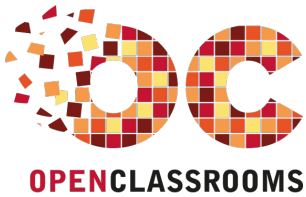
Un wiki avec Yaws en 13 minutes 37 secondes

Par rks`



www.openclassrooms.com

Sommaire

Un wiki avec Yaws en 13 minutes 37 secondes

Par rks`

Mise à jour : 01/01/1970

Ce tutoriel est destiné à des personnes connaissant déjà le langage de programmation Erlang ou aux personnes curieuses. Qu'ils soient toutefois prévenus que je vais tenir pour connues les principes de base du langage.

À l'instar de nombreux tutos tels « un forum en 20 minutes avec Django » ou encore « un blog en 37 minutes 25 secondes avec RoR » j'ai choisi pour ce tuto un titre que j'espère accrocheur et humoristique : quand bien même ce serait de la publicité mensongère, passons, l'idée est là. De plus, ce tutoriel se démarque de ceux cités précédemment car il n'explique non pas comment utiliser tel ou tel framework web, mais plutôt comment configurer et utiliser un serveur web, j'ai nommé yaws.

À savoir qu'il existe un framework de développement web fonctionnant avec Yaws nommé ErlyWeb. J'ai toutefois décidé de ne pas détailler son utilisation ici pour plusieurs raisons. Tout d'abord, il est encore en développement, donc incomplet et surtout **très peu** documenté. Ensuite, le support de Mnesia (la base de données Erlang, pour ceux qui ne la connaîtraient pas encore) y est plus qu'approximatif. Je vais donc vous présenter Yaws qui a les avantages de nous permettre l'utilisation de Mnesia et d'être bien documenté. Cela va de plus nous éviter d'avoir à mettre en place une architecture de type MVC.

Dans ce tutoriel, je vais tout d'abord vous expliquer brièvement comment configurer Yaws pour le wiki auquel nous souhaitons aboutir. Je vais aussi vous présenter les *bases* du fonctionnement de Yaws avant de nous lancer dans la création du wiki.
Sommaire du tutoriel :

Initiation

Je m'excuse par avance car je risque d'en choquer certains, mais je ne vais pas vous expliquer comment installer Yaws. En effet, je considère que lorsqu'on s'intéresse à Yaws, c'est qu'on programme en Erlang, et qu'une personne programmant en Erlang est un minimum curieuse et débrouillarde, parce que bon, faut déjà être motivé pour faire de l'Erlang. Mais bref...

Configurer Yaws

Bon, quand bien même je ne vous parle pas de l'installation de Yaws en elle-même, je vais quand même vous toucher deux mots quant à sa configuration. Tout ce dont nous allons nous occuper ici est de la modification du fichier yaws.conf. Encore une fois, je ne peux pas vous dire où le trouver, car je ne connais ni OS X, ni Windows et même sous Linux, cet emplacement varie selon votre installation, mais je suppose que vous saurez le trouver vous-mêmes. Chez les linuxiens, il sera très probablement dans /etc ou dans /usr/local/etc, pour les autres... Eh bien cherchez, c'est votre OS après tout.

Le fichier yaws.conf

Ceci est le fichier de configuration de Yaws, c'est grâce à ce fichier que nous allons spécifier les serveurs que nous souhaitons

créer, à quelle IP écouter pour être connectés à ces serveurs, et surtout : où se trouvent les fichiers que nous souhaitons rendre disponibles.

Vous vous apercevrez que ce fichier est abondamment commenté. Parmi les variables qui vont nous servir, on peut noter dès le début du fichier : ebin_dir. Il faudra ici ajouter le dossier dans lequel on placera les fichiers beam allant avec votre projet. C'est tout pour les variables, intéressons-nous aux serveurs ; la *création* d'un serveur suit ce motif :

Code : Autre

```
<server localhost>
        port = 8000
        listen = 127.0.0.1
        docroot = /usr/local/var/yaws/www
        appmods = <cgi-bin, yaws_appmod_cgi>
</server>
```

Ça, c'est le serveur par défaut. Vous trouverez en dessous un autre serveur, avec plus d'options, il ne nous servira pas : vous pouvez donc entièrement le commenter. Il faudra ensuite modifier le serveur par défaut pour qu'il redirige les visiteurs vers notre wiki. Pour cela, il faut modifier la variable docroot ; vous pouvez aussi si vous le souhaitez modifier le port d'écoute, bien que ce ne soit pas obligatoire. Voici par exemple la configuration de mon serveur :

Code : Autre

```
<server localhost>
        port = 8000
        listen = 127.0.0.1
        docroot = /home/dark-side/wiki/www
        appmods = <cgi-bin, yaws_appmod_cgi>
</server>
```

Voilà, vous êtes désormais parés pour la suite ! 

Lancer votre serveur

Informations complémentaires

Nos pages seront écrites dans des fichiers .yaws, fichiers dans lesquels nous pourrons écrire de l'HTML. Pour introduire du code en Erlang, on se sert des balises `<erl> ... </erl>`. Entre ces balises devra **toujours** figurer une fonction `out/1` dont l'argument sera un record que nous étudierons plus tard. Ces fonctions pourront renvoyer deux types de variables, un tuple du type : `{html, VosDonnees}`, qui est la manière la plus simple d'afficher des données, ou bien renvoyer un tuple du type `{ehtml, Ehtml}`, où `Ehtml` peut être au choix du type :

- `[Ehtml]`
- `{Tag}`
- `{Tag, Attrs}`
- `{Tag, Attrs, Corps}`

Où `Tag` est un atome du nom d'une balise HTML, `Attrs` sera une liste de tuple du type : `{NomAttrs, Valeur}` ; de même, `NomAttrs` sera un atome, `Valeur` pourra par contre être soit un atome, soit un string. Et pour finir, `Corps` sera, lui, une variable similaire à `Ehtml`, suivant donc à nouveau la syntaxe décrite précédemment. Pour que ce soit bien clair, voilà un exemple :

Code : Erlang

```
{ehtml,
```

```
        {table, [{bgcolor, red}],
           [
              {tr, [],
                 [
                    {td, [], "(0, 0)"},
                    {td, [], "(1, 0)"},
                    {td, [], "(2, 0)"}
                 ]},
              {tr, [],
                 [
                    {td, [], "(0, 1)"},
                    {td, [], "(1, 1)"},
                    {td, [], "(2, 1)"}
                 ]},
              {tr, [],
                 [
                    {td, [{colspan, "3"}], "(*, 2)"}
                 ]}
           ]}
     }
```

Qui produira le code HTML suivant :

Code : HTML

```
<table bgcolor="red">
  <tr>
    <td>(0, 0)</td>
    <td>(1, 0)</td>
    <td>(2, 0)</td>
  </tr>
  <tr>
    <td>(0, 1)</td>
    <td>(1, 1)</td>
    <td>(2, 1)</td>
  </tr>
  <tr>
    <td colspan="3">(*, 2)</td>
  </tr>
</table>
```

Bon, c'est sûr que présenté comme ça, ce n'est pas forcément très esthétique, mais c'est très utile, et **très** pratique. Cela permet en effet d'afficher des structures plus *complexes* qu'avec un simple tuple {</minicode>html, </minicode>}. La dernière chose à savoir est que vous pouvez créer d'autres fonctions que la fonction `out/1` entre les balises `<erl></erl>`. Attention par contre car ces fonctions sont locales, c'est-à-dire que vous ne pourrez y accéder que dans le *bloc erl* dans lequel elles sont définies.
Voilà : je crois que tout est dit.

Votre première page

On va faire simple, affichons un *Hello, World!*, voyez plutôt :

Code : HTML

```
<html>
  <head>
    <title>Wiki</title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-
8" />
  </head>

  <body>
```

```
        <p>Un peu d'HTML...<br>
        <erl>
          out(_) ->
             Hello = io_lib:format("Hello, ~p!", ["World"]),
             {html, Hello}.
        </erl>
        <br>Et encore du HTML...</p>
      </body>
    </html>
```

Bon, d'accord : si j'avais voulu faire simple, j'aurais fait :
`out(_) -> {html, "Hello, World!"} .`
Mais c'était pour vous montrer qu'on peut **vraiment** mettre n'importe quel type de code entre ces balises.

Lancer Yaws

Placez-vous dans le dossier où se trouve l'exécutable, ouvrez la console et faites : `yaws -i`. Cela va lancer Yaws dans le mode interactif, c'est-à-dire avec l'interprète Erlang habituel. Ce mode est très utile lors du développement. Une fois que votre projet sera stable, vous voudrez probablement lancer Yaws comme un daemon : pour cela, il suffit de faire : `yaws -D -heart`. Le `-D` va vous permettre de lancer Yaws en tant que daemon et l'option `-heart` va lancer le programme du même nom, qui va se charger de relancer Yaws si celui-ci crashe, ou ne répond plus.
Une fois ceci fait, vous pouvez ouvrir votre browser et vous rendre à l'adresse : http://localhost:8000.


Selon votre installation, il se peut que Yaws plante dès le lancement. Pour corriger ça, il suffit de le lancer en tant qu'administrateur ! 

Préparer le terrain

Mise en place de la base de données

Nous entrons maintenant dans le vif du sujet, à savoir les différents préparatifs avant de véritablement coder votre wiki. Il faut tout d'abord créer la base de données et les tables qui vont avec. Pour cela, nous utiliserons Mnesia, **la** base de données d'Erlang. Que je suppose connue de vous autres Erlangeux qui me lisez. Nous utiliserons un fichier .erl tiers, dans lesquels nous placerons une fonction qui se chargera des différentes tâches de création et d'initialisation. Je ne vais pas m'étendre sur ce fichier, qui ne nous intéresse que très peu, et qui est très facilement compréhensible, n'hésitez toutefois pas à le modifier pour qu'il corresponde à vos besoins.

Code : Erlang

```
-module(init_mnesia).
-export([do_this_once/0, pre_fill/0]).

-record(article, {id, titre, contenu}).

do_this_once() ->
    mnesia:create_schema([node()]), %% Crée une base de données sur
le noeud courant.
    mnesia:start(),                 %% Lance la base de données
tout juste créée.

    %% La commande suivante ajoute une table nommée « article » et
contenant les champs
    %% du record article. On précise que les données seront
ordonnées et qu'il y aura
    %% une copie faite sur le disque dur.
    mnesia:create_table(article, [{attributes, record_info(fields,
article)},
       {type, ordered_set},
       {disc_copies, [node()]}]),
```

```
        mnesia:stop(). %% On quitte la base de données pour que les
    changements soient enregistrés.

    pre_fill() ->
        Row = #article{id="0", titre="test", contenu="Ceci est un
    test."},
        F = fun() -> mnesia:write(Row) end,  %% crée la fonction qui va
    enregistrer Row dans la bdd
        mnesia:transaction(F).               %% on effectue la
    transaction
```

Ajoutez maintenant cette ligne à votre yaws.conf :
`ebin_dir={chemin_de_votre_projet}/beams`.
Ouvrez ensuite un interprète Erlang basique, compilez votre fichier puis quittez. Placez ensuite le .beam que vous aurez obtenu dans un sous-dossier de votre projet que vous appellerez beams. Lancez Yaws et faites :
`init_mnesia:do_this_once().`, cela va vous créer votre base de données ainsi que la table `article`. Vous pouvez aussi vous servir de la fonction `pre_fill/0` afin de créer un premier article, histoire de voir à quoi ressemble votre wiki dès que vous le lancez, sans avoir à créer un article en l'utilisant.

Attention toutefois à bien relancer Mnesia avant d'utiliser cette fonction. À noter que désormais, chaque fois que vous lancerez Yaws, il faudra faire un `mnesia:start()`.

Le problème

Là où les choses se corsent, c'est que Mnesia contrairement à SQL n'a pas de champ du type « auto$_{increment}$ ». Ce qu'il va falloir faire, c'est qu'à chaque fois que nous allons créer un nouvel article, nous allons devoir regarder dans la table l'id du dernier article rédigé. Heureusement pour nous, il y a déjà une fonction faite pour ça, c'est : `mnesia:last/1`. Celle-ci vous renvoie la *clé* du dernier élément de la table dont le nom a été passé en argument. La *clé* est la valeur dont se sert Mnesia pour ordonner les éléments de nos tables lorsque celles-ci sont de type `ordered_set`. Cette clé est par défaut la valeur du premier champ de l'élément. Dans notre exemple, ce sera la valeur du champ id, exactement ce qu'il nous faut ! 

GET et POST

Maintenant que vous avez un serveur prêt à l'emploi, il serait intéressant de voir comment ça fonctionne précisément. Il est notamment temps de s'intéresser au traitement des requêtes de type `GET` et `POST`. Pour ces dernières, nous allons devoir nous servir du record dont je vous ai parlé précédemment, si, si, souvenez-vous : celui que vos fonctions `out/1` reçoivent en argument. En fait, ce record est nommé `arg`. Il contient les champs : `headers`, `req` et `querydata`. Si durant votre utilisation de Yaws vous utiliserez manuellement le premier (puisque c'est lui-même un record), il existe des fonctions prédéfinies destinées à l'utilisation des deux autres : c'est ce que nous allons voir maintenant.

Les requêtes de type GET

Pour parser le résultat de ces requêtes, on va se servir de la fonction `yaws_api:parse_query/1`. Elle attend un record de type `arg` en argument et renvoie une liste de tuples. Ces tuples contiennent deux chaînes de caractères : la première contient le nom de la requête et la seconde sa valeur ; par exemple sur la page : http://www.monsite.eu/foo.yaws?bar=baz, le résultat de parse_query sera : `[{"bar", "baz"}]`.
Facile, n'est-ce pas ? 

Les requêtes de type POST

Le fonctionnement est ici identique, sauf qu'il faudra utiliser la fonction : `yaws_api:parse_post/1` au lieu de la fonction parse_query. Bref, rien de bouleversant. 
Nous pouvons désormais nous mettre au travail et commencer la création à proprement parler de notre wiki !

Créer des pages

Eh oui, il est grand temps de mettre les mains dans le cambouis ! 

Avant de nous occuper de l'affichage des pages, je pense qu'il convient de se concentrer sur leur création. Pour cela, on va créer une page edit.yaws qui va contenir le formulaire permettant de créer la page ou d'éditer des pages déjà créées. Commençons par mettre en place l'architecture HTML basique :

Code : HTML

```
<html>
  <head>
    <title>Wiki</title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-
8" />
  </head>

  <body>
    <p>Vous êtes actuellement en train de rédiger un article.<br>
    <erl>
      out(_) ->
         {ehtml,
             {form, [{action, save.yaws}, {method, post}],
                 [
                     {input, [{type, hidden}, {name, id}, {value,
"new"}], ""},
                     {input, [{type, text}, {name, titre}, {value,
"Titre"}], "<br>"},
                     {textarea, [{name, contenu}, {rows, 10}, {cols,
80}], "Votre article"},
                     {input, [{type, submit}, {value, "Ok"}], ""}
                 ]
             }
         }
    </erl>
    <br>Et encore du HTML...</p>
  </body>
</html>
```

Eh ! Pourquoi est-ce que tu utilises un tuple `ehtml`, au lieu de placer du code HTML normal ?

Bonne question ! Eh bien en fait l'utilisation d'un tuple `ehtml` va pouvoir me permettre d'adapter très facilement la page pour qu'elle soit préremplie si nous éditons une page déjà créée. Bref, en attendant, voyons plutôt comment réceptionner le résultat de ce formulaire et comment l'enregistrer. Pour cela, créons une page save.yaws.

Enregistrer la page

À partir de maintenant, je vais me passer de toute la structure HTML autour du code Erlang, car c'est légèrement répétitif et ça ralentit la lecture.

Sur ce : place au code !

Code : Erlang

```
out(Arg) ->
    case yaws_api:parse_post(Arg) of
        [{"id", I}, {"titre", T}, {"contenu", C}] ->
            case I of
                "new" -> Id = bdd:getNextId();
                _ -> Id = I
            end,
            case bdd:save(Id, T, C) of
                ok -> {html, "<p>Votre article a bien été
```

```
        sauvegardé.</p>"};
                        _ -> {html, "<p>Un erreur est apparu, impossible
        d'enregistrer.</p>"}
                    end;
                _ -> {html, "<p>Erreur, requête inconnue.</p>"}
            end.
```

Pour commenter rapidement ce que nous faisons ici : le premier filtrage de motif nous sert juste à vérifier que l'utilisateur est arrivé ici grâce au bon formulaire. Ensuite, on regarde si c'est un nouvel article ou juste un article qui a été modifié et on en déduit l'id en conséquence. Vient ensuite le moment où on essaye d'enregistrer dans la base de données : là, si tout se passe bien, on dit au visiteur que son article a bien été sauvegardé, sinon on l'informe qu'il y a eu un problème.

Bon sinon, je crois que quelques explications supplémentaires s'imposent. Notamment au niveau des expressions suivantes : `bdd:getNextId/0` et `bdd:save/3`. Voyez-vous, même s'il est possible de coder directement dans nos fichiers .yaws, je ne suis pas vraiment favorable à cette possibilité. Et ce, pour deux raisons :

1. ça nuit à la lisibilité ;
2. ce n'est pas vraiment le top côté réutilisation du code.

C'est pourquoi je place le *gros* de mes codes dans des modules séparés que je peux ainsi appeler depuis n'importe quelle page. On voit d'ailleurs que j'ai déjà créé un nouveau module : bdd. Encore une fois pour ce qui est de la lisibilité : je préfère séparer les morceaux de code qui n'ont rien à voir ensemble, et je vous encourage à faire de même, quitte à avoir parfois des modules ne contenant qu'une unique fonction. Bref, voyons ce module :

Code : Erlang

```
-module(bdd).
-compile(export_all).

-record(article, {id, titre, contenu}).

getNextId() ->
    Fun = fun() -> mnesia:last(article) end,
    case mnesia:transaction(Fun) of
        {aborted, _Raison} -> error;
        {atomic, Id} -> integer_to_list(list_to_integer(Id + 1))
    end.

save(Id, Titre, Contenu) ->
    Row = #article{id=Id, titre=Titre, contenu=Contenu},
    Fun = fun() -> mnesia:write(Row) end,
    case mnesia:transaction(Fun) of
        {aborted, _Raison} -> error;
        _ -> ok
    end.
```

Si vous connaissez déjà le fonctionnement de Mnesia, je pense que vous n'avez eu aucun problème avec ce fichier. Pour les autres je vais rapidement décrire le fonctionnement de la fonction `save`, qui est la plus compliquée et vous devriez facilement comprendre `getNextId`. Commençons : `Row` contient un record de type article, c'est-à-dire correspondant à la table article. On essaye de l'écrire dans la table à l'aide de la fonction `mnesia:write/1` ; si la transaction échoue, Mnesia nous renvoie un tuple du type `{aborted, Raison}`, on informe alors l'utilisateur que l'écriture a échoué, sinon on dit que tout est ok.

Afficher le résultat

Bien, maintenant que nous sommes capables de créer des articles, ça serait cool de les afficher, vous ne trouvez pas ? Pour cela on va créer une page see.yaws qui va se servir de l'id de l'article pour l'afficher. Pour lui communiquer cet id, on va se servir de requêtes de type `GET`. Comme je vous l'ai dit, le traitement de cette requête est **identique** au traitement d'une requête de type `POST`, si ce n'est la fonction qui change. Voyez vous-mêmes :

Code : Erlang

```
out(Arg) ->
```

```
        case yaws_api:parse_query(Arg) of
            [{"id", Id}] ->
                {Id, Titre, Contenu} = bdd:getById(Id),
                Edit = io_lib:format("<a href=\"edit.yaws?id=~s\">Éditer
cet article.</a>", [Id]),
                {ehtml,
                    [
                        {h1, [], Titre},
                        {p, [], Contenu},
                        {p, [], Edit}
                    ]
                };
            [] -> {html, "Erreur : vous devez préciser l'article à
afficher"};
            _ -> {html, "Erreur : requête non reconnue"}
    end.
```

Comme vous pouvez le voir, je fais appel à une nouvelle fonction :`bdd:getById/1`. Celle-ci va simplement sélectionner un article dans la base de données à l'aide de son id. Voyez plutôt :

Code : Erlang

```
getById(I) ->
    do(qlc:q([X || X <- mnesia:table(article),
      I == X#article.id])).
```

Cette fonction ne devrait pas vous sembler étrange si vous avez déjà utilisé Mnesia, pour les autres ? Disons que c'est un poil différent de SQL : avec Mnesia, les requêtes (de sélection du moins) sont effectuées grâce à de la compréhension de liste, une notion que vous vous **devez** de connaître si vous faites de l'Erlang avec un minimum de sérieux.

Bref. Cette fois, il nous faut faire deux choses pour que cette fonction marche : inclure une *library* et créer la fonction `do`.
Commençons par la library : placez la ligne suivante dans votre module bdd (avant toute fonction) :
`-include_lib("stdlib/include/qlc.hrl").`
Voici ensuite la fonction `do` :

Code : Erlang

```
do(Q) ->
    F = fun() -> qlc:e(Q) end,
    {atomic, Val} = mnesia:transaction(F),
    Val.
```

C'est une fonction générique qu'on retrouve dans quasiment tous les projets utilisant Mnesia. Elle se charge tout simplement d'effectuer la transaction (c'est-à-dire exécuter la requête) avec Mnesia et de nous faire parvenir le résultat.

Accéder à see.yaws

Tout cela est bien joli, mais je doute que vos visiteurs aient l'idée de taper l'adresse de cette page dans leur navigateur, et ne parlons pas de leur connaissance des id des articles. Une des possibilités (et c'est celle que nous mettrons en place ici) est de modifier votre page d'accueil pour lister *tous* les articles, et pour chaque article de faire un lien vers see.yaws avec l'id correspondant. Au boulot !

Code : Erlang

```
out(_) ->
    List = bdd:getAll(),
    ToShow = lists:map(fun({Id, Titre}) ->
```

```
            Lien = io_lib:format("<a href=\"see.yaws?id=~s\">~s</a>",
    [Id, Titre]),
        {li, [], Lien}
       end, List),
        {ehtml, {ul, [], ToShow}}.
```

Dans ce code, on va tout d'abord récupérer la liste de tous les articles existant dans la variable List, ensuite on va créer une nouvelle liste nommée ToShow. Cette liste va être de la même longueur que List, cela étant dû au fait que chacun de ses éléments seront ceux de List auquel on aura appliqué une fonction. Pour cela, nous nous servons de la fonction `lists:map/2`. Vous remarquerez que la fonction appliquée est ici une fonction anonyme introduite à l'aide du mot clé fun. Bref, cette fonction ne fait rien de particulier, elle formate juste les éléments de List pour qu'ils soient ensuite affichables par Yaws. Mais sinon : que nous manque-t-il, cette fois ? Au hasard, je dirais :`bdd:getAll/0`. Bon, heureusement qu'elle est facile, celle là, en plus c'est comme si on l'avait déjà faite, il suffit de reprendre `getById` en enlevant un garde. Le résultat :

Code : Erlang

```
getAll() -> do(qlc:q([ {X#article.id, X#article.titre} || X <-
mnesia:table(article)])).
```

Il faut bien penser à exporter toutes les fonctions (hormis la fonction `do`) qu'on vient de rajouter, sinon vous risquez d'avoir quelques problèmes.

Une fois que vous aurez compilé tous vos fichiers et placé les .beam au bon endroit, je crois que nous serons parés pour quelques tests dans la joie et la bonne humeur !

Édition et améliorations

Éditer vos pages

Bon : maintenant que vous pouvez créer des pages, il serait bien de pouvoir les éditer, c'est un wiki après tout. Pour cela, revenons sur le code de la page edit.yaws. Il faut la modifier pour recevoir le résultat d'une éventuelle requête de type `GET`. On sélectionnerait alors dans la table l'article demandé et on s'en servirait pour préremplir les champs de notre formulaire. Bref, après quelques secondes d'édition on aboutit à :

Code : Erlang

```
out(Arg) ->
    case yaws_api:parse_query(Arg) of
        [{"id", Id}] ->
            {Id, T, C} = bdd:getById(Id);
        _ -> T = "Titre", Id = "new", C = "Votre article"
    end,
    {ehtml,
     {form, [{action, save.yaws}, {method, post}],
      [
       {input, [{type, hidden}, {name, id}, {value, Id}], ""},
       {input, [{type, text}, {name, titre}, {value, T}], "<br>"},
       {textarea, [{name, contenu}, {rows, 10}, {cols, 80}], C},
       {input, [{type, submit}, {value, "Ok"}], ""}
      ]
     }
    }.
```

Et voilà ! Le wiki est désormais fini ! Comme promis, on a mis moins de XXX pour le créer, comme quoi même sans le dernier framework à la mode, on peut créer des trucs fonctionnels très rapidement.

Listing des fichiers

Je vous liste ici tous les fichiers et vous mets les codes correspondants, afin d'être sûr que vous n'ayez rien oublié ; j'ai aussi un peu modifié la structure HTML de certaines pages afin de faciliter la navigation des visiteurs ; bref, voilà les fichiers :

bdd.erl

Secret (cliquez pour afficher)

Code : Erlang

```
-module(bdd).
-compile(export_all).

-record(article, {id, titre, contenu}).

-include_lib("stdlib/include/qlc.hrl"). % requis pour les requêtes
vers Mnesia

do(Q) ->
    F = fun() -> qlc:e(Q) end,
    {atomic, Val} = mnesia:transaction(F),
    Val.

getNextId() ->
    Fun = fun() -> mnesia:last(article) end,
    case mnesia:transaction(Fun) of
    {aborted, _Raison} -> error;
    {atomic, Id} -> integer_to_list(list_to_integer(Id) + 1)
    end.

getById(I) ->
    do(qlc:q([{X#article.id, X#article.titre, X#article.contenu}
|| X <- mnesia:table(article),
                I == X#article.id])).

getAll() -> do(qlc:q([ {X#article.id, X#article.titre} || X <-
mnesia:table(article)])).

save(Id, Titre, Contenu) ->
    Row = #article{id=Id, titre=Titre, contenu=Contenu},
    Fun = fun() -> mnesia:write(Row) end,
    case mnesia:transaction(Fun) of
    {aborted, _Raison} -> error;
    _ -> ok
    end.
```

edit.yaws

Secret (cliquez pour afficher)

Code : HTML

```
<html>
  <head>
    <title>Wiki</title>
    <meta http-equiv="Content-Type" content="text/html;
charset=utf-8" />
  </head>

  <body>
```

```
      <p>Vous êtes actuellement en train de rédiger un article.<br>
        <erl>
          out(Arg) ->
              case yaws_api:parse_query(Arg) of
              [{"id", Id}] -> [{_, T, C}] = bdd:getById(Id);
              _ -> T = "Titre", Id = "new", C = "Votre article"
              end,
              {ehtml,
               {form, [{action, save.yaws}, {method, post}],
                [
                 {input, [{type, hidden}, {name, id}, {value, Id}],
""},
                 {input, [{type, text}, {name, titre}, {value, T}],
"<br>"},
                 {textarea, [{name, contenu}, {rows, 10}, {cols,
80}], C},
                 {input, [{type, submit}, {value, "Ok"}], ""}
                ]
               }
             }.
        </erl>
        <br>Et encore du HTML...</p>
    </body>
  </html>
```

index.yaws

Secret (cliquez pour afficher)

Code : HTML

```
  <html>
    <head>
      <title>Wiki</title>
      <meta http-equiv="Content-Type" content="text/html;
charset=utf-8" />
    </head>

    <body>
      <p>Bienvenue sur mon super wiki !<br>
        <erl>
      out(_) ->
          List = bdd:getAll(),
          ToShow = lists:map(fun({Id, Titre}) ->
                      Lien = io_lib:format("<a href=\"see.yaws?
id=~s\">~s</a>", [Id, Titre]),
                                      {li, [], Lien}
                end, List),
          {ehtml, {ul, [], ToShow}}.
        </erl>
        <br>Et encore du HTML...</p>
    </body>
  </html>
```

init_mnesia.erl

Secret (cliquez pour afficher)

Code : Erlang

```
-module(init_mnesia).
-export([do_this_once/0, pre_fill/0]).

-record(article, {id, titre, contenu}).

do_this_once() ->
    mnesia:create_schema([node()]), %% Crée une base de données
sur le noeud courant.
    mnesia:start(),                 %% Lance la base de données
tout juste créée.

    %% La commande suivante ajoute une table nommée « article »
et contenant les champs
    %% du recorde article. On précise que les données seront
ordonnées et qu'il y aura
    %% une copie faite sur le disque dur.
    mnesia:create_table(article, [{attributes, record_info(fields,
article)},
         {type, ordered_set},
         {disc_copies, [node()]}]),

    mnesia:stop(). %% On quitte la base de données pour que les
changements soit enregistrés.

pre_fill() ->
    Row = #article{id="0", titre="test", contenu="Ceci est un
test."},
    F = fun() -> mnesia:write(Row) end,  %% crée la fonction qui
va enregistrer Row dans la bdd
    mnesia:transaction(F).               %% on effectue la
transaction
```

save.yaws

Secret (cliquez pour afficher)

Code : HTML

```
<html>
  <head>
    <title>Wiki</title>
    <meta http-equiv="Content-Type" content="text/html;
charset=utf-8" />
  </head>

  <body>
    <p>
      <erl>
        out(Arg) ->
            case yaws_api:parse_post(Arg) of
            [{"id", I}, {"titre", T}, {"contenu", C}] ->
                case I of
         "new" -> Id = bdd:getNextId();
         _ -> Id = I
            end,
            case bdd:save(Id, T, C) of
         ok -> {html, "Votre article a bien été sauvegardé."};
         _ -> {html, "Une erreur est apparue, impossible
d'enregistrer."}
           end;
           _ -> {html, "Erreur, requête inconnue."}
           end.
      </erl>
      <br>Retour à l'<a href="index.yaws">index</a>.</p>
  </body>
```

```
</html>
```

see.yaws

Secret (cliquez pour afficher)

Code : HTML

```
<html>
  <head>
    <title>Wiki</title>
    <meta http-equiv="Content-Type" content="text/html;
charset=utf-8" />
  </head>

  <body>
    <erl>
      out(Arg) ->
          case yaws_api:parse_query(Arg) of
              [{"id", Id}] ->
                  [{_, Titre, Contenu}] = bdd:getById(Id),
                  Edit = io_lib:format("<a href=\"edit.yaws?
id=~s\">Éditer cet article.</a>", [Id]),
                  {ehtml,
                   [
                    {h1, [], Titre},
                    {p, [], Contenu},
                    {p, [], Edit}
                   ]
                  };
              [] -> {html, "Erreur : vous devez préciser l'article
à afficher"};
              _ -> {html, "Erreur : requête non reconnue"}
          end.
    </erl>
  </body>
</html>
```

Améliorations possibles

Bon, vous aurez pu vous en rendre compte, ce wiki est **très** basique. Notamment au niveau de l'affichage des pages, mais pas seulement. Voici une liste (non exhaustive) d'améliorations possibles :

- un système d'administration, histoire de supprimer les pages inutiles que des floodeurs pourraient créer ;
- un système d'historique pour les articles ;
- des catégories pour les articles ;
- ?

Bon courage en tout cas ! 
Vous savez désormais créer un wiki avec yaws ! N'hésitez pas à aller sur le site officiel pour découvrir comment utiliser les cookies, les sessions, les includes et d'autres systèmes utiles dans ce style !
Si jamais vous avez des questions, n'hésitez pas à passer sur le forum ou sur IRC !

Notes

Ce tuto a été écrit non pas en zCode, mais en mdown, un petit langage de mise en forme agréable à utiliser (et qui peut produire du zCode), conçu par rz0 ; vous pourrez trouver une comparaison entre la syntaxe mdown et le zCode sur cette page.

Merci à iPoulet, Pmol, Rudy et les autres pour leurs conseils et leurs nombreuses relectures !

Un tutoriel signé PHM