LTHMS

Zeste de savoir

Petit tour d'horizon des dépendances de pi-hoole

20 janvier 2019

Table des matières

% PETIT TOUR D'HORIZON DES DÉPENDANCES DE PI-HOOLE % lthms % 06 mai 2018

1. Introduction

Cela fait maintenant presque trois ans que j'utilise assez régulièrement Haskell. Je pense avoir acquis une certaine maîtrise —à défaut d'une maîtrise certaine — du langage. Si une part non négligeable de mon apprentissage s'est faite en écrivant du code, j'ai aussi lu pas mal de ressources très bien écrites et au moins aussi intéressantes. Ce sont très certainement grâce à elles qu'aujourd'hui, j'ai pu publier les premières versions de `pi-hoole` , une collection d'outils écrits justement en Haskell. En le faisant, je me suis rendu compte que ce qui était intéressant avec ce projet, c'était qu'il m'avait fait toucher à énormément de choses.

J'ai donc décidé de rendre à Internet ce qu'il m'a apporté, en couchant sur le papier numérique un modeste retour d'expérience ; et comme Zeste de Savoir — et tout ce qu'il représente — est aussi une ressource que j'utilise souvent, je me suis dit qu'un billet était un support qui avait du sens.

Mon objectif, vous l'aurez peut-être compris en lisant le titre, est de faire le point sur les dépendances dont `pi-hoole` tire parti. Hackage propose un nombre impressionnant de bibliothèques, qui sont dans des états variés. Il peut parfois être difficile d'y faire son choix. De ce point de vue, j'espère que ce qui suivra vous aidera, si un jour vous avez des besoins similaires aux miens.

i La vignette provient d'une œuvre originale de Joe Matthews, publiée ici en CC-0.

2. Un point sur le projet

Avant de me lancer à corps perdu dans mon « retour d'expérience », je juge utile de m'attarder un peu sur `pi-hoole` ; pour parler de `pi-hoole`, il me faut faire d'abord un petit crochet et évoquer Pijul ; comment le faire, sans d'abord toucher quelques mots de Darcs ?

Alors que tout le monde ne jure bien souvent que par git ou Mercurial, j'étais tombé il y a de cela quelques années sur darcs , un gestionnaire de version (VCS, de son petit acronyme anglais) qui proposait une approche différente. Sans entrer forcément dans les détails, Darcs est construit sur l'idée que deux changements qui ne modifient pas les mêmes lignes d'un projet devrait pouvoir être appliqué l'un après l'autre, ou l'autre après l'un, sans que l'ordre ne change rien à l'état final de dépôt. C'est très différent de ce que propose git, par exemple. L'ordre, pour lui, compte ; appliquer (*cherry-pick*) deux changements indépendants dans un ordre ou dans l'autre amènera le projet dans un état différent. D'ailleurs, le *cherry-picking* de git crée un nouveau *commit* ; les gens qui ont l'habitude d'utiliser cette fonctionnalité se sont très certainement retrouvés confronté à des conflits provoqués par la rencontre d'un « *commit* cloné » et de son original, lors d'un *merge*.

Pijul est un petit nouveau dans la cour des VCS. Il s'inscrit dans la continuité de Darcs, avec des concepts et des objectifs très similaires — notamment des patches qui *commutent*, selon la formule consacrée — mais une représentation de ce qu'est un dépôt et des changements, des algorithmes et une implémentation très différentes.

Actuellement en version `0.10.1`, un peu plus d'un an après l'ouverture de son code avec la publication de sa version `0.3`, il se stabilise lentement. Son écosystème demeure — ce n'est guère surprenant — encore pauvre. Si l'auteur principale de Pijul propose une forge logicielle « à la GitHub » , il n'existe pas encore de solution pérenne et éprouvée pour l'auto-hébergement de dépôts.

C'est là qu'intervient `pi-hoole` ; mon objectif avec ce projet est de proposer quelque chose d'approchant le duo gitolite + cgit. Le premier permet de faire du contrôle d'accès pour des dépôts hébergés sur un serveur personnel, en se basant sur l'authentification SSH par clef public. Le second est une interface web pour visualiser l'état courant de dépôts git publics.

Codé en Haskell, `pi-hoole` fournit trois exécutables distincts :

— `pi-hoole-cfg` sert à générer un fichier `.authorized_keys`, qui va contraindre l'utilisateur authentifié avec une clef public donnée à exécuter sa commande au travers d'un proxy particulier, en l'occurence…
— `pi-hoole-shell` est appelé par OpenSSH, après l'authentification de l'utilisateur réussie ; son but est d'autoriser, ou non, l'utilisateur à exécuter un appel à Pijul particulier, en se basant sur un fichier de configuration (en YAML) qui décrit pour chaque dépôt et utilisateur une liste de droits en écriture et lecture ;
— `pi-hoole-web`, enfin, permet de cloner les dépôts dits publics, mais fournit aussi des pages web résumant les dits dépôts.

Il y a donc pas mal de choses différentes à gérer. Fort heureusement, l'écosystème Haskell fournit tout ce dont j'ai besoin.

3. Parser du YAML : aeson et yaml

Le point de départ du projet `pi-hoole`, c'est le contrôle d'accès. Or, qui dit contrôle d'accès dit politique de sécurité. Il faut, pour chaque ressource (ici, des dépôts) définir les actions (lire ou écrire dans les branches des dépôts Pijul) que les acteurs (des utilisateurs authentifiés par SSH, ou des anonymes clonant *via* HTTP) seront autorisés à effectuer.

Niveau format de fichier, j'aime bien le YAML et mon choix s'est donc porté vers ce dernier. Il existe une très bonne bibliothèque pour gérer ce format, très justement nommée `yaml` . Elle est en fait une surcouche à une autre bibliothèque très puissante, `aeson` .

`aeson`, c'est *la* référence Haskell pour utiliser du JSON. Son fonctionnement est finalement assez classique : elle décrit un type générique (`Value`) qui représente une valeur encodée en JSON, et fournit deux *typeclasses* pour sérialiser un type donné en JSON (`ToJSON`, moralement `a -> Value`) ou déserialiser du JSON vers un type donné (`FromJSON`, moralement `Value -> Maybe a`).

Écrire les instances de `ToJSON` et `FromJSON` à la main est parfois nécessaire (c'était notamment mon cas pour certains types de `pi-hoole`), mais il faut savoir que grâce au mécanisme des `Generic` de Haskell, elles peuvent souvent être dérivés automatiquement. Ça marche très bien dans le cadre des *records*, même s'il faut activer une pragma pour dériver automatiquement les instances de la *typeclass* `Generic`.

Simplement avec ce bout de code :

```haskell
{-# LANGUAGE DeriveGeneric #-}

import           Data.Aeson    (FromJSON, ToJSON)
import           GHC.Generics  (Generic)

data Author = Author { userName :: String
                     , email    :: String
                     , posts    :: [Int]
                     }
  deriving (Generic)

instance ToJSON Author
instance FromJSON Author
```

On est capable de générer et/ou de parser ce JSON :

```json
{
  "userName": "lthms",
  "email": "my@email.com",
  "posts": [12, 154, 295]
}
```

Et ce YAML :

```yaml
userName: "lthms"
email: "my@email.com"
posts:
  - 12
  - 154
  - 295
```

Au final, le fichier de configuration de `pi-hoole` ressemble à ça :

```
groups:
  +owner: [.vhf, .spacefox, .sandhose]
  +contrib: [.artragis, .Eskimon, .Situphen]

repositories:
  zestedesavoir/zds-site:
    anon: +r[master]
    +contrib: +r +w[contrib]
    +owner: +r +w
```

Le coup d'implémentation est assez faible ; l'avantage, c'est qu'on a une configuration très lisible, ÀMHA.

4. La ligne de commande : optparse-generic

Il arrive toujours un moment où « ça commence à marcher » et qu'il convient de se soucier de l'interface utilisateur. La plupart des logiciels que je développe s'utilisent en ligne de commande. Il existe plusieurs façons d'implémenter une bonne interface en ligne de commande (CLI, en anglais). Ma bibliothèque préférée pour le faire en Haskell est `optparse-generic`, en cela qu'elle limite drastiquement le travail nécessaire pour avoir quelque chose de convaincant. Grâce à la *typeclass* `Generic` (encore elle), il est possible d'avoir quelque chose de fonctionnel en quelques lignes de code.

Le but est de définir un type qui va servir de spécification pour notre CLI.

Si je prends l'exemple de `celtchar`, un outil que j'ai écrit pour pouvoir écrire mes récits dans un langage de mon cru et obtenir de beaux PDF en bout de chaîne, ce type ressemble à ça :

```haskell
data Command =
    Build { root   :: FilePath
          , output :: Maybe FilePath
          }
  | Deps { root :: FilePath
```

```
        }
  | New FilePath
  deriving (Generic, Show)
```

D'une manière similaire à ce qu'il faut faire avec `aeson`, on peut instancier la bonne *typeclass* pour le type command, sans avoir besoin d'écrire la moindre ligne de code :

```
instance ParseRecord Command
```

À l'usage, on retrouve quelque chose de familier et de facile à utiliser :

```
$ celtchar --help
celtchar

Usage: celtchar (build | deps | new | info)

Available options:
  -h,--help                Show this help text

Available commands:
  build
  deps
  new
$ celtchar build --help
Usage: celtchar build --root STRING [--output STRING]

Available options:
  -h,--help                Show this help text
```

5. Une application web : wai et wrap

Pijul a été codé pour qu'il soit possible de cloner un dépôts qui soit simplement derrière un serveur web donnant accès à des fichiers statiques. Il `GET` les fichiers internes dont il a besoin. Le but de `pi-hoole-web` est de servir de proxy HTTP, de se mettre en proxy de ces appels et de déterminer si, pour un dépôt donné, le rôle `anon` a les droits supplémentaires.

J'ai décidé de faire de `pi-hoole-web` un démon, qui est sollicité par `nginx` pour savoir quoi faire. Une configuration minimale du célèbre serveur web ressemble à ça :

```
server {
  listen 80;
  server_name pijul.lthms.xyz;
```

```
4
5     access_log /var/log/nginx/pijul.lthms.xyz.access.log;
6     error_log /var/log/nginx/pijul.lthms.xyz.error.log;
7
8     location / {
9       proxy_pass http://localhost:8080/;
10    }
11  }
```

Puisqu'on en est à digresser, je ne me suis pas embêté à faire de `pi-hoole-web` un démon au sens UNIX du terme : je laisse `systemd` s'en charger. J'ai une petite *unit* pour ça (qu'il faudra que je modifie pour durcir un peu le processus, notamment l'empêcher de lire ou d'écrire n'importe où).

```

```

Bref.

Pour en revenir au Haskell de la conversation, j'avais besoin d'une bibliothèque minimale pour faire une application Web. Pas de fonctionnalités évoluées, globalement je n'avais qu'un besoin : avoir accès au type de la requête (`GET`, `POST`, etc.) et au `path` (par exemple, `pijul/pijul/.pijul/changes.wUAdAoNq`). Il me suffisait ensuite de parser ce chemin, pour en déduire l'opération qu'essayait de faire `pijul` ; dans le cas où cette dernière était légale au sens de la politique de sécurité définie en YAML par l'administrateur, alors je pouvais exécuter la-dite commande et retourner son résultat au client web.

Pour faire ce genre de choses, la solution la plus populaire en Haskell est d'utiliser `wai` . Pour ce que j'en sais, la plupart des *frameworks* web écrit en Haskell utilise cette brique de base eux aussi (c'est le cas notamment de `servant` , une bibliothèque avec du typage fou pour faire des API Web facilement).

Une `Application` web au sens `wai` est globalement une fonction qui, pour une `Request` donnée, doit produire une `Response` (le tout dans la monade `IO` pour pouvoir interagir avec le monde extérieur). La bibliothèque fournit un certain nombre de fonctions pour manipuler des `Request` et créer des `Response`, je n'avais pas besoin de beaucoup plus.

Manipuler `wai` permet cependant de toucher du doigt un des détails les plus *chiants* (j'ose le mot) de Haskell.

Les chaînes de caractères !

Pour ceux qui l'ignorent, une chaîne de caractère (`String`) en Haskell, c'est… une liste de `Char`. Littéralement. Ce n'est pas très efficace, selon tous les points de vue, alors les gens ont défini des types alternatifs (`Text` et `ByteString`, en gros). Et puis, Haskell est paresseux (*lazy*) par défaut, mais ce n'est pas toujours une bonne idée. Alors `Text` et `ByteString` ont tous les deux deux implémentations (une stricte, une paresseuse)…

L'usage s'impose d'utiliser `Text` pour du… texte, et `ByteString` pour le binaire. Mais en attendant, il n'est pas rare d'avoir une bibliothèque qui fonctionne avec un `Text` paresseux, une

autre avec un `ByteString` strict, etc. On passe son temps à convertir d'un type à l'autre, c'est assez déprimant.

Bref.

Une fois notre `Application` définie, on peut utiliser `wrap`, qui est un serveur web écrit en Haskell, paramétrisé par une `Application` telle que définie par `wai`. Easy.

6. Moteur de template : shakespeare

Je voulais terminer ce petit tour d'horizon des bibliothèques Haskell utilisées dans le cadre de `pi-hoole` par `shakespeare`. Si le but premier de `pi-hoole` est de fournir des fonctionnalités proches de ce que peut apporter gitolite, je veux aussi donner un proto cgit à l'écosystème de Pijul.

Ça veut donc dire générer des pages web, ce que je fais dans `pi-hoole-web` grâce à `shakespeare`. Il s'agit d'un projet fournissant un certain nombre de langage de *template* web. Pour ma part, j'en utilise deux :

— Hamlet, pour le HTML
— Cassius, pour le CSS

Hamlet est intéressant, parce qu'on peut mettre du code Haskell dedans (c'est ma passion) et que le langage est sensible à l'indentation. La *template* « racine » de mon projet ressemble donc à cela :

```

```

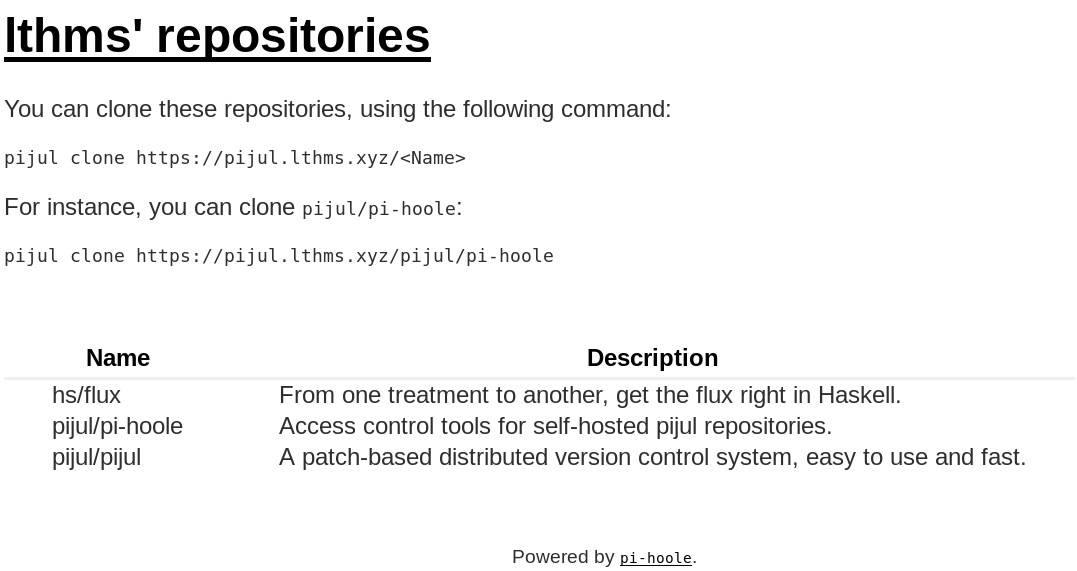
Pour les utiliser, la première chose à faire est de spécifier l'ensemble des routes disponibles dans l'application web. Comme souvent en Haskell, on fait ça en définissant un type. Pour le moment, je n'ai qu'une page d'accueil et un fichier de CSS, donc mon type `Route` est très simple.

```
1 data Route = Home
2            | Css
```

On définit ensuite une fonction qui transforme les valeurs de `Route` en des URL. Après ça, on peut utiliser Hamlet et Cassius avec ce type `Route`. L'idée est la suivante : plutôt que de taper directement les URL à la main dans les *templates*, on va utiliser des valeurs de `Route`, avec l'opérateur `@{...}` (par exemple, `@{Home}`).

Cassius ressemble pas mal à du SASS, en gros.

Après, je ne suis pas un développeur web, et ça se ressent. Au final, ma page listant les dépôts publics sur mon serveur ressemble à cela .



lthms' repositories

You can clone these repositories, using the following command:

```
pijul clone https://pijul.lthms.xyz/<Name>
```

For instance, you can clone `pijul/pi-hoole`:

```
pijul clone https://pijul.lthms.xyz/pijul/pi-hoole
```

Name	Description
hs/flux	From one treatment to another, get the flux right in Haskell.
pijul/pi-hoole	Access control tools for self-hosted pijul repositories.
pijul/pijul	A patch-based distributed version control system, easy to use and fast.

Powered by pi-hoole.

FIGURE 6. – pi-hoole-web en action

7. Conclusion

Et voilà, c'est tout pour ce petit panel de bibliothèque Haskell. J'ai vraiment beaucoup aimé écrire `pi-hoole` et je pense que c'est en partie dû au fait que l'écosystème Haskell fournit toutes les briques de base dont j'avais besoin. J'ai ainsi pu me concentrer sur ce que je voulais vraiment faire (du contrôle d'accès), sans avoir besoin de me soucier à la tuyauterie. Je pense que c'est vraiment important.

J'espère que vous aurez pu trouver quelque chose à retirer de ce billet. C'est encore mieux si il a pu vous aider dans vos propres développement Haskellien.