eliom_sessions.No_data ->
              [login_box sp false persist_session_connect_action;
               p [em [pcdata "The only user is 'toto'."]]]
          )))


(* ---------------------------------------------------------- *)
(* Handler for persist_session_connect_action (user logs in):  *)

let persist_session_connect_action_handler sp () login =
  Eliom_sessions.close_session  ~sp () >>= fun () ->
  if login = "toto" (* Check user and password :-) *)
  then begin
    Eliom_sessions.set_persistent_session_data
      ~table:my_persistent_table ~sp login >>= fun () ->
    return []
  end
  else return [Bad_user]


(* ---------------------------------------------------------- *)
(* Registration of main services:                             *)

let () =
  Eliom_predefmod.Xhtml.register
    ~service:persist_session_example
    persist_session_example_handler;
  Eliom_predefmod.Actions.register
    ~service:persist_session_connect_action
    persist_session_connect_action_handler
```

As it is not possible to serialize closures, there is no persistent session service table. Be very carefull if you use both persistent session data tables and service session tables, as your session may become inconsistent (use the session service table only for volatile services, like coservices with timeouts).

[New in 0.99.5 - EXPERIMENTAL] Session groups

The idea is complementary to that of the "session name". While the optional `session_name` parameter allows for a single session to have multiple buckets of data associated with it, a session_group parameter (also optional) allow multiple sessions to be referenced together. For most uses, the session group is the user name. It allows to implement features like "close all sessions" for one user (even those opened on other browsers), or to limit the number of sessions one user may open at the same time.

Session groups have been suggested by Dario Teixeira and introduced in Eliom 0.99.5. Dario explains: *Consider the following scenario: a user logs in from home using a "Remember me on this computer" feature, which sets a (almost) no-expiration cookie on his browser and session timeouts of infinity on the server. The user goes on vacation, and while logging from a cyber-café, she also sets the "Remember me" option. Back home she realises her mistake, and wishes to do a "global logout", ie, closing all existing sessions associated with her user name.*

```
(**********************************************************)
(************ Connection of users, version 5 ***************)
(**********************************************************)


(* ------------------------------------------------------- *)
(* We create one main service and two (POST) actions       *)
(* (for connection and disconnection)                      *)

let connect_example5 =
  Eliom_services.new_service
    ~path:["groups"]
    ~get_params:Eliom_parameters.unit
    ()

let connect_action =
  Eliom_services.new_post_service'
    ~name:"connect5"
    ~post_params:(Eliom_parameters.string "login")
    ()

(* As the handler is very simple, we register it now: *)
let disconnect_action =
  Eliom_predefmod.Actions.register_new_post_service'
    ~name:"disconnect5"
    ~post_params:Eliom_parameters.unit
    (fun sp () () ->
      Eliom_sessions.close_session  ~sp () >>= fun () ->
      Lwt.return [])


(* ------------------------------------------------------- *)
(* login ang logout boxes:                                 *)

let disconnect_box sp s =
  Eliom_predefmod.Xhtml.post_form disconnect_action sp
    (fun _ -> [p [Eliom_predefmod.Xhtml.string_input
                    ~input_type:`Submit ~value:s ()]]) ()

let login_box sp =
  Eliom_predefmod.Xhtml.post_form connect_action sp
    (fun loginname ->
      [p
         (let l = [pcdata "login: ";
                   Eliom_predefmod.Xhtml.string_input
                     ~input_type:`Text ~name:loginname ()]
         in l)
     ])
    ()
```

```
(* ------------------------------------------------------- *)
(* Handler for the "connect_example5" service (main page):    *)

let connect_example5_handler sp () () =
  let sessdat =
    Eliom_sessions.get_volatile_data_session_group  ~sp () in
  return
    (html
       (head (title (pcdata "")) [])
       (body
          (match sessdat with
          | Eliom_sessions.Data name ->
              [p [pcdata ("Hello "^name); br ()];
              disconnect_box sp "Close session"]
          | Eliom_sessions.Data_session_expired
          | Eliom_sessions.No_data -> [login_box sp]
          )))


(* ------------------------------------------------------- *)
(* Handler for connect_action (user logs in):                *)

let connect_action_handler sp () login =
  Eliom_sessions.close_session  ~sp () >>= fun () ->
  Eliom_sessions.set_volatile_data_session_group
    ~set_max:(Some 10)  ~sp login;
  return []


(* ------------------------------------------------------- *)
(* Registration of main services:                          *)

let () =
  Eliom_predefmod.Xhtml.register
    ~service:connect_example5 connect_example5_handler;
  Eliom_predefmod.Actions.register
    ~service:connect_action connect_action_handler
```

As we will see later, there are three kinds of sessions (services, volatile data and persistent data). It is highly recommended to set a group for each of them!

Other concepts

Pre-applied services

Services or coservices with GET parameters can be preapplied to obtain a service without parameters. Example:

```
let preappl = Eliom_services.preapply coucou_params (3,(4,"cinq"))
```

It is not possible to register something on a preapplied service, but you can use them in links or as fallbacks for coservices.

Void action [New in 1.1.0]

`Eliom_services.void_action`: is a special non-attached action, with special behaviour: it has no parameter at all, even non-attached parameters. Use it if you want to make a link to the current page without non-attached parameters. It is almost equivalent to a POST non-attached service without POST parameters, on which you register an action that does nothing, but you can use it with `<a>` links, not only forms. Example:

```
Eliom_duce.Xhtml.a
  ~service:Eliom_services.void_action
  ~sp
  {{ "cancel" }}
  ()
```

Giving information to fallbacks

Fallbacks have access to some information about what succeeded before they were called. Get this information using **`Eliom_sessions.get_exn sp`**; That function returns a list of exceptions. That list contains **`Eliom_common.Eliom_Link_too_old`** if the coservice was not found, and **`Eliom_common.Eliom_Service_session_expired`** if the "service session" has expired.

It is also possible to tell actions to send information to the page generated after them. Just place exceptions in the list returned by the action. These exceptions will also be accessible with **`Eliom_sessions.get_exn`**. Here is the new version of the **example of session with actions:** by:

```
(************************************************************)
(************ Connection of users, version 6 ****************)
(************************************************************)

(* ---------------------------------------------------------- *)
(* We create one main service and two (POST) actions          *)
(* (for connection and disconnection)                         *)

let connect_example6 =
  Eliom_services.new_service
    ~path:["action2"]
    ~get_params:unit
    ()

let connect_action =
  Eliom_services.new_post_service'
    ~name:"connect6"
    ~post_params:(string "login")
    ()

(* new disconnect action and box:                             *)

let disconnect_action =
  Eliom_predefmod.Actions.register_new_post_service'
    ~name:"disconnect6"
    ~post_params:Eliom_parameters.unit
    (fun sp () () ->
      Eliom_sessions.close_session  ~sp () >>= fun () ->
      return [])

let disconnect_box sp s =
```

```
  Eliom_predefmod.Xhtml.post_form disconnect_action sp
    (fun _ -> [p [Eliom_predefmod.Xhtml.string_input
                    ~input_type:`Submit ~value:s ()]]) ()


exception Bad_user

(* ---------------------------------------------------------- *)
(* new login box:                                             *)

let login_box sp session_expired action =
  Eliom_predefmod.Xhtml.post_form action sp
    (fun loginname ->
      let l =
        [pcdata "login: ";
         string_input ~input_type:`Text ~name:loginname ()]
      in
      let exnlist = Eliom_sessions.get_exn sp in
      (* If exnlist is not empty, something went wrong
         during an action. We write an error message: *)
      [p (if List.mem Bad_user exnlist
      then (pcdata "Wrong user")::(br ())::l
      else
        if session_expired
        then (pcdata "Session expired")::(br ())::l
        else l)
     ])
    ()

(* ---------------------------------------------------------- *)
(* Handler for the "connect_example6" service (main page):    *)

let connect_example6_handler sp () () =
  let group =
    Eliom_sessions.get_volatile_data_session_group  ~sp ()
  in
  return
    (html
       (head (title (pcdata "")) [])
       (body
          (match group with
          | Eliom_sessions.Data name ->
              [p [pcdata ("Hello "^name); br ()];
              disconnect_box sp "Close session"]
          | Eliom_sessions.Data_session_expired ->
              [login_box sp true connect_action;
               p [em [pcdata "The only user is 'toto'."]]]
          | Eliom_sessions.No_data ->
              [login_box sp false connect_action;
               p [em [pcdata "The only user is 'toto'."]]]
          )))

(* ---------------------------------------------------------- *)
(* New handler for connect_action (user logs in):             *)

let connect_action_handler sp () login =
  Eliom_sessions.close_session  ~sp () >>= fun () ->
  if login = "toto" (* Check user and password :-) *)
  then begin
```

```
    Eliom_sessions.set_volatile_data_session_group
      ~set_max:(Some 10)  ~sp login;
    return []
  end
  else return [Bad_user]

(* ---------------------------------------------------------- *)
(* Registration of main services:                             *)

let () =
  Eliom_predefmod.Xhtml.register
    ~service:connect_example6 connect_example6_handler;
  Eliom_predefmod.Actions.register
    ~service:connect_action connect_action_handler
```

If the actions raises an exception (with **`Lwt.fail`**), the server will send an error 500 (like for any other service). Think about catching the exceptions and put them in the list if they correspond to usual cases you want to handle while generating the page after the action.

Disposable coservices

It is possible to set a limit to the number of uses of (attached or non-attached) coservices. Just give the maximum number of uses with the optional `?max_use` parameter while creating your coservices. Example

```
let disposable = new_service ["disposable"] unit ()

let _ = register disposable
    (fun sp () () ->
      let disp_coservice =
        new_coservice
          ~max_use:2 ~fallback:disposable ~get_params:unit ()
      in
      register_for_session sp disp_coservice
        (fun sp () () ->
          return
            (html
              (head (title (pcdata "")) [])
              (body [p [pcdata "I am a disposable coservice";
                        br ();
                        a disp_coservice sp [pcdata
                           "Try me once again"] ()]]))
        );
      return
        (html
          (head (title (pcdata "")) [])
          (body [p [(if List.mem
                         Eliom_common.Eliom_Link_too_old
                         (Eliom_sessions.get_exn sp)
                     then pcdata "Your link was outdated. \
I am the fallback. I just created a new disposable coservice. \
You can use it only twice."
                     else
                     pcdata "I just created a disposable coservice. \
You can use it only twice.");
                     br ();
                     a disp_coservice sp [pcdata "Try it!"] ()]])))
```

Timeout for sessions

The default timeout for sessions in one hour. Sessions will be automatically closed after that amount of time of inactivity from the user. You can change that value for your whole site during initialisation using:

```
Eliom_sessions.set_global_volatile_timeout (Some 7200.)
```

Here 7200 seconds. `None` means no timeout.

You can change that value for your whole site after initialisation using:

```
Eliom_sessions.set_global_volatile_timeout ~sp (Some 7200.)
```

You can change that value for one user only using:

```
Eliom_sessions.set_volatile_session_timeout ~sp (Some 7200.)
```

Note that there is also a possibility to change the default value for Eliom in the configuration file like this:

```
    <extension module="path_to/eliom.cma">
      <timeout value="7200"/>
    </extension>
```

`value="infinity"` means no timeout.

Warning: that default may be overriden by each site using **`Eliom_sessions.set_global_volatile_timeout`** or **`Eliom_sessions.set_default_volatile_timeout`**. If you want your user to be able to set the default in the configuration file for your site (between `<site>` and `</site>`), you must parse the configuration (**`Eliom_sessions.get_config ()`** function, see below).

Timeout for coservices

It is also possible to put timeouts on coservices using the optional parameter `?timeout` of functions `new_coservice`, `new_coservice'`, etc. Note that session coservices cannot survive after the end of the session. Use this if you don't want your coservice to be available during all the session duration. For example if your coservice is here to show the results of a search, you probably want it to be available only for a short time. The following example shows a coservice with timeout registered in the session table.

```
let timeout = new_service ["timeout"] unit ()

let _ =
  let page sp () () =
    let timeoutcoserv =
      register_new_coservice_for_session
        ~sp ~fallback:timeout ~get_params:unit ~timeout:5.
        (fun _ _ _ ->
           return
             (html
               (head (title (pcdata "Coservices with timeouts")) [])
               (body [p
                 [pcdata "I am a coservice with timeout."; br ();
                  pcdata "Try to reload the page!"; br ();
                  pcdata "I will disappear after 5 seconds of \
inactivity." ];
                 ])))
    in
    return
      (html
        (head (title (pcdata "Coservices with timeouts")) [])
        (body [p
          [pcdata "I just created a coservice with 5 seconds \
timeout."; br ();
           a timeoutcoserv sp [pcdata "Try it"] (); ];
          ]))
  in
  register timeout page
```

Registering coservices in public table during session

If you want to register coservices in the public table during a session, (that is, after the initialisation phase of your module), you must add the optional `~sp` parameter to the `register` function. Remember that using `register` without `~sp` is possible only during initialisation!

We recommend to put a timeout on such coservices, otherwise, they will be available until the end of the server process, and it will not be possible to re-create them when the server is relaunched.

The following example is a translation of the previous one using the public table:

```
let publiccoduringsess =
  new_service ["publiccoduringsess"] unit ()

let _ =
  let page sp () () =
    let timeoutcoserv =
      register_new_coservice
        ~sp ~fallback:publiccoduringsess
        ~get_params:unit ~timeout:5.
        (fun _ _ _ ->
           return
             (html
               (head (title (pcdata
                 "Coservices with timeouts")) [])
               (body [p
                 [pcdata "I am a public coservice with timeout.";
                 br ();
```

```
                  pcdata "I will disappear after 5 seconds of \
inactivity." ];
                ])))
    in
    return
      (html
        (head (title (pcdata "Public coservices with timeouts")) [])
        (body [p
          [pcdata "I just created a public coservice with \
5 seconds timeout."; br ();
           a timeoutcoserv sp [pcdata "Try it"] (); ];
          ]))
  in
  register publiccoduringsess page
```

Defining an exception handler for the whole site

When an exception is raised during the generation of a page, or when the page has not been found or has wrong parameters, an HTTP error 500 or 404 is sent to the client. You may want to catch these exceptions to print your own error page. Do this using **`Eliom_services.set_exn_handler`**. Here is the handler used by this tutorial:

```
let _ = Eliom_services.set_exn_handler
   (fun sp e -> match e with
    | Eliom_common.Eliom_404 ->
        Eliom_predefmod.Xhtml.send ~code:404 ~sp
          (html
             (head (title (pcdata "")) [])
             (body [h1 [pcdata "Eliom tutorial"];
                    p [pcdata "Page not found"]]))
    | Eliom_common.Eliom_Wrong_parameter ->
        Eliom_predefmod.Xhtml.send ~sp
          (html
             (head (title (pcdata "")) [])
             (body [h1 [pcdata "Eliom tutorial"];
                    p [pcdata "Wrong parameters"]]))
    | e -> fail e)
```

Giving configuration options to your sites

You can add your own options in the configuration file for your Web site. For example:

```
    <eliom module="path_to/yourmodule.cmo">
      <youroptions> ...
    </eliom>
```

Use **`Eliom_sessions.get_config ()`** during the initialization of your module to get the data between `<eliom>` and `</eliom>`. Warning: parsing these data is very basic for now. That feature will be improved in the future.

More about sessions

By default, Eliom is using three cookies :

> One for session services,
> one for volatile session data,
> one for persistent session data.

They correspond to three different sessions (opened only if needed). **`Eliom_sessions.close_session`** closes all three sessions, but you may want to

desynchronize the three sessions by using **`Eliom_sessions.close_persistent_session`** (persistent session), **`Eliom_sessions.close_service_session`** (session services), or **`Eliom_sessions.close_data_session`** (volatile data session). There is also **`Eliom_sessions.close_volatile_session`** for both volatile data session and session services. The module **`Eliom_sessions`** also contains functions for setting timeouts or expiration dates for cookies for each kind of session.

If you need more sessions (for example several different data sessions) for the same site, you can give a name to your sessions by giving the optional parameter `?session_name` to functions like **`Eliom_sessions.close_data_session`**, **`register_for_session`**, or **`Eliom_sessions.get_volatile_session_data`**. Note that this tutorial has been implemented using this feature, even if it has been hidden for the sake of simplicity. That's how the different examples of sessions in this tutorial are independant.

Secure services [New in 1.1.0]

You may want to impose HTTPS for some of your services. To do that, use the optional parameter `~https:true` while creating your service.

It is also possible to require http or https while creating a link or a form (using the optional parameter `~https:true`). But it is never possible to make an http link towards an https service, even if you request it.

Warning: if the protocol needs to be changed (from http to https or vice versa), Eliom will generate absolute URLs. The host name and port numbers are guessed from the IP and the configuration by default, but it is recommended to specify them in the configuration file. For example:

```
<host name="*.org" defaulthostname="www.mywebsite.org"
defaulthttpport="8080" defaulthttpsport="4433"> ... </host>
```

Secure sessions [New in 1.1.0]

For security reasons, Eliom does not use the same cookies in https and http. Secure sessions are using secure cookies (i.e. Ocsigen will ask the browsers to send the cookie only if the protocol is secure). Thus it is not possible to access secure session if the user is using http. If the user is using https, Eliom will save data and services in secure session. But it is possible to access unsecure session data and to register unsecure session services using the optional parameter `~secure:false` when calling functions like `Eliom_sessions.set_volatile_session_data`, `Eliom_sessions.get_persistent_session_data`, `Eliom_predefmod.Xhtml.register_for_session`, etc.

Advanced forms and parameters

This section shows more advanced use of page parameters and corresponding forms.

Parsing parameters using regular expressions

Eliom_parameters.regexp allows to parse page parameters using (Perl-compatible) regular expressions. We use the module `Netstring_pcre`, from *OCamlnet*. See the documentation about OCamlnet for more information. The following example shows a service that accepts only parameters values enclosed between `[` and `]`:

```
let r = Netstring_pcre.regexp "\\\\[(.*)\\\\]"

let regexp =
  Eliom_predefmod.Xhtml.register_new_service
    ~path:["regexp"]
    ~get_params:(regexp r "$1" "myparam")
    (fun _ g () ->
      return
        (html
           (head (title (pcdata "")) [])
           (body [p [pcdata g]])))
```

```

```

Boolean checkboxes

Page may take parameter of type `bool`. A possible use of this type is in a form with *boolean checkboxes*, as in the example below:

```
(* Form with bool checkbox: *)
let bool_params = register_new_service
    ~path:["bool"]
    ~get_params:(bool "case")
  (fun _ case () ->
    return
    << <html>
         <head><title></title></head>
         <body>
         <p>
           $pcdata (if case then "checked" else "not checked")$
         </p>
         </body>
       </html> >>)

let create_form_bool casename =
    <:xmllist< <p>check? $bool_checkbox ~name:casename ()$ <br/>
      $string_input ~input_type:`Submit
        ~value:"Click" ()$</p> >>

let form_bool = register_new_service ["formbool"] unit
  (fun sp () () ->
     let f = get_form bool_params sp create_form_bool in
     return
     << <html>
          <head><title></title></head>
          <body> $f$ </body>
        </html> >>)
```

Important warning: As you can see, browsers do not send any value for unchecked boxes! An unchecked box is equivalent to no parameter at all! Thus it is not possible to distinguish between a service taking a boolean and a service taking no parameter at all (if they share the same URL). In Eliom *services are tried in order of registration!* The first matching service will answer.

Other types similar to bool:

- **`Eliom_parameters.opt`** (page taking an optional parameter),

Eliom_parameters.sum (either a parameter or another).

Type `set`

Page may take several parameters of the same name. It is useful when you want to create a form with a variable number of fields. To do that with Eliom, use the type **Eliom_parameters.set**. For example `set int "val"` means that the page will take zero, one or several parameters of name `"val"`, all of type `int`. The function you register will receive the parameters in a list. Example:

```
let set = register_new_service
    ~path:["set"]
    ~get_params:(set string "s")
  (fun _ l () ->
    let ll =
      List.map
        (fun s -> << <strong>$str:s$ </strong> >>) l
    in
    return
    << <html>
         <head><title></title></head>
         <body>
         <p>
           You sent:
           $list:ll$
         </p>
         </body>
       </html> >>)
```

These parameters may come from several kinds of widgets in forms. Here is an example of a form with several checkboxes, all sharing the same name, but with different values:

```
(* form to set *)
let setform = register_new_service
    ~path:["setform"]
    ~get_params:unit
    (fun sp () () ->
      return
        (html
           (head (title (pcdata "")) [])
           (body [h1 [pcdata "Set Form"];
                  get_form set sp
                    (fun n ->
                      [p [pcdata "Form to set: ";
                          string_checkbox ~name:n ~value:"box1" ();
                          string_checkbox
                            ~name:n ~value:"box2" ~checked:true ();
                          string_checkbox ~name:n ~value:"box3" ();
                          string_checkbox ~name:n ~value:"box4" ();
                          string_input ~input_type:`Submit
                            ~value:"Click" ()]])
                 ])))
```

Once again, note that there is no difference between an empty set or no parameter at all. If you register a service without parameters and a service with a set of parameters on the same URL, the firstly registered service that matches will answer.

Select

Here is an example of a select box.

```
let select_example_result = register_new_service
    ~path:["select"]
    ~get_params:(string "s")
    (fun sp g () ->
      return
        (html
           (head (title (pcdata "")) [])
           (body [p [pcdata "You selected: ";
                     strong [pcdata g]]])))

let create_select_form =
  (fun select_name ->
    [p [pcdata "Select something: ";
        Eliom_predefmod.Xhtml.string_select ~name:select_name
          (Eliom_predefmod.Xhtml.Option ([] (* attributes *),
            "Bob" (* value *),
            None (* Content, if different from value *),
            false (* not selected *))) (* first line *)
          [Eliom_predefmod.Xhtml.Option ([], "Marc", None, false);
          (Eliom_predefmod.Xhtml.Optgroup
          ([],
           "Girls",
           ([], "Karin", None, false),
           [([a_disabled `Disabled], "Juliette", None, false);
            ([], "Alice", None, true);
            ([], "Germaine", Some (pcdata "Bob's mother"), false)]))]
          ;
        Eliom_predefmod.Xhtml.string_input
          ~input_type:`Submit ~value:"Send" ()]])

let select_example = register_new_service ["select"] unit
  (fun sp () () ->
     let f =
       Eliom_predefmod.Xhtml.get_form
         select_example_result sp create_select_form
     in
     return
       (html
         (head (title (pcdata "")) [])
         (body [f])))
```

To do "multiple" select boxes, use functions like **`Eliom_predefmod.Xhtml.string_multiple_select`**. As you can see in the type, the service must be declared with parameters of type **`set`**.

Clickable images

Here is an example of clickable image. You receive the coordinates the user clicked on.

```
let coord = register_new_service
    ~path:["coord"]
    ~get_params:(coordinates "coord")
  (fun _ c () ->
    return
  << <html>
       <head><title></title></head>
```

```
       <body>
       <p>
         You clicked on coordinates:
         ($str:(string_of_int c.abscissa)$,
          $str:(string_of_int c.ordinate)$)
       </p>
       </body>
     </html> >>)

(* form to image *)
let imageform = register_new_service
    ~path:["imageform"]
    ~get_params:unit
    (fun sp () () ->
      return
        (html
           (head (title (pcdata "")) [])
           (body [h1 [pcdata "Image Form"];
                  get_form coord sp
                    (fun n ->
                      [p [image_input
                            ~src:(make_uri ~service:(static_dir sp)
                              ~sp ["ocsigen5.png"])
                            ~name:n
                            ()]])
                ])))
```

You may also send a value with the coordinates:

```
let coord2 = register_new_service
    ~path:["coord2"]
    ~get_params:(int_coordinates "coord")
  (fun _ (i, c) () ->
    return
  << <html>
       <head><title></title></head>
       <body>
       <p>
         You clicked on coordinates:
         ($str:(string_of_int c.abscissa)$,
          $str:(string_of_int c.ordinate)$)
       </p>
       </body>
     </html> >>)

(* form to image *)
let imageform2 = register_new_service
    ~path:["imageform2"]
    ~get_params:unit
    (fun sp () () ->
      return
        (html
           (head (title (pcdata "")) [])
           (body [h1 [pcdata "Image Form"];
                  get_form coord2 sp
                    (fun n ->
                      [p [int_image_input
                            ~src:(make_uri ~service:(static_dir sp)
                              ~sp ["ocsigen5.png"])
```

```
                              ~name:n
                              ~value:3
                              ()]])
                ])))
```

Type `list`

Another way (than **`Eliom_parameters.set`**) to do variable length forms is to use indexed lists (using **`Eliom_parameters.list`**). The use of that feature is a bit more complex than `set` and still experimental. Here is an example of service taking an indexed list as parameter:

```
(* lists *)
let coucou_list = register_new_service
    ~path:["coucou"]
    ~get_params:(list "a" (string "str"))
  (fun _ l () ->
    let ll =
      List.map (fun s -> << <strong>$str:s$</strong> >>) l in
    return
      << <html>
           <head><title></title></head>
           <body>
           <p>
             You sent:
             $list:ll$
           </p>
           </body>
         </html> >>)
```

Here is an example of link towards this service: **coucou?a.str[0]=toto&a.str[1]=titi**.

Warning: As for sets or bools, if a request has no parameter, it will be considered as the empty list. Services are tried in order of registration.

As you see, the names of each list element is built from the name of the list, the name of the list element, and an index. To spare you creating yourself these names, Eliom provides you an iterator to create them.

```
(* Form with list: *)
let create_listform f =
  (* Here, f.it is an iterator like List.map,
     but it must be applied to a function taking 2 arguments
     (unlike 1 in map), the first one being the name of the
     parameter, and the second one the element of list.
     The last parameter of f.it is the code that must be
     appended at the end of the list created
   *)
  f.it (fun stringname v ->
    <:xmllist< <p>Write the value for $str:v$:
      $string_input ~input_type:`Text ~name:stringname ()$ </p> >>)
    ["one";"two";"three";"four"]
    <:xmllist< <p>$string_input ~input_type:`Submit
                        ~value:"Click" ()$</p> >>
```

```
let listform = register_new_service ["listform"] unit
  (fun sp () () ->
     let f = get_form coucou_list sp create_listform in
     return
      << <html>
           <head><title></title></head>
           <body> $f$ </body>
         </html> >>)
```

Important warning: As we have seen in the section about boolean (or optional) parameters, it is not possible to distinguish between a boolean with value "false", and no parameter at all. This causes problems if you create a list of boolean or optional values, as it is not possible to know the length of the list. In that case, Eliom always takes the shortest possible list.

Forms and suffixes

Service with "suffix" URLs have an equivalent version with usual parameters, allowing to create forms towards such services. Example:

```
(* Form for service with suffix: *)
let create_suffixform ((suff, endsuff),i) =
    <:xmllist< <p>Write the suffix:
      $int_input ~input_type:`Text ~name:suff ()$ <br/>
      Write a string: $user_type_input
         ~input_type:`Text ~name:endsuff
         Ocsigen_extensions.string_of_url_path$ <br/>
      Write an int: $int_input ~input_type:`Text ~name:i ()$ <br/>
      $string_input ~input_type:`Submit ~value:"Click" ()$</p> >>

let suffixform = register_new_service ["suffixform"] unit
  (fun sp () () ->
     let f = get_form isuffix sp create_suffixform in
     return
      << <html>
           <head><title></title></head>
           <body> $f$ </body>
         </html> >>)
```

Uploading files

The **`Eliom_parameters.file`** parameter type allows to send files in your request. The service gets something of type **`Ocsigen_extensions.file_info`**. You can extract information using this using these functions (from **`Eliom_sessions`**):

```
val get_tmp_filename : Ocsigen_extensions.file_info -> string
val get_filesize : Ocsigen_extensions.file_info -> int64
val get_original_filename : Ocsigen_extensions.file_info -> string
```

`Eliom_sessions.get_tmp_filename` allows to know the actual name of the uploaded file on the hard disk. **`Eliom_sessions.get_original_filename`** gives the original filename.

To make possible the upload of files, you must configure a directory for uploaded files in Ocsigen's configuration file. For example:

```
  <uploaddir>/tmp</uploaddir>
```

Files are kept in this directory only during the request. Then they are automatically cancelled. Thus your services must copy them somewhere else themselves if they want to keep them. In the following example, we create a new hard link to the file to keep it (the destination must be on the same partition of the disk).

```
let upload = new_service
    ~path:["upload"]
    ~get_params:unit
    ()

let upload2 = register_new_post_service
   ~fallback:upload
   ~post_params:(file "file")
    (fun _ () file ->
      let to_display =
        let newname = "/tmp/thefile" in
        (try
          Unix.unlink newname;
        with _ -> ());
        Unix.link (Eliom_sessions.get_tmp_filename file) newname;
        let fd_in = open_in newname in
        try
          let line = input_line fd_in in close_in fd_in; line (*end*)
        with End_of_file -> close_in fd_in; "vide"
      in
      return
        (html
           (head (title (pcdata "Upload")) [])
           (body [h1 [pcdata to_display]])))


let uploadform = register upload
    (fun sp () () ->
      let f =
        (post_form upload2 sp
           (fun file ->
             [p [file_input ~name:file ();
                 br ();
                 string_input ~input_type:`Submit ~value:"Send" ()
               ]]) ()) in
      return
        (html
           (head (title (pcdata "form")) [])
           (body [f])))
```

Predefined constructs

Images, CSS, Javascript

To include an image, simply use the function **`XHTML.M.img`**:

```
img ~alt:"Ocsigen"
    ~src:(Eliom_predefmod.Xhtml.make_uri ~service:senddoc ~sp
```

```
      ["ocsigen1024.jpg"])
    ()
```

The function **Eliom_predefmod.Xhtml.make_uri** creates the relative URL string from current URL (in `sp`) (see above) to the URL of the image in the static directory configured in the configuration file.

To simplify the creation of `<link>` tags for CSS or `<script>` tags for Javascript, use the following functions:

```
css_link ~uri:(make_uri ~service:(static_dir sp) ~sp ["style.css"]) ()
```

```
js_script ~uri:(make_uri ~service:(static_dir sp) ~sp ["funs.js"]) ()
```

Basic menus

To make a menu on your web page, you can use the function **Eliom_tools.menu**. First, define your menu like this:

```
let mymenu current sp =
  Eliom_tools.menu ~classe:["menuprincipal"]
    (home, <:xmllist< Home >>)
    [
     (infos, <:xmllist< More info >>);
     (tutorial, <:xmllist< Documentation >>)
   ] current sp
```

Here, `home`, `infos`, and `tutorial` are your three pages (generated for example by **Eliom_services.new_service**).

Then `mymenu ~service:home sp` will generate the following code:

```
<ul class="menu menuprincipal">
  <li class="current first">Home
  </li>
  <li><a href="infos">More info</a>
  </li>
  <li class="last"><a href="tutorial">Documentation</a>
  </li>
</ul>
```

Personalise it in your CSS style-sheet.

Eliom_tools.menu takes a list of services without GET parameters. If you want one of the link to contains GET parameters, pre-apply the service.

How to make a menu entry with GET parameters?

Preapply your service.

Hierarchical menus

```
(* Hierarchical menu *)
open Eliom_tools

let hier1 = new_service ~path:["hier1"] ~get_params:unit ()
let hier2 = new_service ~path:["hier2"] ~get_params:unit ()
let hier3 = new_service ~path:["hier3"] ~get_params:unit ()
```

```
let hier4 = new_service ~path:["hier4"] ~get_params:unit ()
let hier5 = new_service ~path:["hier5"] ~get_params:unit ()
let hier6 = new_service ~path:["hier6"] ~get_params:unit ()
let hier7 = new_service ~path:["hier7"] ~get_params:unit ()
let hier8 = new_service ~path:["hier8"] ~get_params:unit ()
let hier9 = new_service ~path:["hier9"] ~get_params:unit ()
let hier10 = new_service ~path:["hier10"] ~get_params:unit ()

let mymenu =
  (
   (Main_page hier1),

   [([pcdata "page 1"], Site_tree (Main_page hier1, []));

    ([pcdata "page 2"], Site_tree (Main_page hier2, []));

    ([pcdata "submenu 4"],
     Site_tree
       (Default_page hier4,
         [([pcdata "submenu 3"],
          Site_tree
             (Not_clickable,
              [([pcdata "page 3"], Site_tree (Main_page hier3, []));
               ([pcdata "page 4"], Site_tree (Main_page hier4, []));
               ([pcdata "page 5"], Site_tree (Main_page hier5, []))]
             )
          );

          ([pcdata "page 6"], Site_tree (Main_page hier6, []))]
       )
    );

    ([pcdata "page 7"],
     Site_tree (Main_page hier7, []));

    ([pcdata "disabled"], Disabled);

    ([pcdata "submenu 8"],
     Site_tree
       (Main_page hier8,
        [([pcdata "page 9"], Site_tree (Main_page hier9, []));
         ([pcdata "page 10"], Site_tree (Main_page hier10, []))]
       )
    )
  ]
  )

let f i s sp () () =
  return
    (html
       (head (title (pcdata ""))
          ((style ~contenttype:"text/css"
             [cdata_style
 "a {color: red;}\n
  li.eliomtools_current > a {color: blue;}\n
  .breadthmenu li {\n
    display: inline;\n
    padding: 0px 1em;\n
    margin: 0px;\n
```

```
    border-right: solid 1px black;}\n
  .breadthmenu li.eliomtools_last {border: none;}\n
                "])::
                structure_links mymenu ~service:s ~sp)
              )
        (body [h1 [pcdata ("Page "^string_of_int i)];
               h2 [pcdata "Depth first, whole tree:"];
               div
                 (hierarchical_menu_depth_first
                    ~whole_tree:true mymenu ~service:s ~sp);
               h2 [pcdata "Depth first, only current submenu:"];
               div (hierarchical_menu_depth_first mymenu ~service:s ~sp);
               h2 [pcdata "Breadth first:"];
               div
                 (hierarchical_menu_breadth_first
                    ~classe:["breadthmenu"] mymenu ~service:s ~sp )]))


let _ =
  register hier1 (f 1 hier1);
  register hier2 (f 2 hier2);
  register hier3 (f 3 hier3);
  register hier4 (f 4 hier4);
  register hier5 (f 5 hier5);
  register hier6 (f 6 hier6);
  register hier7 (f 7 hier7);
  register hier8 (f 8 hier8);
  register hier9 (f 9 hier9);
  register hier10 (f 10 hier10)
```

Examples

Writing a forum

As an example, we will now write a small forum. Our forum has a main page, summarising all the messages and a page for each message. All the functions to access the database and print the result are left to the reader. We only want to show the structure of the site. Suppose you have written a function `news_headers_list_box` that writes the beginning of messages, and `message_box` that write a full message.

```
(* All the services: *)

let main_page = new_service ~path:[""]
    ~get_params:unit ()

let news_page = new_service ["msg"] (int "num") ()

(* Construction of pages *)

let home sp () () =
  page sp
    [h1 [pcdata "Mon site"];
     news_headers_list_box
       sp anonymoususer news_page]

let print_news_page sp i () =
```

```
  page sp
    [h1 [pcdata "Info"];
     message_box i anonymoususer]

(* Services registration *)

let _ = register
  ~service:main_page
  home

let _ = register
  ~service:news_page
  print_news_page
```

Now the same example with a login box on each page. We now have two versions of each page: connected and not connected. We need two actions (for connection and disconnection). Suppose we have the functions `login_box`, `connected_box`, and `connect`.

```
(* All the services: *)

let main_page = new_service ~path:[""] ~get_params:unit ()

let news_page = new_service ["msg"] (int "num") ()

let connect_action =
  new_post_coservice'
    ~post_params:(string "login" ** string "password")

(* Construction of pages *)

let home sp () () =
   match get_volatile_session_data ~table:my_table ~sp () with
   | Eliom_sessions.Data_session_expired
   | Eliom_sessions.No_data ->
     page sp
       [h1 [pcdata "My site"];
        login_box sp connect_action;
        news_headers_list_box sp anonymoususer news_page]
   | Eliom_sessions.Data user ->
      page sp
        [h1 [pcdata "Mon site"];
         text_box "Bonjour !";
         connected_box sp user disconnect_action;
         news_headers_list_box sp user news_page]

let print_news_page sp i () =
   match get_volatile_session_data ~table:my_table ~sp () with
   | Eliom_sessions.Data_session_expired
   | Eliom_sessions.No_data ->
      page sp
        [h1 [pcdata "Info"];
         login_box sp connect_action;
         message_box i anonymoususer]
   | Eliom_sessions.Data user ->
      page sp
        [h1 [pcdata "Info"];
         connected_box sp user disconnect_action;
         message_box i user]

(* Services registration *)

let _ = register
  ~service:main_page
  home

let _ = register
  ~service:news_page
  print_news_page

let launch_session sp user =
  set_volatile_session_data my_table sp user

let _ = Eliom_predefmod.Actions.register
  ~action:connect_action
    (fun h (login, password) ->
      launch_session sp (connect login password); return [])
```

Miniwiki

Ocsigen's source code contains an example of Wiki written with Eliom by Janne Hellsten. It is called *Miniwiki*.