# Introduction naïve au développement d'applications web avec Ocsigen

Xavier Van de Woestyne

Decembre 2014

Pendant très longtemps, j'ai été amené à concevoir des applications web. J'ai eu l'occasion d'expérimenter plusieurs technologies (PHP, Ruby on Rails, Django, Spring, ASP.NET et Yaws + Erlang). Je dois avouer n'avoir été que trop rarement satisfait par ces outils, principalement car depuis ma découverte des langages fonctionnels statiquement typés, j'ai réellement du mal à m'en passer. C'est pour cette raison qu'à l'annonce de Ocsigen, j'ai été véritablement emballé. Cependant, Ocsigen est, actuellement, pauvre en ressource et assez dépaysant (cette opinion n'est absolument pas à prendre comme un reproche mais plus comme une constatation et s'explique par l'innovation démontrée dans Ocsigen et sa jeunesse)), j'ai donc décidé de lui accorder un petit article proposant à sa fin, une implémentation pratique où chaque étape sera décrite !

Un des prérequis de ce cours est une connaissance du langage OCaml (version 4.01.x)

## Présentation sommaire de Ocsigen

Ocsigen est une suite de logiciels libres écrite en OCaml, développée par le laboratoire français PPS. Son objectif principal réside dans la conception d'applications web fiables au moyen du langage de programmation OCaml qui est, en plus d'être extrêmement fiable, très expressif.
Ocsigen propose des réponses à des problèmatiques récurrentes du développement web, dans certains cas assez différentes des solutions proposées par d'autres technologies plus populaires.

Un des arguments en faveur d'Ocsigen est l'usage *d'un seul langage pour tout faire*, en effet, qu'il s'agisse de l'écriture de la couche statique (le HTML, vérifiant la validité de ce dernier), du Javascript ou du SQL (vérifiant la bonne formation d'une requête), Ocsigen est dôté d'outils et d'extensions de syntaxes pour ne demander au programmeur, qu'une connaissance de Ocaml.

> Bien qu'en règle général, j'aime ne pas m'enfermer dans une forte dépendance à un outil, je dois avouer que l'excès de flexibilité de Javascript m'a quelque fois écoeuré. En écrivant du Javascript avec OCaml, on préserve un des intérêts majeur du langage (de mon point de vue), son typage fort. Le déboguage d'applications usant massivement de Javascript n'est plus un problème.

N'utilisant qu'un seul langage, Ocsigen permet de manipuler les couches *client-serveur* de manière aisée, de la même manière que Meteor.js. Il parait que l'on appelle ça *une plate-forme de développement isomorphique*, mais je ne suis pas convaincu de l'apellation.

L'organisation des fichiers et des points d'entrés d'une application réalisée avec Ocsigen est aussi très différente de ce que l'on peut habituellement voir. En effet, toute la navigation est articulée autour d'une notion de service, correspondant à une valeur OCaml à part entière et pouvant être interconnecté à d'autre services. Un service peut être caractérisé par un chemin d'accès, et des paramètres **GET** et **POST** statiquement typés. (De même qu'un service ne peut être qu'une action atomique, sans chemin particulier, nous parlerons dans ce cas de coservice). Les applications Ocsigen étant compilées, un lien ne peut pointer vers un service inexistant, une vérification complémentaire des liens cassés est donc effectué à la compilation.

A ça s'ajoute *Lwt*, une bibliothèque pour effectuer des traitements concurrent, soit une fragmentation de tâches en sous-programmes s'exécutant simultanéments (où chaque processus décide lui même de céder la main à son successeur). Cette bibliothèque apporte un confort indéniable dans la construction de tâches asyncrones et son portage pour **Js_of_OCaml** (le Javascript statiquement typé de Ocsigen) est admirablement pertinent dans le cadre d'exécution de Javascript.

A ces axes brievement présenté s'ajoute une collection d'outil pour résoudre des problèmatiques classiques du développement web, que nous tâcherons de survoler dans la partie pratique de cet article.

### Pourquoi utiliser Ocsigen

Si j'ai décidé de m'intéresser à Ocsigen, c'est avant tout par intérêt pour le langage OCaml. Cependant, après usage, il est évident que, comme pour le développement d'application classique, le typage statique est véritablement un atout indéniable. On évite une quantité de tests chronophage, et beaucoup de vérifications "plus bas niveau" sont effectuées par Ocsigen. Le développeur n'a donc plus à s'en soucier. De plus, le fait, par exemple, que le compilateur vérifie la bonne sémantique du HTML accélère le flot de travail (n'obligeant plus à vérifier chaque fois la validité de son code HTML).

De mon point de vue, Ocsigen se pose de bonnes questions face à des problèmatiques récurrente. Et même si je ne pense pas qu'il devienne (même si je l'aimerais) un incontournable absolu, au même titre que PHP ou Ruby On Rails, je pense qu'il met en lumière certaines bonnes pratiques, qui seront sûrement adoptées par d'autres outils, potentiellement plus démocratiques. (On peut noter une certaines similitude avec le projet OPA, par exemple).

#### Des utilisateurs d'Ocsigen

Ocsigen n'est actuellement pas un des mastodonte de la programmation web, cependant, il possède tout de même une liste d'utilisateurs. En voici quelques'uns choisis à la volée :

- Facebook : Utilisation de Js_of_OCaml (pour leur développement interne).
- Cumulus : Un petit outil de partage de liens (dont le code source m'aura été très utile pour mon apprentissage).
- BeSport : Ils viendraient de recruter Vincent Balat, le chef du projet Ocsigen.
- Prumgrana : Un projet qui semble gagner tous les concours où il s'inscrit.

Quoi qu'il en soit, je n'ai pas eu l'occasion de lire de témoignage désaprobateur sur Ocsigen, sauf peut être … le mien, après avoir tenté un Hackathon sans une préparation suffisante… mais sachez que j'ai revu mon opinion sur le projet après avoir pris le temps de me plonger dedans.

## Implémentation d'une plateforme de micro-blogging

Pour présenter l'implémentation concrète d'une application avec Ocsigen, j'ai choisi un exemple bien peu original. Cependant, je pense qu'il est intéressant car il permet de présenter plusieurs aspects du framework (création d'un **CRUD**, donc usage d'une base de données et de Macaque, utilisation de **ocaml-safepass** pour crypter les mots de passes et éventuellement l'exposition et l'usage de **webservices**).

> Avant de me lancer dans l'explication détaillée, je tiens à préciser que cet article est aussi une manière d'éprouver Ocsigen, pour moi, dans un contexte très pratique. Je ne suis pas du tout un expert et il est possible que certains choix structurels soient discutables. Si vous avez des choses à redire n'hésitez pas à vous servir des commentaires, je suis ouvert à toute critique !

Par soucis de lisibilité (et car ça ne présente pas beaucoup d'intérêt), le CSS sera mis de côté. On évoquera comment utiliser sa propre feuille de style, mais je ne parlerai pas du code CSS quand il n'aura pas de rapport direct avec l'implémentation de l'application.

### Installation

L'installation d'Ocsigen est véritablement simple depuis la mise en place de OPAM. Après l'avoir installé (et initialisé), l'installation de Ocsigen peut se limiter à :

```
opam install eliom
```

Pour ma part, j'utilise la version `4.01.0` de OCaml et l'installation (un peu longue) s'est déroulée sans soucis. L'usage d'OPAM est un véritable plus car c'est lui qui nous permettra d'installer les paquets additionnels nécéssaire à notre application. (On pourrait faire une analogie avec les *gems* de Ruby par exemple).
Sans rentrer dans les détails de l'anatomie du paquet, voici quelques petits compléments explicatifs sur la constitution du paquet `eliom` :

- `Js_of_ocaml` : Pour produire du Javascript bien typé (et en ocaml)
- `Macaque` et `PgOcaml` : Pour la construction de requêtes en OCaml
- `Deriving` : Permet de dériver un type dans une représentation (en JSon par exemple)
- `Lwt` : Bibliothèque pour les traitements concurrents
- `` `Camlp4 `` : Ocsigen offre une collection d'extension de syntaxes
- `` `TyXML `` : Une bibliothèque de génération de XML statiquement typé

Le paquet est aussi évidemment composé d'un serveur web, d'outils pour la gestion des calendriers/dates et de beaucoup d'autres outils. Comme vous pouvez le voir, Ocsigen est riche en composants.

### Raisonnement général de l'application

Avant de se lancer dans l'écriture de code OCaml, nous allons penser conceptuellement notre application, proposant des diagrammes minimaliste (et la structure de la base de données).

Un utilisateur non connecté peut :

- Lire les messages publiés par les titulaires d'un compte
- Se connecter
- S'enregistrer

Un utilisateur connecté peut :

- Lire les messages publiés par les titulaires d'un compte
- Se déconnecter
- Créer un message
- Afficher ses messages
- Modifier ses messages
- Modifier les paramètres de son compte

Les constituants de cette application sont donc les **messages** et les **utilisateurs**. Un message sera caractérisé par un **identifiant unique**, une **date de publication**, un **utilisateur posteur** et un **contenu**. Alors qu'un utilisateur sera lui caractérisé par un **identifiant unique**, un **pseudonyme**, une **adresse courriel** et un **mot de passe**.

Voici un diagramme réellement minimaliste de notre base de de données. Comme il l'indique, la relation est effectuée au niveau de l'utilisateur, qui permet de relier les messages publiés par un utlisateur. (Rien de bien compliqué).

| users | ← messages |
|---|---|
| `user_id` **PK** | `message_id` **PK** |
| `nickname` | `publication_date` |
| `email` | `user_id` **FK** |
| `password` | `content` |

Et le code SQL (j'utilise **Postgres**) pour générer la base de données est ici, il n'est pas expliqué de manière méticuleuse car je suppose qu'il ne présente absolument rien de nouveau. Cependant, comme pour la structure choisie pour ce tutoriel, je suis ouvert à toute critique dans les commentaires !

```
-- Création des séquences pour l'auto increment
CREATE SEQUENCE users_id_seq;
CREATE SEQUENCE messages_id_seq;

-- Création de la table d'utilisateurs
CREATE TABLE users (
  user_id integer PRIMARY KEY,
  nickname text NOT NULL,
  email text NOT NULL,
  password text NOT NULL
);
```

```
-- Création de la table de messages
CREATE TABLE messages (
  message_id integer PRIMARY KEY,
  publication_date timestamp NOT NULL,
  user_id integer REFERENCES users (user_id)
      ON DELETE CASCADE,
  content text NOT NULL
);
```

Rien de bien compliqué, mais c'est amplement suffisant pour notre exemple. Comme on peut le voir, la table message est reliée par le champs `user_id` et la suppression d'un utilisateur supprimera tous ses messages. A partir de ce stade de l'article, j'estimerai que la base de données à été initialisée et comprend ces deux tables.

### Structure et construction d'un premier projet

Une manière commune d'initialiser un projet Ocsigen (une fois ce dernier installé) est d'utiliser l'outil `eliom-distillery` qui crée un projet vierge et les éléments nécéssaire au déploiement d'une application.

> L'outil `eliom-distillery` est très utile pour la construction d'application web. Cependant, certains lui reprocheront l'impossibilité de construire des bibliothèques. Pour ça, je vous conseillerai `OASIS`. Cependant ce n'est pas la thématique de cet article.

L'usage de `eliom-distillery` est généralement :

```
eliom-distillery -name nom_du_projet
```

Qui vous proposera la création d'un répertoire du nom du projet choisi. Ce répertoire contient les fichiers minimums nécéssaires au lancement d'une application minimaliste Ocsigen.
Observons maintenant le contenu de ce répertoire:

- `votre_projet.eliom` : premier fichier de votre application
- `votre_projet.conf.in` : le fichier template de configuration. Vous n'avez à priori jamais à y toucher
- `Makefile` : le Makefile de lancement/déploiement de l'application. Vous n'avez à priori jamais à le modifier non plus

- `Makefile.options` : c'est le fichier utilisé pour générer le `Makefile` et la configuration du serveur. Lui peut être modifié, notamment pour ajouter des extensions externes, ou encore modifier le port du serveur de test (ou d'exécution). Il n'est au final qu'une simple liste de variables à modifier ou enrichir
- `static/` : le répertoire contenant les *assets* du projets (CSS, images, Javascript normal)
- `README` : ce fichier donne plus ou moins les explications que je donne dans cette rubrique (mais en Anglais!).

Le fichier `.eliom` est le premier fichier de sources. Et chaques fichiers `.eliom` (ou `.eliomi` pour les interfaces) est considéré automatiquement comme constituant de l'application. (Il est possible d'étendre ce statut aux fichiers `.ml` et `.mli` en enrichissant les variables `SERVER_FILES` et `CLIENT_FILES`).
Voyons le code de ce fichier généré avec la commande `eliom-distillery -name microblog` :

```
{shared{
  open Eliom_lib
  open Eliom_content
  open Html5.D
}}

module Microblog_app =
  Eliom_registration.App (
    struct
      let application_name = "microblog"
    end)

let main_service =
  Eliom_service.App.service ~path:[] ~get_params:Eliom_parameter.unit ()

let () =
  Microblog_app.register
    ~service:main_service
    (fun () () ->
      Lwt.return
        (Eliom_tools.F.html
           ~title:"microblog"
           ~css:[["css";"microblog.css"]]
           Html5.F.(body [
             h2 [pcdata "Welcome from Eliom's distillery!"];
           ])))
```

Rassurez-vous, nous étudierons son anatomie plus tard. Retenons juste que par défaut, une application générée avec `eliom-distillery` ne propose qu'un seul

service sur la racine du serveur.

### Tester son application fraîchement générée

Comme l'indique le `README` (pour ceux l'ayant lu), on peut tester localement son projet, pour cela il suffit de lancer la directive :

```
make test.byte
```

Qui compilera les fichiers relatifs à notre application et exécutera le serveur sur le port de test (défini dans `Makefile.options`). La page est donc accessible à http://localhost:8080 (si vous n'avez pas modifié le port de test par défaut). Si vous n'avez pas modifié votre fichier `.eliom`, une fois le serveur de test lancé, la page devrait fièrement afficher : *"Welcome from Eliom's distillery!"*.

### Anatomie du .eliom

Le fichier `eliom` généré peut être assez déroutant pour le profane, nous allons donc rapidement survoler ses constituants, non pas pour les détailler méticuleusement, mais pour proposer un survol rapide de ce qui constitue une application Ocsigen.

```
{shared{
  open Eliom_lib
  open Eliom_content
  open Html5.D
}}
```

Cette partie ne concerne que les "importations". Comme vous pouvez le voir, les importations sont entourées par `{shared{` et `}}`. Cela indique que les importations sont partagées au client et au serveur.
Concrètement, Ocsigen propose un traitement uniforme du client et du serveur, il est donc possible de ne contrôler que du code client ou que du code serveur, ou les deux.

- `{client{du_code}}` : exécuter `du_code` uniquement côté client
- `{server{du_code}}` : exécuter `du_code` uniquement côté serveur
- `{shared{du_code}}` : exécutera `du_code` des deux côtés.

Sans mise en contexte, le code est toujours exécuté serveur, c'est pour cette raison que l'on verra rarement de code entre `{server{ ... }}`.

```
module Microblog_app =
  Eliom_registration.App (
    struct
      let application_name = "microblog"
    end)
```

Cette partie génère un module (`NomApplication_app`) lui conférant (c'est un module généré par un autre, `Eliom_registration.App` est un foncteur ne requierant qu'une fonction `application_name`). C'est au travers de ce module que nous enregistrerons nos services. C'est un peu le point d'entrée de l'application.

```
let main_service =
  Eliom_service.App.service
    ~path:[]
    ~get_params:Eliom_parameter.unit
    ()
```

Cette partie consiste à définir un service, en l'occurence, le service "par défaut". Ce dernier n'a pas de chemin (donc il est accessible via l'url : `http://monsite` et n'attend aucun paramètre.

Nous préciserons la notion de service plus tard, cependant, c'est au travers de ce mécanisme qu'Ocsigen permet de faire abstraction des fichiers physiques accessibles par l'url, comme c'est le cas en PHP (par exemple).

```
let () =
  Microblog_app.register
    ~service:main_service
    (fun () () ->
      Lwt.return
        (Eliom_tools.F.html
           ~title:"microblog"
           ~css:[["css";"microblog.css"]]
           Html5.F.(body [
             h2 [pcdata "Welcome from Eliom's distillery!"];
           ])))
```

Une fois créé, c'est au lancement de l'application que nous enregistrerons notre service et que nous définirons la page que dois renvoyer le service. Le code ci-dessus exprime que l'on enregistre le service `main_service`. Et que l'on spécifie que ce service, qui ne reçoit aucun argument GET et POST (les deux units de la fonction anonyme donnée en argument) renverra une page HTML ne contenant, en plus des constituants classique d'une page, un message en `<h2>` : `"Welcome from Eliom's distillery!"`.

Concrètement, un service est un élément (réutilisable) créé et caractérisé par un chemin d'accès et des paramètres. Une fois qu'il est défini, il est enregistré et correspond à un point d'entrée de l'application. Renvoyant, dans beaucoup de cas, une page HTML (conçue, ici, via TyXML, mais il existe d'autres manières).

Maintenant que nous avons étudié la structure d'une page `.eliom` générée par `eliom-distillery`. Nous allons nous familiariser avec les pages et les services au moyen de petits exercices pratiques.

### Le Hello World

Dans cette partie, nous allons tâcher de faire l'habituel **Hello World**, mais avec quelques variantes. En trichant, il serait très facile de ne faire que remplacer `h2 [pcdata "Welcome from Eliom's distillery!"];` par `h2 [pcdata "Hello World"];`, mais ce serait, non seulement tricher, et totalement inutile. Cette fois nous allons directement proposer une structure de fichiers et réaliser notre Hello World sur plusieurs modules. Je vous invite à construire, via `eliom-distillery` un nouveau projet baptisé `hello`.

**Proposition d'organisation** Comme je l'ai évoqué dans l'introduction, cette structure est issue de mon propre raisonnement (mais tout de même *fortemment inspirée$ de ce que j'ai pu observer sur Github, pour les quelques projets, principalement Cumulus, que j'ai trouvé). Donc si jamais des gurus de Ocsigen ont des commentaires à faire, qu'ils n'hésitent surtout pas ! Cet article est aussi un prétexte à ma progression !

Donc pour éviter d'avoir des fichiers trop longs, dans le contexte de cette application Hello World, je propose d'organiser le tout de cette manière :

- `services.eliom` : Le module où nous définirons tous nos services.
- `gui.eliom` : (Je ne suis pas convaincu par son nom) Le module qui offrira les fonctions pour la génération de page, pour éviter de répéter le code de structure d'une page.
- `pages.eliom` : Le module qui décrira formellement chaques pages de l'application.
- `hello.eliom` : Le point d'entrée de l'application, soit le fichier qui enregistrera les services, et qui générera le foncteur de l'application.

**Elaguons l'inutile** Comme nous allons reprendre notre application depuis le début, je propose de supprimer le superflu (que nous réécrirons dans les bons fichiers). Notre `hello.eliom` pourra se limiter à :

```
(* Directives d'importation *)
{shared{
```

```
  open Eliom_lib
  open Eliom_content
  open Html5.D
}}


(* Création du module principal pour  *)
(* enregistrer les services, notamment *)
module Hello_app =
  Eliom_registration.App (
    struct
      let application_name = "hello"
    end)
```

En compilant via `make test.byte`, tout marchera sauf qu'en se rendant à l'adresse du serveur (ici, j'ai modifié, dans le `Makefile.options` le port de test car `8080` est déjà utilisé par `Yaws`, un serveur Erlang) on arrive sur une erreur 404. Ce qui est logique car aucun service n'a été défini.

**Une page type** Avant de concevoir les services, et s'intéresser aux paramètres, nous allons généraliser le comportement d'une page type. Pour cela rendons nous dans le fichier `gui.eliom`.

Ce que je propose, c'est d'importer les modules dont nous aurons besoin pour construire le squelette d'une page, ensuite de réaliser trois quatre fonctions.

- Une fonction pour créer une page vide
- Une fonction pour le header
- Une fonction pour le footer
- Une fonction pour créer une page avec le header et le footer.

Ce qui nous permettra de créer rapidement une page via une simple fonction, ne prenant en argument que la liste de balises nécéssaire à la création du contenu que nous désirons pour une page:

```
(* Les importations ne doivent être, cette fois  *)
(* effectives uniquement côté serveur, donc pas la peine  *)
(* d'utiliser {{shared{{}} *)

(* contient les modules de création de contenu (Html5)*)
open Eliom_content
(* Contient le HTML avec la sémantique du DOM, pour les
   application hybrides
 *)
```

```
open Html5.D


(* Squelette d'une page *)
let skeleton title body_content =
  Eliom_tools.F.html
    ~title:title
    ~css:[["css"; "hello.css"]]
    (Html5.F.body body_content)

(* Header et footer *)
let header =
  [div
     [
       h1 [pcdata "Application Hello !"];
       hr ();
     ]
  ]
let footer =
  [div
    [
      hr ();
      span [pcdata "Fin de l'application"]
    ]
  ]

(* Page type *)
let std_page body_content =
  skeleton
    "Hello World application"
    (header @ body_content @ footer)
```

Concrètement, on reproduit, mais de manière plus fragmentée le code fournit par `eliom-distillery`. La nuance entre `Html5.F` et `Html5.D` est assez sensible mais concrètement, retenons juste que les éléments construits par `D` permettent d'être *client-server* et manipulé dans les deux côtés. A priori, la structure de notre page (`<html><head>...</head><body>`) ne devra jamais être manipulée par du Javascript, on peut la créer via le module `F`. Le contenu de la page pouvant être potentiellement manipulé, on utilisera `D`.
Le manuel de `Eliom` conseille, pour les débutants de n'utiliser que `D`. Cependant, je me base sur le code fourni par `eliom-distillery` pour le squelette de ma page, donc j'utilise `F`.

**Construction de HTML** Dans la création du module `gui.eliom`, nous avons créé du Html via `TyXML`. Voyons en détail comment cela fonctionne. Typ-

iquement les balises sont des fonctions ayant le même nom que leur représentation HTML. De ce fait :

```
div [pcdata "Hello"]
```

produira le code HTML suivant:

```
<div>Hello</div>
```

TyXML utilise des types fantômes pour représenter les balises, vous pouvez vous référer à cet article qui évoque sommairement le concept sous-jacent. et le `pcdata` sert à définir du "texte brute". C'est typiquement le texte racine de chaque balise pouvant afficher du texte.
Voyons un exemple un peu plus complexe et essayons de générer ce code via TyXML :

```
<div>
  <h1>Bonjour !</h1>
  <hr />
  <div>
    <p>Voici la liste de mes envies :</p>
    <ul>
      <li>Une glace</li>
      <li>Le livre Real World OCaml</li>
      <li>Une chemise blanche</li>
    </ul>
  </div>
</div>
```

Voici sa représentation TyXML :

```
div
[
  h1 [pcdata "Bonjour !"];
  hr ();
  div
  [
    p [pcdata "Voici la liste de mes envies :"];
    ul
    [
      li [pcdata "Une glace"];
      li [pcdata "Le livre Real World OCaml"];
      li [pcdata "Une chemise blanche"];
    ]
  ]
]
```

Les retours à la ligne ne sont absolument pas obligatoire, ce que l'on retient, c'est qu'une balise pouvant en contenir d'autres a cette sémantique : `balise [liste d'autres balises]` et qu'une balise "unique", de cette forme `<balise />` (soit `<hr />` ou `<br />` par exemple) respecte cette sémantique : `balise ()`.

Un des avantage de passer par TyXML est de garantir que le HTML produit est correctement typé (selon le W3C) (essayons donc de mettre une `<div>` dans un `<span>` et le code ne compilera même pas !), mais aussi d'être généralement plus court à écrire.

Il est évidemment possible d'enrichir les balises d'attributs, pour ça on utilise l'argument labelisé `~a`, qui attend une liste d'attributs. Par exemple une liste de classes, ou un identifiant :

```
div
  ~a:[a_id "identifiant"; a_class ["rouge; vert"]]
  [
     span ~a:[a_id "say_hello"] [pcdata "Hello !"]
  ]
```

Donnera le code HTML suivant :

```
<div id="identifiant" class="rouge vert">
  <span id="say_hello">Hello !</span>
</div>
```

Maintenant, la compréhension de `gui.eliom` devrait être plus évidente et nous pouvons même modifier ce dernier pour que le contenu donné à la fonction `std_page` soit placé dans une div à l'identifiant `content` et que le header et le footer soient aussi identifié !

```
(* Les importations ne doivent être, cette fois  *)
(* effectives uniquement côté serveur, donc pas la peine  *)
(* d'utiliser {{shared{{}} *)

(* contient les modules de création de contenu (Html5)*)
open Eliom_content
(* Contient le HTML avec la sémantique du DOM, pour les
   application hybrides
 *)
open Html5.D


(* Squelette d'une page *)
let skeleton title body_content =
```

```
  Eliom_tools.F.html
    ~title:title
    ~css:[["css"; "hello.css"]]
    (Html5.F.body body_content)

(* Header et footer *)
let header =
  div
    ~a:[a_id "header"]
    [
      h1 [pcdata "Application Hello !"];
      hr ();
    ]

let footer =
  div
    ~a:[a_id "footer"]
    [
      hr ();
      span [pcdata "Fin de l'application"]
    ]

(* Page type *)
let std_page body_content =
  skeleton
    "Hello World application"
    [
      header;
      div ~a:[a_id "content"] body_content;
      footer
    ]
```

Notre code est plus lisible, et au moyen de CSS, on peut accéder à priori à chaque élément constituants de notre page car le header, le footer et le contenu sont identifiés !

**Le premier service** Nous allons pouvoir attaquer notre premier service, son rôle sera que dès que nous nous rendrons sur la page d'accueil de notre application, on affichera, dans une page type, le texte "Bienvenue le monde". C'est un service qui ne prendra aucun argument. Il permettra de nous familiariser avec le conception de services.
Rendons nous dans le module `services.eliom` et créons notre premier service :

```
(* Directives d'importation *)

(* Raccourci vers le module qui permet de définir
 des services en tout genre *)
open Eliom_service

(* Raccourci vers le module qui expose les types
   d'entrée du service *)
open Eliom_parameter

(* Définition du premier service *)
let main =
  App.service
    ~path:[]
    ~get_params:unit
    ()
```

Typiquement, après les directives d'importation, pour rendre le code moins verbeux, on défini dans la variable `main` un `App service`, soit un service attaché à l'url racine. Si on avait mit comme path : `~path:["a";"b"]`, l'accès au service aurait été `monsite.com/a/b`. Cependant ici, nous avons défini le service mère, donc accessible via la racine de l'application. Et on a défini qu'il ne prenait aucun argument `GET`.

Attention, des services peuvent pointer vers le même chemin pour peu que leur paramètres (GET ou POST) soient différents.

> Le service défini est un App.service, soit un service "Application", il existe plusieurs types de services différents et nous tâcherons d'en survoler plusieurs durant cette initiation. Pour le moment, nous nous limitons à un App.service.

Maintenant que notre service est préparé, nous allons pouvoir créer la fonction que le service renverra (lorsque nous l'enregistrerons). Pour ça nous allons créer le module `pages.eliom` dans lequel nous créerons une fonction `main` dont le rôle sera de renvoyer la page correspondante au service.

Une fonction de renvoi de service à deux arguments, correspondants respectivements aux paramètres **GET** et **POST**. Notre service `main` n'a pas de paramètres, il prendra donc deux `unit`.
Il est important de retenir que chaque page doit être mise dans un **contexte Lwt** donc chaque page doit être renvoyée au moyen de la fonction `Lwt.return`. Pour ça, nous allons modifier légèrement notre fonction `std_page` pour ne plus avoir à s'en occuper.

```
(* Page type *)
let std_page body_content =
```

```
  Lwt.return (
      skeleton
       "Hello World application"
       [
         header;
         div ~a:[a_id "content"] body_content;
         footer
       ]
    )
```

Nous pouvons maintenant nous occuper de `Pages.main` qui sera très simple car son rôle est simplement d'afficher "Bienvenue le monde !" :

```
(* contient les modules de création de contenu (Html5)*)
open Eliom_content
(* Contient le HTML avec la sémantique du DOM, pour les
   application hybrides
 *)
open Html5.D
open Gui

(* Service principal *)
let main () () =
  std_page
    [
      h2 [pcdata "Bienvenue le monde !"];
    ]
```

Comme vous pouvez le voir, nous avons importé les mêmes modules que dans le module `gui.eliom` pour pouvoir écrire confortablement du HTML et nous avons ajouté l'importation du module `gui.eliom` pour pouvoir appeler directement la fonction `std_page` (mais c'est vraiment de la coquetterie car j'aurais pu simplement préfixer ma fonction de cette manière : `Gui.std_page`).

Maintenant que nous avons tous les ingrédients requis, nous pouvons **enregistrer** notre service pour le rendre accessible. Les personnes qui auraient tenté de compiler le projet auraient eu droit à une belle erreur indiquant que le service n'est pas enregistré. Rendons nous dans le module `hello.eliom` et procédons à l'enregistrement du module via une fonction exportée par le module `Hello_app` nommée `register` et dont la sémantique est :

```
Hello_app.register
  ~service:variable_du_service
  fonction de renvoi
```

Donc nous pouvons enrichir notre module `hello.eliom` de l'ajout du service (situé dans les dernières lignes).

```
(* Directives d'importation *)
{shared{
  open Eliom_lib
  open Eliom_content
  open Html5.D
}}


(* Création du module principal pour  *)
(* enregistrer les services, notamment *)
module Hello_app =
  Eliom_registration.App (
    struct
      let application_name = "hello"
    end)

(* Enregistrement du service main *)
let _ =
  Hello_app.register
    ~service:Services.main
    Pages.main
```

Maintenant nous pouvons compiler notre projet et nous rendre à l'adresse du serveur local, notre "Bienvenue le monde !" devrait normalement être affiché dans le gabarit que nous avions établit au moyen de `std_page`.

Il est évident que pour n'effectuer qu'un simple "Hello World", il aurait été sans doute plus facile de ne modifier que le contenu du fichier généré par `eliom-distillery`, cependant, maintenant, nous bénéficions d'une architecture saine pour concevoir rapidement d'autres services. Ajoutons par exemple un service `bonjourNuki` qui affichera "Bonjour Nuki !" sur l'url `monsite.com/bonjour/nuki`:

Commençons par créer le service dans `services.eliom` en y ajoutant ceci :

```
let bonjour_nuki =
  App.service
    ~path:["bonjour";"Nuki"]
    ~get_params:unit
    ()
```

Ensuite créons sa page dans `pages.eliom` en lui ajoutant ceci :

```
(* Service Bonjour Nuki *)
let bonjour_nuki () () =
  std_page
    [
      h2 [pcdata "Bonjour Nuki"];
    ]
```

Et terminons par enregistrer le service de `hello.eliom` en y ajoutant ceci :

```
(* Enregistrement du service Bonjour Nuki *)
let _ =
  Hello_app.register
    ~service:Services.bonjour_nuki
    Pages.bonjour_nuki
```

Après le lancement de la commande `make test.byte`, on peut se rendre à l'adresse (sur le port 8000 dans mon cas, car j'ai modifié mon Makefile.options) : http://localhost:8000/bonjour/Nuki qui affichera "Bonjour Nuki"

**Utilisation des paramètres GET** Notre service `bonjourNuki` est très mignon, mais franchement utile, nous aimerions un peu plus de généricité en proposant un service capable, via l'url, de désigner la personne à saluer. On peut le faire via les paramètres. Nous allons nous servir des variables **GET** pour passer un prénom en argument et le saluer. Pour cela, je propose un nouveau service `bonjour` qui aura une forme un peu différente du précédent. Premièrement, supprimons tout ce qui est relatif à notre service `bonjourNuki` et lançons nous (dans `services.eliom`) :

```
(* Un service pour saluer *)
let bonjour =
  App.service
    ~path:[:"bonjour"]
    ~get_params:(string "prenom")
    ()
```

Au contraire des deux précédents services, nous lui adjoignont un paramètres **GET** de type string et dont le nom est `prenom`. L'existence de ce paramètre implique donc que le premier argument de la fonction qui retourne la page doit être une chaîne de caractère :

```
(* Service pour dire bonjour *)
let bonjour prenom () =
  std_page
```

```
    [
      h2 [pcdata ("Bonjour " ^ prenom)]
    ]
```

Par contre, l'enregistrement du service ne change absolument pas (et c'est un des intérêt principale de notre découpe modulaire) :

```
(* Enregistrement du services Bonjour *)
let _ =
  Hello_app.register
    ~service:Services.bonjour
    Pages.bonjour
```

Concrètement, nous avons précisé qu'un service `bonjour` attend un paramètre (passé via l'url) nommé `prénom`. La fonction page prend ce paramètre en argument et l'affiche. En compilant notre projet, le service principal est toujours accessible et si nous nous rendons à l'url http://localhost:8000/bonjour?prenom=Nuki, notre application affiche bien "Bonjour Nuki". Vous pouvez remplacer le prénom par n'importe quelle chaine de caractères et notre superbe application saluera bien le prénom passé en argument.

Si nous tention d'accéder au service `bonjour` sans lui passer d'argument, ou en lui en passant trop, ou en ommettant le paramètre prénom, l'application affichera une erreur `Wront parameters arguments`.
En effet, les services sont typés et il n'est pas possible d'accéder à une page au travers d'un chemin qui n'a pas été pensé. C'est un gain de temps monumental en vérification. Personnellement, je me souviens de PHP ou je devais vérifier chacune de mes variables pour garantir le bon déroulement de la page. Cette vérification s'ajoute à l'impossibilité de produire du HTML non valide comme un gain de temps réel dans l'écriture d'application web!

### A propos de la sémantique des paramètres

> Même si nous n'avons pas encore survolé les paramètres POST, cette précision est valable pour eux aussi.

Actuellement, nous n'avons travaillé qu'avec un seul paramètre, ça se prêtait bien car la fonction de renvoi ne peut prendre que deux paramètres, un pour les arguments **GET** (le premier) et un pour les arguments **POST** (le second) (ce qui introduit implicitement déjà le fait que l'on peut avoir des services prenant des arguments **POST** ET **GET**).

Concrètement il est évidemment possible d'utiliser de multiples arguments, par exemple, pour créer un service demandant un prénom, un nom, un age (par exemple), on pourrait imaginer ce service :

Définition du service dans le module `services.eliom`:

```
let infos =
  App.service
    ~path:["informer"]
    ~get_params:(
      string "prenom"
      ** string "nom"
      ** int "age"
    ) ()
```

On se sert de l'opérateur `**` pour construire une valeur de services multiples. Du côté de la fonction de renvoi, le paramètre GET sera représenté par *n-uplet* ayant une forme similaire à une liste :

```
(* Service de renvoi d'information *)
let infos (prenom, (nom, age)) () =
  std_page
  [
    ul
    [
      li [
          strong [pcdata "Prenom : "];
          pcdata prenom
        ];
      li [
          strong [pcdata "Nom : "];
          pcdata nom
        ];
      li [
          strong [pcdata "Age : "];
          pcdata (string_of_int age)
        ];
      ]
    ]
```

Le type des paramètres de la fonction seront `(string * (string * int)) -> unit`. La raison qui explique cette forme est que les services sont typés et une manière de représenter des données conjointes est l'utilisation de *n-uplet*.

Comme pour les autres services, l'enregistrement est assez simple (et sémantiquement identique aux autres) :

```
(* Enregistrement du services infos *)
let _ =
  Hello_app.register
    ~service:Services.infos
    Pages.infos
```

Après compilation, je vous invite à tester http://localhost:8000/informer?prenom=Xavier&age=25&nom=VDW qui donne des informations sur ma personne. Comme vous pouvez le voir, les arguments doivent avoir le bon type (mettre `chevre` en valeur au paramètre `age` renverra une erreur), par contre, ils ne doivent pas spécialement être dans le bon ordre.

**Pour plus d'esthétique dans les URL's** Il est courant de ne plus voir des url's chargé de paramètres **GET**. Par exemple la page `http://www.monsite.com?page=News&id=45` devient souvent (pour plaire aux moteurs de recherche) `http://www.monsite.com/News/45`. On appelle ça de la réécriture d'URL ou de l'URL rewritting. Typiquement il s'agit de paramétrer une route vers une page. Ocsigen dispose d'un mécanisme permettant de rendre les chemins plus agréable à la lecture, au moyen de la fonction `suffix` sur les paramètres. Réécrivons notre service `bonjour` pour qu'au lieu de cette adresse `http://localhost:8000/bonjour?prenom=Nuki` on utilise `http://localhost:8000/bonjour/Nuki`. Il suffit de modifier la définition de notre service de cette manière :

```
(* Un service pour saluer *)
let bonjour =
  App.service
    ~path:["bonjour"]
    ~get_params:(suffix (string "prenom"))
    ()
```

L'usage de la fonction `suffix`, présente dans le module `Eliom_parameter` permet cette représentation. D'ailleurs il existe d'autre fonctionnalité, comme par exemple celle de rendre certains paramètres optionnels. Pour plus de précisions, je vous invite à vous rendre sur la page du manuel correspondante.

**Création de liens interne à l'application** Un dernier point concernant les paramètres **GET** ou plus généralement en rapport avec les relations entre les services est la construction de liens. Contrairement à des outils plus classiques, Ocsigen crée un lien interne à l'application en créant un lien vers un service. L'avantage de cette méthode est qu'elle permet de prévenir les liens cassés. On ne peut pointer vers un service qui n'existe pas ou encore vers un service en lui donnant les mauvais argument. Pour ça on se sert de la fonction `Html.D.a`.

Nous allons ajouter sur notre service principal un lien pour dire bonjour à un certain `Nuki`, pour ça, il suffit de modifier la fonction `Pages.main` :

```
(* Service principal *)
let main () () =
  std_page
```

```
[
  h2 [pcdata "Bienvenue le monde !"];
  a
    ~service:Services.bonjour
    [pcdata "Dire bonjour à Nuki"]
    "Nuki"
]
```

Le fonctionnement de la fonction `a` est assez évident. On spécifie le service vers lequel le lien doit pointer. Le texte à afficher et le dernier argument sera le ou les paramètres **GET**. (Dans le cas de paramètres multiples, on utilisera la forme `(parametre1, (parametre2, parametre3))` par exemple).

> A partir de maintenant, je ne rappellerai plus explicitement les fichiers dans lesquels ajouter les portions de codes que je montre car je pense que vous devriez être familier avec la structure pour laquelle j'ai optée. On retiendra que la définition formelle d'un service se fera dans services.eliom, l'enregistrement des services dans hello.eliom, la construction des pages dans pages.eliom et les assets de l'interface dans gui.eliom.

**Des formulaires** Maintenant que nous sommes familier avec la construction de services avec des paramètres **GET**, nous pouvons attaquer la construction de services avec des paramètres **POST**.
La première chose à faire est de construire un service prenant en argument des paramètres **POST**. Pour ça il faudra enregistrer un autre type de service :

```
let post_bonjour =
  App.post_service
    ~fallback:main
    ~post_params:(string "prenom")
    ()
```

Au contraire d'un service *normal*, un service **POST** ne possède pas de ~**path** et n'est donc pas *bookmarkable*. En effet, on n'y accède pas via une adresse mais via un formulaire. L'argument ~**fallback** permet de rédiriger le service vers un autre service en cas d'erreur (de mauvais typage par exemple). Dans notre cas, on fera pointer le service vers le service par défaut.

Nous pouvons implémenter la page, elle sera fortement semblable à la page du service `bonjour` sauf que l'argument sera au POST et non au GET :

```
(* Service pour dire bonjour via un post*)
let post_bonjour () prenom =
```

```
  std_page
    [
      h2 [pcdata ("Bonjour " ^ prenom)]
    ]
```

Comme pour les services habituels, il nous faut l'enregistrer pour pouvoir l'utiliser. Une fois plus, il y a bien peu de choses qui changent:

```
(* Enregistrement du service post_bonjour *)
let _ =
  Hello_app.register
    ~service:Services.post_bonjour
    Pages.post_bonjour
```

Maintenant il faut construire un formulaire. Il serait possible de les créer manuellement, mais Ocsigen a prévu (une fois de plus) une manière d'automatiser la construction de formulaires. Pour créer un formulaire, on utilise la fonction `Html5.D.post_form`. Elle prend en argument un service (vers laquelle le formulaire doit pointer) et une fonction chargée de construire les formulaires. Par exemple, modifions le code de notre service `main` pour afficher un formulaire pointant vers notre services fraîchement créer :

```
(* Service principal *)
let main () () =
  let form prenom =
    [
      string_input
    ~name:prenom
    ~input_type:`Text
    ();
      string_input
    ~input_type:`Submit
    ();
    ]
  in
  std_page
    [
      h2 [pcdata "Bienvenue le monde !"];
      a
    ~service:Services.bonjour
    [pcdata "Dire bonjour à Nuki"]
    "Nuki";
      br ();
      post_form
    ~service:Services.post_bonjour
```

```
  form ()
  ]
```

Concrètement, les deux ajouts importants sont la fonction `form`, qui servira a générer le formulaire. Il s'agit d'une fonction prenant en argument le paramètre **POST** (en respectant la forme expliquée plus haut : `(a, (b, c))`. Cette fonction renvoie une liste de balise générée avec TyXML. On se sert des fonctions `string_input` (et de l'attribut ~**name** pour le champ de texte servant a récupérer un paramètre qui lui prendra comme nom un des argument du formulaire).

Ensuite, on se sert de la fonction `post_form` pour créer le formulaire en lui donnant le service pointé, le formulaire et `unit`. Vous pouvez compiler votre application et vous verrez que la page d'accueil est munie d'un petit formulaire fonctionnel, pointant vers le service `post_bonjour`.

**D'autres types de services** Dans la construction de cette petite application, un peu inutile, nous avons survolés quelques deux types de services. Sachez cependant qu'il en existe beaucoup d'autres, de même qu'il existe des *co-services*, permettant de ne représenter qu'une action. Ces autres types seront survolés dans la suite de cet article, lors de l'implémentation concrète de notre système de micro-blogging. Pour plus d'informations, n'hésitez pas à vous référer aux liens (et dans le contexte des services plus principalement au lien du manuel de Eliom) fournis en annexe.

A ce stade-ci, nous avons survolé une manière de naviguer dans la page, mais il reste encore beaucoup de choses a survoler. En effet, nous ne pouvons pas encore réaliser grand chose de concret avec ce que nous avons survolé.

### De la persistance

Nous ne pourrions sérieusement pas penser implémenter notre plateforme de micro-blogging sans avoir appréhender la persistance de données de notre application. En général, la persistance de données est un point crucial du développement d'applications web. Pour notre exemple, j'utiliserai **PostGresSQL**, (l'installation de ce système ne sera pas évoqué ici car il existe beaucoup d'articles bien plus claires que ce que je pourrais écrire).

Une fois de plus Ocsigen propose une solution étonnemment originale et fiable pour l'écriture de requête SQL. Pour ceux ayant déjà eu l'occasion de faire du PHP, sans framework, et pourquoi pas, avant la démocratisation de *PDO*, vous serez familier avec le fait d'écrire une requête SQL dans une chaine de caractères qui sera évaluée à l'exécution. Comme nous sommes des programmeurs OCaml, et que nous aimons les applications bien typées, ce genre de technique sauvage est à exclure, car on ne pourrait pas vérifier, *a la compilation* la cohérence de la requête. Une fois de plus Ocsigen propose une solution étonnemment

originale et fiable face à l'écriture de requête, en effet, le framework est dôté d'une bibliothèque, MaCaQue (pour Macro CamL Queries), pour rédiger des requêtes SQL vérifée à la compilation.

MaCaque permet de minimiser les erreurs à l'éxécution en vérifiant la bonne constitution d'une requête (en tirant partit des types fantômes). La bibliothèque est dôté d'une extension de syntaxe proposant une écriture proche de celle des compréhension.

**Inclure macaque dans son projet** Je vous propose de créer un projet pour l'application de micro-blogging, je l'ai nommée microblog (`eliom-distillery -name microblog`). Comme l'application Hello, nous organiserons nos fichiers d'une manière similaire, cependant, l'application étant plus riche, nous en aurons un peu plus.

Grâce au fichier `Makefile.options`, ajouter Macaque à la liste des paqets est assez simple. Il suffit d'ajouter un `SERVER_PACKAGES`. Donc la ligne `SERVER_PACKAGES :=` devient : `SERVER_PACKAGES := macaque.syntax`. On ajoute l'extension de syntaxe de Macaque (et macaque).

C'est dans le `Makefile.options` que l'on ajoute les paquets, séparés par un espace. Macaque ne concerne (à priori) que le serveur, on se contente donc de le placer dans la variable `SERVER_PACKAGES`. Nous pouvons maintenant utiliser Macaque dans notre projet !

**Principe général** Macaque repose principalement sur une couche bas niveau de PG'OCaml, un des outils les plus standards pour la communication avec un serveur Postgres. Il est tout à fait possible de se servir de PG'OCaml pour écrire des requêtes dans un projet utilisant Macaque, ce qui pourra parfois être utile étant donnée que Macaque ne prend pas en charge toutes les fonctionnalités de Postgres. (Cependant, je ne pense pas que c'est dans ce didacticiel que nous nous en rendrons compte).

Macaque, pour compiler correctement les requêtes (et les vérifier), doit avoir une connaissance du schéma de données des tables, c'est pour cela que je commence généralement par décrire les tables dans un fichiers spécifique (`tables.eliom`). Ensuite, je développe chacune des interfaces pour les tables dans des fichiers séparés, par exemple, la gestion utilisateur dans un fichier `users.eliom`.

**Accès à la base de données concurrent** Macaque sert à décrire des tables et des requêtes, mais la connexion est toujours prises en charge par PG'OCaml. Par défaut, l'accès à la base de données et bloquant, il n'est donc pas possible d'effectuer plusieurs connexions simultannées, ce qui n'est probablement pas dérangeant pour une application local, mais l'est un peu plus pour une application web, qui peut potentiellement être utilisé par plusieurs utilisateurs simultanément (ça dépend un peu de votre popularité).

Pour pallier à ce “problème”, on utilisera une version “spécialisée” de Macaque régie en interne par **Lwt**, la bibliothèque de traitement concurrent, permettant un accès non bloquant à la base de données. J'utilise un autre fichier, `db.eliom` qui contiendra toutes les fonctionnalités relatives à la base de données.

PG'OCaml dispose d'un foncteur capable de créer les utilitaires nécéssaire à la mise à disposition des outils de communication avec la base de données, nous lui donnerons comme module un module préparé spécialement pour, permettant les accès concurrent à la base de données. Voici le code que j'utilise pour `db.eliom`, j'ai pris l'habitude de l'utiliser pour chacun de mes projets utilisant Ocsigen et Macaque :

```
(* Multi access *)
module Lwt_thread = struct
  include Lwt
  include Lwt_chan
end
module Lwt_PGOCaml = PGOCaml_generic.Make(Lwt_thread)
module Lwt_Query = Query.Make_with_Db(Lwt_thread)(Lwt_PGOCaml)
```

**Connexion et fonctions utilitaires** Maintenant que nous avons créé les outils de manipulation de la base de données au moyen du foncteur `PGOCaml_generic`, nous allons créer la connexion et les outils pour effectueur des requêtes.

> En PHP, et dans d'autres technologies, il est courant de définir des variables pour l'hôte, l'utilisateur, le mot de passe et la base de données, ce qui permet de n'avoir qu'un seul fichier à changer en cas de changement de serveur (ou de déploiement). Je me sers d'un fichier `config.eliom` dans lequel j'ajoute :

```
(* Configuration de la connexion SQL *)
module SQL =
  struct
    let host = "localhost"
    let user = "nuki_psql"
    let pass = "MOT DE PASSE"
    let bdd  = "microblog"
  end
```

La connexion n'est pas très complexe, elle repose sur PG'OCaml et se limite à :

```
let connect () =
  Lwt_PGOCaml.connect
```

```
    ~host:Config.SQL.host
    ~user:Config.SQL.user
    ~password:Config.SQL.pass
    ~database:Config.SQL.bdd
    ()
```

Et un simple appel de `connect ()` suffira à connecter à la base de données. Cependant, la connexion à la base de données n'est pas spécialement une action que l'on voudrait effectuer "tout le temps". Et donc au lieu de se connecter "tout le temps", on aimerait garder une collection de connexion ouverte et de ne réutiliser les connexions ouvertes libre. Pour ça, il faut utiliser le module `Lwt_pool`, qui permettra de garder un `pool` de connexions ouvertes.

Pour cela, on se servira des deux fonctions du module `Lwt_pool` :

- `create n ?check ?validate f` où :
  - `n` : Nombre de membres maximum de la collection
  - `check` : Est appelé si l'utilisation d'un élément de la collection a échoué (nous ne nous en servirons pas dans cet exemple)
  - `validate` : Valide un élément avant son utilisation, et le recrée si l'élément est invalide
  - `f` : la fonction utilisée pour créer un nouvel élément
- `use p f` : Prend un élément du pool `p`, et le passe en argument à la fonction `f`.

Il faut spécifier le type de retour de la fonction `pool` pour éviter d'avoir une `value restriction`, ou alors l'emballer dans une fonction qui prend `unit`. La fonction `use` utilisant `pool` permet l'écriture de l'interface d'accès à une base de données. En règle général je me sers toujours de ce code qui ne demande pas de modifications (outre le fichier `config.eliom`) pour permettre l'accès rapide à une base de données :

```
(* Pool de connexion *)
let pool  =
  fun () -> (
    Lwt_pool.create
      16
      ~validate:Lwt_PGOCaml.alive
      connect
  )

(* Utilisation d'une resource du pool *)
let use f ?log x =
```

```
  Lwt_pool.use
    (pool ())
    (fun db -> f db ?log  x)

(* Interface d'accès à la base de données *)
let view ?log x =
  use Lwt_Query.view ?log x

let view_one ?log x =
  use Lwt_Query.view_one ?log x

let view_opt ?log x =
  use Lwt_Query.view_opt ?log x

let query ?log x =
  use Lwt_Query.query ?log x

let value ?log x =
  use Lwt_Query.value ?log x

let value_opt ?log x =
  use Lwt_Query.value_opt ?log x
```

L'argument `?log` permet de renvoyer la la requête sous forme de chaîne sur le channel de sortie, s'il est donné. C'est un outil pratique pour le déboguage.

Les fonction suffixées par `_opt` renvoient une option contenant le potentiel résultat, plutôt qu'une liste. Les fonctions suffixées par `_one` ne renvoient qu'une seule valeur (ou une exception en cas de retour nul). La différence entre une vue et une valeur est que la vue est un résultat de requête classique alors que la valeur est un champ en particulier.

Maintenant que nous avons tous les outils pour communiquer avec une base de données, nous allons pouvoir décrire formellement les tables que nous utiliserons dans notre application, pour que Macaque ait une connaissance du schéma de base de données et que les requêtes puissent être vérifiées.

> A ce stade ci, je partirai du principe que la base de données a bien été créée et que les tables présentées dans la description du projet ont été installées. (De même que le fichier `config.eliom` est correctement adpté à votre configuration).

### Description des tables du projet

### Liens de références pour la rédaction de cet article

Voici une tentative de construction de liste exhaustive des références que j'ai utilisées pour l'écriture de cet article.

- Page officiel de Ocsigen
  - Petits tutoriels
  - Manuel de Eliom
  - Manuel de Lwt
  - Cheatsheet
- Dépôt de MACAQUE
- Dépôt de Cumulus
- Ocsigen : une bouffée d'air frais pour le web
- Rethinking web interractions
- Ocsigen AT ICFP
- Guide du programmeur Eliom