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. Par exemple :

```
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 !