noIPasaran

Service web vérifiant l'intégrité des serveurs DNS

Rapport de l'épreuve intégrée

Section Bachelier en Informatique

EPFC
Avenue Charles Thielemans, 2
1150 Woluwe-Saint-Pierre

Elève
Bernard DEBECKER
bernard.debecker@gmail.com

Promoteur
Olivier Nisole
olnisole@epfc.eu

Juin 2013

Remerciements

Je remercie Olivier Nisole, mon promoteur, qui malgré son emploi du temps chargé a toujours été disponible. Ses conseils avisés m'ont fait progresser considérablement dans mon travail.

Et bien sûr, Alain Silovy pour avoir accepté mon sujet.

Je remercie l'inventeur de Node.js, Ryan Dahl. Son langage est formidable et réellement passionnant.

C'est grâce à l'émission Tracks d'Arte que j'ai appris l'existence des contre-cultures et de certains mouvements alternatifs. Cette vision nouvelle a lentement fait germer l'idée de ce TFE dans mon esprit.

La sortie du film The Pirate Bay Away From Keyboard au début de cette année m'a donnée un angle de vue différent sur ce procès très médiatisé dans lequel s'affrontaient les majors et les fondateurs du site d'échange de fichiers torrents, The Pirate Bay.

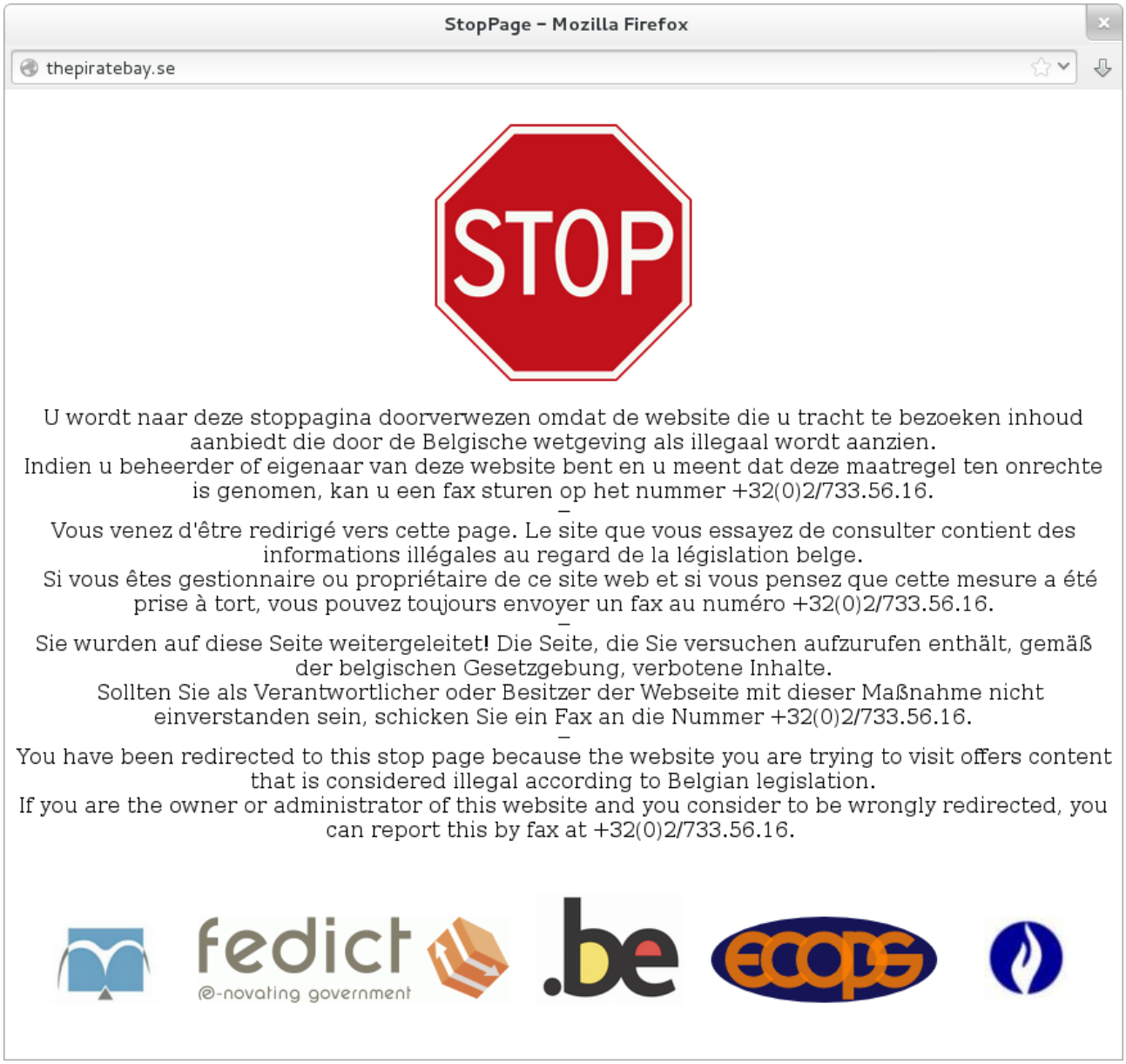
Des principes qui m'ont plu et que j'ai adopté, le Datalove. C'est une philosophie qui prône l'amour de la communication, toute forme de communication ; qui défend l'échange et l'accès à l'information, comme un droit et surtout comme un besoin. Ou quand la liberté et l'intérêt commun l'emportent sur le profit personnel. Ceci explique la liberté totale d'utilisation, de redistribution ou même de ré-attribution de mon travail.

A COMPLETER

Table des matières

1. Introduction

Le 26 septembre 2011, la cour d'appel d'Anvers a rendu un arrêt donnant à Telenet et à Belgacom, les plus importants fournisseurs d'accès à Internet belge, 14 jours pour mettre en place un blocage DNS de 11 adresses associées au site The Pirate Bay sous peine d'astreinte. Belgacom et Telenet se sont exécutés, les requêtes vers ces sites ont aussitôt été redirigées vers un site informant du blocage.



Il n'a fallu que quelques mois à Belgacom, voyant que certains de ses abonnés contournaient le blocage DNS en utilisant d'autres serveurs que les leurs, pour bloquer purement et simplement toutes requêtes vers le site. Or, aucun arrêt n'a été rendu dans ce sens, cette décision est propre à Belgacom.
Il existe donc une forme de censure sur Internet en Belgique. Il n'en fallait pas plus pour attiser ma curiosité.

J'ai appris qu'il existait un grand nombre de sites internet bloqués en Belgique répartis en quatre grandes familles :

- Les sites de vente de médicaments
- Les sites de téléchargement illégaux (fortement discutable)
- Les sites de vente d'objets contrefaits
- Les sites de jeux non-autorisés (pour lesquels il existe une liste noire publique, 74 sites actuellement, sur le site de la Commission des jeux de hasard du SPF Justice)

Je regrette qu'une liste noire complète des sites interdits ne soit pas accessible au public.
La frontière entre la protection du consommateur et la censure est mince et dépend, selon moi, essentiellement de la transparence de ses actions et de la communication avec les citoyens.
Ne sommes-nous pas en droit de savoir quels sites précisément sont interdits dans notre pays ? Pourquoi cette liste noire est-elle gardée aussi farouchement ?

Bien sûr, à l'étranger, la situation est parfois bien pire.

A COMPLETER

Dès lors, comment avoir confiance dans les serveurs DNS que les FAI mettent à la disposition de leurs abonnés ? Comment vérifier que les FAI ne font pas de l'excès de zèle dans le blocage ?

2. But du projet

Le but du projet est de permettre à un utilisateur sans connaissance approfondie en réseau ou en informatique en général de vérifier la qualité des serveurs DNS proposés par son fournisseur d'accès à Internet.

La vérification se fera soit par le biais d'une requête individuelle en introduisant l'adresse d'un site internet à vérifier, soit en demandant une vérification globale de l'intégrité de son fournisseur d'accès qui fonctionnera sur base d'une liste de sites internet sensibles qui s'affinera à l'utilisation.

Le service devant, par essence, être accessible à n'importe qui dans des conditions pas toujours optimales (client d'un cybercafé, employé d'une entreprise avec accès restreint), j'ai décidé que mon projet serait une application web qui ne nécessite aucune installation ou extension de navigateur Internet.

L'interface est volontairement simple et dépourvue de termes techniques afin d'éviter de faire fuir un utilisateur non averti.
Les utilisateurs confirmés ne seront pas en reste, les détails du test pouvant s'afficher sur demande.

3. Technologies et méthodes utilisées

3.1. Côté serveur

3.1.1. Node.js

Node.js est un framework événementiel créé pour développer des applications réseau scalables en JavaScript, principalement des serveurs web. Il repose sur la machine virtuelle V8 de Google.

Pour illustrer sa simplicité, voici un exemple de serveur HTTP renvoyant un Hello world

```
var http = require('http');

http.createServer(
        function (request, response) {
                response.writeHead(200, {'Content-Type': 'text/plain'});
                response.end('Hello world\n');
        }
).listen(8000);

console.log('Server running at http://localhost:8000/');
```

Il est à constater qu'on travaille ici à un niveau assez bas. Il faut donc définir soi-même le header de la réponse, signaler que la réponse est finie, définir le contenu séparément du header.
C'est ici qu'Express entre en jeu.

Pour justifier l'utilisation de Node.js pour ce projet, je dois d'abord défendre le choix de JavaScript.
Je vais le comparer à Java qui est également exécuté sur une machine virtuelle.

Selon le benchmark ci-dessous, Java est sans surprise plus rapide et plus gourmand en mémoire que JavaScript V8.

Program Source Code	CPU secs	Elapsed secs	Memory KB	Code B	≈ CPU Load
regex-dna					
JavaScript V8	3.78	**3.80**	310,368	373	2% 1% 1% 100%
Java 7	22.63	22.66	557,080	1284	0% 0% 0% 100%
spectral-norm					
JavaScript V8	15.70	**15.71**	7,220	328	0% 1% 1% 100%
Java 7	16.13	16.14	14,924	950	1% 0% 0% 100%
fannkuch-redux					
JavaScript V8	73.30	73.33	5,288	539	0% 0% 0% 100%
Java 7	71.26	71.28	13,936	1282	0% 0% 0% 100%
n-body					
JavaScript V8	42.83	42.85	5,736	1527	0% 1% 0% 100%
Java 7	24.40	24.41	13,996	1424	0% 0% 0% 100%
binary-trees					
JavaScript V8	35.01	35.07	361,480	467	0% 0% 0% 100%
Java 7	15.70	15.75	506,592	603	0% 1% 0% 100%
fasta					
JavaScript V8	19.42	19.43	7,548	791	0% 0% 0% 100%
Java 7	4.92	4.92	14,932	1507	0% 1% 0% 100%
k-nucleotide					
JavaScript V8	338.18	339.02	429,952	451	0% 0% 0% 100%
Java 7	50.69	50.73	494,040	1630	1% 0% 0% 100%
reverse-complement					
JavaScript V8	13.11	13.12	193,164	498	0% 0% 0% 100%
Java 7	1.88	1.92	511,484	745	2% 1% 3% 100%
pidigits					
		No program			
Java 7	1.39	1.40	31,616	1826	0% 1% 1% 99%
fasta-redux					
		No program			
Java 7	0.12	0.12	?	1443	0% 33% 0% 100%
mandelbrot					
JavaScript V8		Bad Output			
Java 7	0.21	0.22	212	796	0% 0% 9% 95%

JavaScript est effectivement plus lent que Java, malgré le formidable moteur V8 développé par Google.
D'où l'intérêt de Node.js qui utilise un modèle orienté événement, non-bloquant.

Pour illustrer cette différence entre les deux langages, je me permets une parabole culinaire :

Albert et Simone ne se connaissent pas et vont, chacun dans leur cuisine, préparer une tarte aux pommes pour leurs invités.
Albert représente Java et Simone, Node.js.
Mes deux protagonistes ne sont pas de grands chefs et vont donc effectuer une seule

opération à la fois durant toute la préparation du goûter.

Albert procède de cette manière :

- Il épluche et coupe les pommes en tranches (5 minutes)
- Il mélange les pommes avec le sucre et les épices (1 minutes)
- Il étale la pâte qu'il a achetée toute faite dans un moule (1 minute)
- Il dispose les pommes sur la pâte (2 minutes)
- Il fait préchauffer le four (10 minutes)
- Il enfourne la tarte (1 minute)
- Cuisson de la tarte (25 minutes)
- La tarte sortie du four, il va mettre la table pour ses invités (10 minutes)

Au total, Albert a mis 55 minutes pour préparer son goûter.

Simone, avec son modèle non bloquant, procède de cette manière :

- Elle fait préchauffer le four (1 minute)
- Elle épluche et coupe les pommes en tranches avec amour (10 minutes)
- Elle mélange les pommes avec le sucre et les épices (2 minutes)
- Elle étale la pâte qu'elle a achetée toute faite dans un moule avec précaution (2 minutes)
- Elle dispose les pommes sur la pâte avec minutie (4 minutes)
- Le four est chaud, elle enfourne la tarte (1 minute)
- Elle peut commencer à mettre la table en veillant à l'alignement parfait des assiettes (20 minutes)
- Il lui reste 5 minutes avant la fin de la cuisson, elle prend un bon livre et se détend en attendant la tarte (5 minutes)

Au total, Simone a mis 45 minutes pour préparer son goûter et a même eu le temps de s'octroyer un moment de lecture.

En conclusion, Albert exécute plus vite les étapes, mais perd un temps conséquent à attendre lors des étapes les plus longues.

Simone est plus lente à l'exécution, mais comme elle utilise un modèle non-bloquant, elle peut passer à l'étape suivante pendant qu'Albert attend bêtement que son four préchauffe.

Bien sûr, il s'agit un exemple grossier et exagéré dans un sens comme dans l'autre. Mais il permet de comprendre une des différences fondamentales entre Java et Node.js.

Pour être un peu plus technique, voici un petit Hello world permettant d'illustrer le modèle non-bloquant.

```
1  setTimeout(function(){
2                  console.log("world");
3          }, 2000);
4
5  console.log("hello")
```

Aussi étonnant que ça puisse paraître, le résultat en console sera bien :

hello
world

Node.js ne permet pas de s'arrêter ou de tourner dans une boucle sans rien faire d'autre.

Que fait alors le programme dans le Hello world ci-dessus, entre le moment où il a affiché *hello* et le moment où il affichera *world* ?
Il *idle*, c'est-à-dire qu'il attend qu'on lui demande de faire quelque chose.
Et comme il n'a plus rien à faire ensuite, le programme s'arrête.

Voyons maintenant un programme qui ne s'arrête pas :

```
1  setInterval(function(){
2                  console.log("world");
3          }, 2000);
4
5  console.log("hello")
```

Dans ce cas, le programme ne s'arrête pas de lui-même, il affiche *hello world world world*... car Node.js sait lorsqu'il lui reste quelque chose à faire. Node.js conserve un compteur de callback, et garde l'event loop actif tant qu'il y a une tâche à effectuer.

Cette notion de callback mérite également d'être développée car c'est une des particularités de Node.js pour exploiter son caractère asynchrone.

Un autre exemple :

```
1  executer = (mot, callback) ->
2          callback mot
3
4  executer 'Hello', (callback) ->
5          console.log callback
```

Il est intéressant d'observer ici que la fonction *executer* reçoit deux paramètres : un mot et un callback. Le callback est une fonction qui sera appliquée sur le mot.

Lorsque j'appelle executer, je lui passe le mot "Hello" et en second argument une fonction anonyme qui affiche en console la valeur retournée.

Ce n'est qu'un petit aperçu de ce que permet un callback. Il est davantage illustré dans les cas d'utilisation (v. Test d'une URL*).*

En plus de ce modèle particulier, Node.js repose sur le moteur JavaScript V8 de Google. Initialement développé pour leur navigateur Chrome, il apporte un fonctionnement innovant pour l'exécution de code JavaScript.
L'intérêt de ce moteur repose sur la compilation du JavaScript en langage machine à l'exécution. Le code compilé est optimisé dynamiquement lors de celle-ci.

Je ne vais trop entrer dans les détails. Je vous invite, si le sujet vous intéresse, à aller

visiter le site de V8 (v. Références).

Ce mode de fonctionnement répond parfaitement à mes besoins pour ce projet car je dois effectuer de nombreuses requêtes sortantes vers des serveurs DNS. Le temps de réponse de ceux-ci fluctuant, je ne peux pas me permettre d'additionner linéairement les temps de réponse. Node.js me permet d'envoyer successivement toutes les requêtes et de recevoir les résultats au fur et à mesure, sans perdre de temps.

Ce projet aurait pu donc s'appeler Simone.

3.1.2. Express

Express est un framework très léger pour Node.js qui lui fournit un panel de fonctionnalités pour construire des applications web. Il permet l'utilisation de middlewares qui sont développés ci-après.

```
1  var express = require('express')
2          ,       app = express.createServer();
3
4  app.get('/', function(req, res) {
5          res.send('Hello world');
6  });
7
8  app.listen(3000);
```

Avec Express, il n'est plus nécessaire de définir le header ou quoi que ce soit, il s'en charge. Dans ce cas-ci, *app.get('/', (...))* ajoute la notion de route que le serveur prendra en charge. Toute autre route ne renverra rien.

La notion de contrôleur comme on l'entend dans une application web en Java n'existe pas en Node.js et Express. Dans mon projet, les routes sont définies dans des fichiers séparés uniquement par volonté de clarté dans la structure.
Il n'y a pas de notion de MVC. La seule qui existe réellement en Express est la notion de vue.

3.1.2.1. Middleware et modules utilisés

a) Connect-assets

Connect-assets sert des fichiers .coffee et .styl compilés en .js et .css à l'exécution du serveur.
Ceci évite la tâche redondante de devoir les compiler manuellement avant l'exécution.
De plus, il permet d'y accéder en utilisant le raccourci :

```
1  != css('nomdufichiercss')
2
3  != js('nomdufichierjs')
```

b) Node-mysql

Node-mysql est un module entièrement écrit en JavaScript est ce qu'il y a de plus performant pour utiliser une base de données de type MySQL.
Son utilisation est simple et ses possibilités très poussées. A usage égal, il a été démontré qu'il peut être plus performant qu'un module écrit en C pour effectuer le même travail. Pour les explications complètes, je vous invite à regarder la présentation vidéo du module faite par son créateur, Felix Geisendörfer (v. Références).

c) Request

Request me permet de faire des requêtes HTTP de manière très simple. Le module HTTP inclus dans Node.js permet de faire le même travail, mais de façon moins intuitive.

SCREEN

d) Marked

Marked est un module très performant qui permet de convertir un fichier Markdown en code HTML.
Il est intégralement écrit en JavaScript.

e) Async

Async rajoute des fonctions d'exécution asynchrone à Node.js.
Il me permet de contrôler précisément l'exécution de fonctions en parallèle et de n'envoyer le callback qu'une fois toutes les opérations effectuées.

SCREEN

f) Underscore

Underscore est un ensemble de fonctions destiné à manipuler différents types d'objets de façon très simple : grouper des tableaux, concaténer des listes sur base de certains paramètres et autres fonctions utiles.

g) MD5

MD5 permet de hasher facilement et simplement n'importe quoi en utilisant le protocole MD5.

```
1  console.log md5 'message'
2
3  >> "78e731027d8fd50ed642340b7c9a63b3"
```

h) Node-toobusy

Node-toobusy permet de vérifier si le temps passé par un opération dans la file d'attente de tâches dans le serveur n'est pas supérieur à une valeur donnée. Par défaut celle-ci est de 70 millisecondes.

i) Native-dns

Native-dns est indispensable pour mon application car la librairie inclue dans Node.js

permettant d'effectuer des requêtes sur un serveur DNS ne permet quasiment aucun paramétrage.

Ce module me permet de faire une même requête sur différent serveurs, de calculer le temps de la requête ou encore de définir le temps de timeout.

3.1.3. CoffeeScript

CoffeeScript est un langage de programmation qui se compile en JavaScript. Sa valeur ajoutée sont les sucres syntaxiques inspirés de Ruby, Python et Haskell. Ils lui permettent de rendre le code plus lisible.
Un autre avantage de CoffeeScript est qu'il permet d'écrire en moyenne 1/3 de ligne en moins qu'un programme équivalent en JavaScript, tout en n'ayant aucun effet négatif sur les performances, au contraire.

Par exemple, une classe JavaScript donne :

```
1  var MaClasse = (function() {
2          function MaClass() {
3                  alert('Constructeur')
4          }
5
6          MaClasse.prototype.faitQQChose = function() {
7                  alert('fait quelque chose')
8          };
9
10         return MaClasse;
11 })();
12
13 c = new MaClasse();
14 c.faitQQChose();
```

L'équivalent CoffeeScript est nettement plus clair et plus court :

```
1  class MaClasse
2          constructeur: ->
3                  alert 'constructeur'
4
5          faitQQChose: ->
6                  alert 'fait quelque chose'
7
8          c = new MaClasse()
9          c.faitQQChose()
```

Ou encore, une boucle *for* pour itérer dans une liste en JavaScript :

```
1  for (var i = 0; i < list.length; i++) {
2          var item = list[i];
3          process(item)
4  }
```

Cette même boucle en CoffeeScript :

```
for item in list
        process item
```

Avec ceci, viennent d'éventuels désavantages :

- Le langage est très sensible aux espaces et à l'indentation
- Le compilateur n'est pas toujours très précis en cas d'erreur

C'est finalement une question d'habitude.

Le Hello World Express en CoffeeScript :

```
express = require 'express'
        app = express.createServer()

        app.get '/', (req, res) ->
                res.send 'Hello world'

        app.listen 3000
```

3.1.4. Jade

Jade est un moteur de template HTML haute-performance développé spécifiquement pour Node.js.
Il permet d'écrire des pages HTML sans balise.

Une simple page HTML telle que :

```
<!DOCTYPE html>
        <html>
                <head>
                        <title>
                                Exemple de HTML
                        </title>
                </head>
                <body>
                        Ceci est une phrase avec un <a href="cible.html">hyperlien</a>.
                        <p>
                                Ceci est un paragraphe
                        </p>
                </body>
        </html>
```

Peut s'écrire en Jade :

```
!!!5
html
        head
                title Exemple de HTML
        body
                Ceci est une phrase avec un
                        a(href="cible.html") hyperlien
                p Ceci est un paragraphe
```

En plus de posséder une structure très légère, Jade permet également d'exécuter du code sur base des valeurs qu'on transmet au template.

Pour illustrer ceci, un exemple pratique d'un template auquel je passe une liste d'éléments :

layout.jade

```
!!!5
html
        head
                title #{title} | Exemple
        body
                block content
```

itemview.jade

```
extends layout

block content
        #rowid.rowclass
                ul.list
                        each item in items
                                li= item.name
```

J'appelle la vue *itemview* en utilisant Express :

```
res.render 'itemview', view: 'itemwiew', title: 'Ceci est un', items: {{name: "Jean"},{name:"Marc"}}
```

Ce qui produit le code HTML suivant :

```
<!DOCTYPE html>
<html>
        <head>
                <title>
                        Ceci est un | Exemple
                </title>
        </head>
        <body>
                <div id="rowid" class="rowclass">
                        <ul>
                                <li>Jean</li>
                                <li>Marc</li>
                        </ul>
                </div>
        </body>
</html>
```

Quelques explications s'imposent :

- Jade supporte l'héritage de template. Un template peut donc en étendre un autre.
- Les attributs *id="unId"* et *class="uneClasse"* d'un élément s'écrivent respectivement *#unId* et *.uneClasse*
- Les tags * * sont optionnels et supposés lors de l'utilisation d'id ou de classe. On peut écrire, en reprenant l'exemple précédent, *#rowId.rowclass* et obtenir le tag parfaitement formé * *
- Les variables passées sont accessibles soit par l'utilisation de la balise *#{variable}* ou directement après un tag *p= variable*

3.1.5. Stylus

Stylus est au CSS ce que Jade est au HTML. Simple, intuitif et complet. Il permet la rédaction de feuilles de style de façon très simple.

Pourquoi écrire :

```
body {
        font: 12px Helvetica, Arial, sans-serif;
}

a.button {
        -webkit-border-radius: 5px;
        -moz-border-radius: 5px;
        border-radius: 5px;
}
```

Quand on peut se contenter de :

```
border-radius()
  -webkit-border-radius arguments
  -moz-border-radius arguments
  border-radius arguments

body
  font 12px Helvetica, Arial, sans-serif

a.button
  border-radius 5px
```

Le code Stylus est évidemment compilé en CSS par la suite.

3.2. Côté client

3.2.1. Bootstrap

Bootstrap est une collection d'outils front-end pour créer simplement des design templates grâce aux fichiers CSS et JS qui le composent.

Je l'utilise principalement pour la structure des éléments et la présentation des fenêtres modales de modification de la partie admin.

3.2.2. Flat-UI

Cette librairie CSS repose sur la structure de Bootstrap, mais offre de la gaieté et de l'originalité dans les couleurs.

3.2.3. jQuery

Cette librairie JavaScript permet de manipuler les éléments du code HTML d'un page. En plus d'être très puissante, elle est extrêmement légère.

Comme la décision a été prise de ne pas utiliser de JavaScript côté client, l'utilisation en est principalement faite dans la partie admin du site, pour faciliter les modifications des informations contenues dans la base de données.

a) DataTables

DataTables est un plugin jQuery permettant de trier, filtrer et organiser facilement la présentation de données sous forme de tableau.

b) Validate

Validate est un plugin de validation qui permet de valider côté client le contenu de champs de type *input text* sur base d'expression régulière. Cela me permet de garder une base de données propre et correctement structurée sans devoir effectuer ces tests côté serveur.

3.3. Hébergement

Le site a pour vocation d'être hébergé par Heroku. Si je n'avais pas un problème de module (v. Problèmes rencontrés).

Heroku est un service de PaaS. Un des avantages d'Heroku, par rapport à un PaaS comme AppFog, est que le code ne doit pas être uploadé avec tous les modules.

Lors de la création d'une application, un dépôt Git est créé. Il suffit alors de pusher le code source sur le dépôt. Une fois le code chargé, le service détecte le type d'application avec le fichier Procfile, règle l'environnement selon ce qui a été déclaré dans le fichier package.json de l'application et télécharge automatiquement les dépendances nécessaires.

Il est ensuite possible d'avoir accès aux logs de l'application, d'y ajouter des add-on pour gérer le cache ou une base de données locale.

La base de données est hébergée sur AlwaysData durant la période de développement afin d'avoir une source de données unique, que le serveur soit exécuté sur le PaaS ou en local.

Il y a un point négatif à cela : le temps des requêtes entre le serveur et la base de données dépend de la rapidité de la connexion internet mise à la disposition du serveur.

Une fois l'application en production, la base de données sera migrée sur ClearDB.
Lors du transfert vers MongoDB, elle sera hébergée sur MongoLab.

4. Résultats

4.1. Architecture du serveur

La structure du serveur est générée par Skeleton.

```
noIPasaran
├┬ app
│ ├── app.coffee
│ ├┬ assets
│ │ ├┬ css
│ │ │ └── styles.styl
│ │ └┬ js
│ │   └── scripts.coffee
│ ├┬ controllers
│ │ └── controllers.coffee
│ ├┬ helpers
│ │ └── index.coffee
│ ├┬ routes
│ │ └── index.coffee
│ └┬ views
│   ├── 404.jade
│   ├── index.jade
│   └── layout.jade
├┬ config
│ └── boot.coffee
├┬ lib
│ └┬ myapp
│   └── my_custom_class.coffee
├── package.json
├── Procfile
├── public
├── README.md
└── server.coffee
```

4.1.1. app

a) app.coffee

app.coffee est le fichier de configuration d'Express.

```
# Modules
express = require 'express'
http = require 'http'
app = express()

# Boot setup
require("#{__dirname}/../config/boot")(app)

# Configuration
app.configure ->
  port = process.env.PORT || 3000
  if process.argv.indexOf('-p') >= 0
    port = process.argv[process.argv.indexOf('-p') + 1]

  app.set 'port', port
  app.set 'views', "#{__dirname}/views"
  app.set 'view engine', 'jade'
  app.locals.pretty = true
  app.use express.static("#{__dirname}/../public/lib")
  app.use express.favicon()
  app.use express.logger('dev')
  app.use express.bodyParser()
  app.use express.methodOverride()
  app.use require('connect-assets')(src: "#{__dirname}/assets")
  app.use app.router

app.configure 'development', ->
  app.use express.errorHandler()

# Routes
require("#{__dirname}/routes")(app)

# Server
http.createServer(app).listen app.get('port'), ->
  console.log "Express server listening on port #{app.get 'port'} in #{app.settings.env} mode"
```

b) assets
Le dossier assets contient les fichiers Stylus et CoffeeScript qui seront compilés en mémoire RAM par connect-assets pour être utilisé par le client.

c) controllers
C'est ici que les fichiers contrôleurs se trouvent. Deux contrôleurs sont définis : admincontroller et maincontroller. Cette division à pour but d'offrir plus de clarté et de lisibilité. Le contrôleur n'en est pas vraiment un, c'est simplement un ensemble de fonctions qui fait le lien entre les données requises par une route et les classes permettant d'accéder à ces données.

d) helpers
helpers.coffee est un fichier généré par Skeleton permettant d'effectuer un chargement automatique des contrôleurs et des classes lors du démarrage du serveur.

e) routes
C'est au sein du fichier index.coffee du dossier routes que les requêtes sont distribuées aux contrôleurs. La politique de sécurité d'accès aux pages est également gérée au sein de ce fichier.

f) views
Le dossier views contient les templates Jade.

4.1.2. config

Le dossier config ne contient qu'un seul fichier, boot.coffee. Il permet, lorsqu'il est chargé dans app.coffee, de charger automatiquement le contenu des dossiers lib et controllers.

4.1.3. lib

Lib contient les classes qui sont utilisées au sein du serveur.

a) check.coffee

Cette classe contient les fonctions qui permettent de garder la base de données de sites intègre.

b) db.coffee

Le fichier db est une pseudo classe de DAO. Il n'y a en effet pas de modèle déclaré dans l'application. Cette classe sera à terme refactorée en une vraie classe de DAO à l'aide de MongoDB et Mongoose qui permettent de déclarer des modèles et de simplifier leurs manipulations dans la base de données.
Ce fichier n'est donc composé que de différentes fonctions permettant d'insérer, modifier, récupérer ou supprimer des informations dans la base de données.

c) distance.coffee

distance.coffee est une petite classe pour calculer la distance réelle entre deux coordonnées géographique à vol d'oiseaux, en tenant compte de la courbure de la terre. Elle permet de fournir à titre indicatif la distance entre un serveur DNS et la position d'un utilisateur.

d) ip.coffee

La classe la plus importante du projet. Elle contient toutes les fonctions nécessaires à la manipulation, la recherche d'informations basées sur les URL et les adresses IP.

e) staticmap.coffee

Cette classe est utilisée pour générer l'URL d'une carte Google Maps Static.

4.1.4. node_modules

C'est ici que sont installés les modules déclarés dans le fichier package.json

4.1.5. public

Ce dossier contient les fichiers qui n'ont pas besoin d'être compilés, tels les images, fichier CSS ou JS provenant de sources externes.

4.1.6. Autres fichiers

a) LICENCE

Le texte de licence devient un indispensable et fait souvent couler beaucoup d'encre sur Internet. J'ai pour ma part opté pour la plus libre d'entre toutes, la Do What The Fuck you want to Public License.

b) Procfile

Fichier spécifique à l'utilisation d'Heroku comme hébergeur. Il permet de déclarer le type d'application qui est déployée.

c) README.md

Le fichier README, incontournable, c'est le fichier de présentation qui est visible sur les dépôts de code. Il est rédigé en markdown.

d) package.json

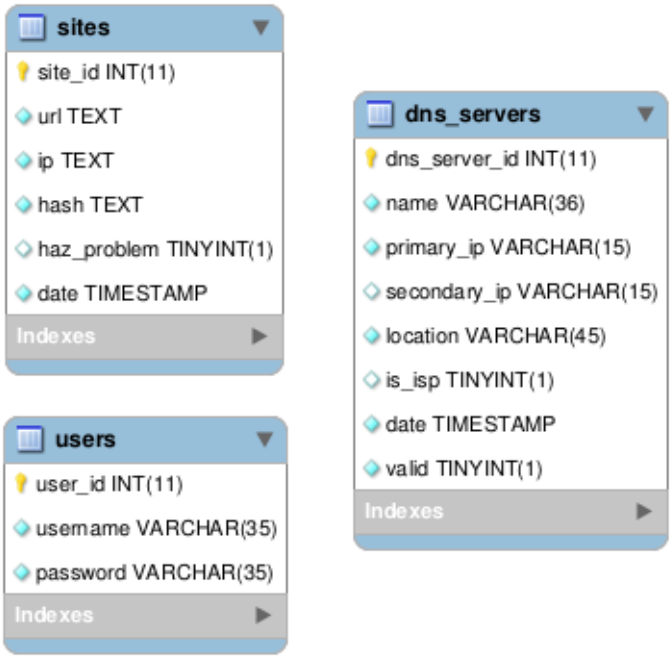
Ce fichier permet de déclarer le nom et les dépendances de l'application. Les dépendances sont téléchargées sur NPM, le packet manager de Node.js. Ce fichier permet de préciser la version de Node.js ou d'un module à utiliser.

e) server.js

Le fichier primaire du serveur. C'est lui qui est appelé lors du chargement du serveur, tel que déclaré dans le fichier package.json.
Il ne contient que deux informations : déclarer explicitement que l'exécution nécessite le compilateur CoffeeScript et que le reste du code se trouve dans le fichier app.coffee.

4.2. Structure de la base de données

La base de données se compose de 3 tables.



Comme on peut le constater, ce modèle n'est absolument pas relationnel. C'est pour ça qu'il est aisément concevable et réalisable de passer à une base de données noSQL dans le futur.

La table dns_servers contient les serveurs DNS enregistrés.

- name : le nom des serveurs DNS, du service qui les emploient ou du fournisseur d'accès à Internet auquel ils appartiennent
- primary_ip et secondary_ip : les adresses IP des deux serveurs
- location : Global si les serveurs ne sont pas lié à un FAI, le pays du FAI dans le cas contraire
- is_isp : Un booléen signalant si les serveurs appartiennent à un FAI
- date : Le timestamp d'insertion du serveur DNS dans la base de donnée
- valid : Un booléen indiquant si le serveur est validé, c'est à dire qu'il peut être utilisé pour effectuer des tests dessus

La table sites contient les sites qui ont été soumis aux tests.

- url : l'URL du site sous la forme www.example.com.
- ip : l'IP la plus probable du site selon les serveurs DNS sûrs. Celle-ci est vérifiée toutes les 24 heures pour les sites persistants
- hash : le code HTML de la page obtenue en effectuant une requête HTTP sur cette URL, hashé en MD5
- haz_problem : un booléen indiquant si le site a connu des problèmes. Permet d'assurer sa persistance dans la base de données
- date : le timestamp de dernière utilisation du site. Si le site n'a pas été demandé depuis 24h et qu'il ne connaît pas de problème, il est supprimé lors du nettoyage quotidien.

La table users est uniquement utilisée pour l'accès à la partie admin du site.

- username : le nom d'utilisateur
- password : le mot de passe hashé en MD5.

4.3. Cas d'utilisations

4.3.1. Test d'une URL

Dans la page index du site se trouve un *input text*. Le contenu de ce champ, lorsque l'utilisateur clique sur le bouton *Go*, est passé en POST à l'url */query.*

example.com

La requête arrive alors au sein du fichier routes/index.coffee qui contient les différentes routes.

```
10          app.post '/query', app.maincontroller.query
```

L'application, app, reçoit donc une méthode de type POST adressée à */query* et elle appelle la méthode query du maincontroller.

```
12              @query = (req, res) ->
13                      queryStr = req.body.query
14                      # Check if the query is an IP or an URL
15                      app.ip.isIp queryStr, (isIp) ->
16                              if isIp
17                                      res.redirect "/ip/#{queryStr}"
18                              else
19                                      # Check if there is a . means it could be a URL
20                                      dot = queryStr.split('.').length - 1
21                                      if dot
22                                              res.redirect "/url/#{queryStr}"
23                                      else
24                                              res.redirect "/404/#{queryStr}"
```

Cette méthode query reçoit en paramètre la requête et la réponse qui servira de callback. La valeur de la requête, c'est-à-dire le contenu de l'input, est extraite du corps de la requête dans la variable *queryStr*. La valeur de cette variable est ensuite testée afin de

déterminer si c'est une adresse IP, une possible URL (déterminée simplement s'il y a un point dans le texte) ou un contenu autre.
Le cas qui nous intéresse ici est si la valeur est une URL. Le callback est alors utilisé pour établir une redirection vers */url/queryStr*
Retour par les routes, et cette fois-ci, c'est la route */url/:url* qui est utilisée.

```
11          app.get '/url/:url', app.maincontroller.url
```

La notation *:url* permet de récupérer la valeur située après */url/* dans les paramètres de la requête.
La méthode *url* du maincontroller est appelée.

```
26              @url = (req, res) ->
27                      # Get the url
28                      url = req.params.url
29                      # clean the url after the first /
30                      urlArr = url.split('/')
31                      url = urlArr[0]
32                      # Add www. in front of the url
33                      url = "www.#{url}" if url.indexOf 'www.', 0 < 0 and url.split('.').length - 1 < 2
34                      app.ip.getIpAndData req, url, (data) ->
35                              console.log data.local[0].primary_result
36                              res.render 'url', view: 'url', title: "#{url}", url: url, ip: data.site.ip,
37                              clientip: data.clientip, country: data.country, resultlocal: data.local
```

L'URL est donc récupérée dans les paramètres. Elle est ensuite formatée pour être sous la forme :

www.example.com

C'est ensuite la méthode *getIpAndData* auquel est passée la requête, nécessaire pour récupérer l'IP du client, ainsi que l'url précédemment récupérée.

```
10          @getIpAndData = (req, url, data) ->
11                  result = new Object()
12                  getSite url, (site) ->
13                          site.ip = site.ip.split ','
14                          result.site = site
15                          getClientIP req, (clientip) ->
16                                  result.clientip = clientip
17                                  getIpCountry clientip, (country) ->
18                                          result.country = country
19                                  getIpISP clientip, (isp) ->
20                                          app.dao.getServerByName isp, (ispServers) ->
21                                                  #console.log ispServers
22                                                  resolveLocalServers url, ispServers, (ispAnswers) ->
23                                                          checkIfAnswerIsValid(result.site.ip, answer, (valid) ->
24                                                                  answer.valid = valid
25                                                          ) for answer in ispAnswers
26                                                          result.local = ispAnswers
27                                                          #console.log result
28                                                          data result
```

Dans getIpAndData les opérations suivantes sont effectuées :

Je vérifie si le site est présent dans la base de données.
S'il est présent, les données du site sont récupérées.
Dans le cas contraire, l'URL est soumise aux serveurs DNS sûrs pour récupérer l'IP qui lui est relative.

La première action va être de récupérer la ou les IP sur une sélection de serveurs DNS

sûrs.

Les serveurs choisis pour cette tâche sont :

- Google Public DNS
  Les points forts des serveurs DNS de Google sont sans aucun doute leur rapidité et leur disponibilité.
  Ils sont théoriquement neutre.
  Le seul point négatif est le fait que l'adresse IP de l'utilisateur (supprimée après 24 heures), son FAI et ses informations de géolocalisation sont enregistrées. Au nom de la performance.

- Level3
  Avant l'arrivée des serveurs DNS de Google, le plus connu était Level3 grâce à son IP facile à retenir : 4.2.2.2. Il ne jouit plus aujourd'hui de sa gloire d'antan, mais est toujours aussi neutre qu'en 1993.

- censurfridns.dk
  Créés par une seule personne, Thomas Steen Rasmussen, ces serveurs DNS danois ont pour but de fournir gratuitement des serveurs DNS certifiés sans censure. Cette initiative est née depuis que le gouvernement danois pratique la censure sur Internet de la même manière que le gouvernement belge : en forçant les FAI à bloquer des résolutions d'adresses.

- SmartViper
  Ceux-ci appartiennent à la société markosweb. Ils représentent une alternative sérieuse à Google DNS en terme de performance.

Je n'ai pas retenu OpenDNS, malgré son nom, car il repose sur les serveurs de Google et se finance grâce à de la pub qu'ils proposent lorsqu'une URL n'est pas résolue.

Les détails de ces serveurs DNS récupérés dans la base de données sont passés à la fonction *resolveGlobalServers*. Pour chaque serveur, c'est la fonction *treatGlobalServer* qui est appelée. Le résultat *resolved* retourné en callback est ajouté à la variable *result*, un tableau de String en effectuant une union. En effet, une union de deux ensembles est l'ensemble qui contient tous les éléments qui appartiennent au premier ou au deuxième. La variable *result* contient bien toujours une seule fois le même résultat.

```
86                resolveGlobalServers = (url, servers, data) ->
87                        result = []
88                        count = 0
89                        treatGlobalServer(url, server, (resolved) ->
90                                result = _.union(resolved, result)
91                                count++
92                                data result if count is servers.length
93                        ) for server in servers
```

Pour chaque fournisseur, l'URL est résolue sur les deux serveurs de façon parallèle.
Une fois les deux résultats de retour, la réponse est renvoyée en callback.

```
102                 treatGlobalServer = (url, server, answers) ->
103                         result = []
104                         async.parallel [
105                                 (callback) ->
106                                         resolve url, server.primary_ip, (answer1) ->
107                                                 result = _.union(result, answer1.addresses) if answer1.addresses
108                                                 callback()
109                                 ,(callback) ->
110                                         resolve url, server.secondary_ip, (answer2) ->
111                                                 result = _.union(result, answer2.addresses) if answer2.addresses
112                                                 callback()
113                                 ], (err) ->
114                                         throw err if err
115                                         answers result
```

La fonction *resolve* est la fonction de plus bas niveau de l'application.

Je construis tout d'abord une variable question qui contient le type de requête à faire sur le serveur DNS.

La requête *req* est ensuite construite sur base de la question, de l'IP du serveur sur lequel la requête va être faite et le timeout. I

ci, le délai réglé à 1000 millisecondes. La plupart des serveurs répondent en moins de 300 millisecondes, il n'est pas nécessaire d'attendre plus d'une seconde pour déclarer un serveur en timeout. Cette valeur pourrait être abaissée afin d'optimiser le temps des requêtes.

Ce qui est déclaré ensuite sont les actions à effectuer en fonction de l'état de la requête.

Si elle timeout, la réponse l'indiquera.

S'il y a un message, chaque adresse IP est ajoutée à la liste d'adresses de la réponse.

Lors de la réception du message de fin, le temps écoulé est calculé et passé à la réponse.

La requête est alors envoyée.

Cette fonction illustre parfaitement le modèle asynchrone de Node.js. On ne sait pas ce qui sera effectué, ni quand, donc on prévoit et on envoie le résultat en callback quand la requêtes est finie.

```
129                 resolve = (url, server, data) ->
130                         question = dns_.Question({
131                                 name: url,
132                                 type: 'A'})
133                         response = new Object()
134                         start = Date.now()
135                         req = dns_.Request({
136                                 question: question,
137                                 server: {address: server},
138                                 timeout: 3000
139                                 })
140                         req.on('timeout', () ->
141                                 response.timeout = true
142                         )
143                         req.on('message', (err, answer) ->
144                                 addresses = []
145                                 getAddress(a, (address) ->
146                                         addresses.push(address) if not _.isUndefined(address)
147                                 ) for a in answer.answer
148                                 response.addresses = addresses
149                         )
150                         req.on('end', () ->
151                                 delta = Date.now() - start
152                                 response.time = delta.toString()
153                                 data response
154                         )
155                         req.send()
```

S'il devait y avoir des IP contradictoires dans les résultats obtenus, chaque IP est appelée à l'aide d'une requête HTTP pour obtenir le contenu HTML de la page. Ce contenu est hashé à l'aide de l'algorithme MD5.
Le résultat de ce hashage est alors comparé.

Si deux IP produisent le même résultat, la probabilité est grande que ces deux adresses IP appartiennent bien au même site.

Le résultat obtenu est alors enregistré dans la base de données et les données du site enregistrées sont alors renvoyées pour la continuité du test.

Dans un même temps, l'IP du client est récupérée sur base de sa requête. Pour cela, deux possibilités :

- Récupération du header de la requête X-Forwarded-For (XFF)
- Récupération de la *remoteAddress* dans les paramètres de connexion de la requête.

Dans le cas où la requête vers le serveur est passée par un proxy pour l'atteindre, le header sera modifié en fonction et structuré de la manière suivante :

X-Forwarded-For: IP client, IP proxy 1, IP proxy 2

Cette information est malgré tout à prendre avec prudence car le header peut être modifié par le client. Malgré cela, c'est tout de même la source principale d'IP cliente qui sera retenue.
S'il ne devait pas y avoir de header, c'est la *remoteAddress* de la connexion qui sera retenue.

```
57                  getClientIP = (req, ip) ->
58                          ipAddress = null
59                          forwardedIpsStr = req.header 'x-forwarded-for'
60                          if forwardedIpsStr
61                                  forwardedIps = forwardedIpsStr.split ','
62                                  ipAddress = forwardedIps[0]
63                          ipAddress = req.connection.remoteAddress if not ipAddress
64                          ipAddress = '81.247.34.211' #BELGIQUE
65                          #ipAddress = '91.121.208.6' #FRANCE
66                          ip ipAddress
```

L'IP récupérée va servir à tenter d'obtenir le fournisseur d'accès à Internet de l'utilisateur.

Il s'agit d'effectuer une requête de type Reverse DNS sur l'IP obtenue. La méthode utilisée n'est pas sûre à 100%. Il est en effet possible que l'IP possède un canonical name autre que celui fourni par le FAI en utilisant un service comme DynDns.

```
79                  getIpISP = (ip, isp) ->
80                          dns.reverse ip, (err, domains) ->
81                                  throw err if err
82                                  #console.log domains
83                                  segments = domains[0].split '.'
84                                  isp segments[segments.length-2]
```

Exemple de résultat pour une requête Reverse DNS sur l'IP 81.247.34.211

211.34-247-81.adsl-dyn.isp.belgacom.be

Si cette méthode ne donne pas de résultat, ou que l'ISP de l'utilisateur n'est pas présent dans la base de données, c'est le pays duquel émane cette IP qui sera retenu.

Dans un cas comme dans l'autre, le ou les serveurs DNS correspondants sont récupérés dans la base de données.

Les requêtes sont de nouveau effectuées sur ces serveurs. Les résultats sont ensuite comparés aux résultats des serveurs sûrs. Un fois les tests terminés, le callback de cette méthode, data, contient les données qui seront affichées dans la page de résultat.

Ces données sont structurées comme suit :

```
{ "site": {     "site_id": 1,
                "url": "www.example.com",
                "ip": ["192.168.0.1"],
                "hash": "78e731027d8fd50ed642340b7c9a63b3",
        "haz_problem": 0,
        "date": "Tue May 21 2013 18:46:50 GMT+0200 (Romance Daylight Time)"},
        "clientip": "127.0.0.1",
        "local": [ {"valid": true,
                "name": "ExampleDnsServer",
                "primary_ip": "195.238.2.21",
                "secondary_ip": "195.268.2.22",
                "primary_result": [{"addresses": ["192.168.0.1"], "time": "30"}],
                "secondary_result": [{"addresses": ["192.168.0.1"], "time": "27"}]
                } ]
}
```

Détaillons ceci :

- *site* est l'objet récupéré dans la base de données. L'adresse IP qu'il contient est celle renvoyée par les serveurs sûrs. Le hash est la valeur hashée du code HTML récupéré sur base de cette adresse IP.
- *clientip* est l'adresse IP du client.
- *local* contient les résultats des serveurs dit locaux. Si, comme dans cet exemple, le fournisseur d'accès Internet du client a pu être récupéré, il n'y a qu'un résultat. Celui-ci contient un booléen, *valid*, résultat du test comparatif entre le résultat de ce serveur et le résultat des serveurs sûrs. Les valeurs de *name*, *primary_ip* et *secondary_ip* sont récupérées de la base de données et passées au client pour information. Les résultats *primary_results* et *secondary_result* contiennent la ou les adresses IP renvoyées par ce serveur et le temps pris par ce serveur pour répondre (en millisecondes).

4.3.2. Routine de nettoyage de la base de données *sites*

Tous les jours, une routine de nettoyage de la base de données *sites* est effectuée.
Celle-ci a pour but principal de ne pas surcharger cette base de données, compte tenu de la rapidité effective pour récupérer les informations qu'elle contient.

Tout site ne posant pas problème et n'ayant pas été consulté durant la journée est supprimé de la base de données.

Par contre, un site posant problème reste de façon indéfinie dans la base de données. Lors de cette routine, l'exactitude de l'adresse IP stockée est vérifiée et le hash du code HTML est régénéré.

SCREEN

Le nettoyage s'effectue à heure fixe tous les jours. Pour éviter de perturber l'activité du site, j'utilise le module *node-toobusy* Ce module permet d'éviter d'effectuer ce nettoyage, potentiellement lourd en requête entrante et sortante, lorsqu'il y a un grand nombre de requêtes client.

Pour cela, il observe le retard qu'a la file d'évènement. Si ce retard est plus grand que 70 millisecondes, le nettoyage est reporté de 5 minutes.

En fonction des performances du serveur, cette valeur peut être abaissée ou augmentée pour coller au mieux avec la disponibilité souhaitée pour le service.

Il est également possible d'effectuer ce nettoyage manuellement depuis la partie Admin.

4.3.3. Contributions des utilisateurs à la base de données *servers*

Les utilisateurs ont la possibilités de soumettre un serveur DNS qui serait inconnu.
La page *Help* a été créée pour cela.

Le format des adresses IP est vérifié côté client avant la soumission du formulaire.
Le pays est tapé en utilisant un typeahead au lieu d'un menu déroulant.

Une fois le formulaire complété et la soumission faite, côté serveur, l'existence du serveur et vérifiée sur base des adresses IP.

Si le serveur est existant, l'utilisateur en est averti sinon le serveur est inséré dans la base de données et l'utilisateur est remercié.

Les serveurs insérés par les utilisateurs peuvent être modifiés, activés ou supprimés dans la partie Admin.

5. Problèmes rencontrés et conclusion

5.1. Problèmes rencontrés

J'ai commencé ce projet alors que je débutais à peine dans Node.js et je n'avais jamais fait de JavaScript. J'ai suivi quelques tutoriaux et ça m'a amené à prendre de mauvaises habitudes.
Les tutoriaux que j'ai suivis oubliaient tous un point important : la structure. Ce qui m'a conduit à produire un code in-maintenable.

C'est à ce moment-là que j'ai décidé de reprendre la structure depuis le début avec l'aide de Skeleton.

Les problèmes structurels passés et le développement réellement commencé, le plus gros obstacle que j'ai rencontré se pose toujours à l'heure actuelle et je ne l'explique toujours pas.

Pour pouvoir effectuer des requêtes à un serveur DNS précis, j'ai trouvé un module, native-dns, qui permet de combler un manque dans la librairie de Node.js. Celle-ci ne permet pas pour le moment d'effectuer ce type de requête paramétrable. Cela pourrait

venir dans un futur proche vu que le développeur du module en question travaille maintenant pour Joyent, la société auquel appartient Node.js.

Au démarrage du serveur, l'application est créée, les différents fichiers et modules importés. Sous Windows, aucun problème, le serveur se lance correctement et effectue parfaitement ce que je lui demande de faire avec ce module.
Sous Linux par contre, impossible de démarrer le serveur, une erreur se produit dans un sous-module de native-dns.

Développer sous Windows n'étant pas un problème en soi, c'est ce que je continue de faire. Le problème réel est pour l'hébergement du serveur. En effet, Heroku, AppFog, Nodejitsu sont autant de PaaS sous Linux. Donc impossible pour l'instant de lancer le serveur hosté.

5.2. L'avenir du projet

Ce projet est voué à perdurer, même si la neutralité du net devenait une réalité pourquoi pas mondiale, rêvons grand !
En attendant , il faut rester prudent et partir sur un principe de précaution.
Internet reste un outil complexe, manipulable de façon parfois très aisée.

Quand j'aurai résolu ce problème lié au module native-dns, je pourrai enfin le mettre en ligne. Le retour d'une base de données MongoDB devrait avoir également avoir lieu.

Il y a également des fonctionnalités que je n'ai pas eu le temps d'implémenter qui sont intéressantes à détailler de façon théorique.

5.2.1. Servir de proxy

Une fonctionnalité qui devra être ajoutée est la possibilité, pour un utilisateur bloqué, de se rendre sur le site qu'il désire en se servant de mon serveur comme proxy. Il faudra cependant se poser les questions de légalité inhérentes à cette fonctionnalité.

5.2.2. Améliorer les performances des tests

Une étape importante est le temps d'exécution des tests afin de minimiser le temps de réponse pour le client. En utilisant le module async, je vais pouvoir mieux contrôler l'exécution des fonctions lentes afin que l'attente du callback ne soit pas bloquante pour l'envoi de la réponse.

Le choix de ne pas avoir de JavaScript côté client pour le test est un choix qui n'aide pas pour les performances. Lorsque l'utilisateur à soumis une URL au test et pendant l'éventuel délai d'attente, il ne voit rien d'autre qu'une page blanche. A terme, il est envisageable de proposer aux clients dont le JavaScript est activé une interface plus dynamique.

Le test lancé, le client serait directement redirigé vers la page de résultat, mais le contenu de celle-ci serait ajouté au fur et à mesure que le serveur progresse dans les tests. Ceci est assez facile à mettre en place en utilisant jQuery et en demandant au serveur d'envoyer chaque résultat, même partiel, en JSON au client.

5.3. Conclusion

A COMPLETER

6. Références

- **Arte Tracks** Les hackers urbains partent à l'abordage des villes
  http://www.arte.tv/fr/les-hackers-urbains-partent-a-l-abordage-des-villes/6909544,CmC=6909554.html
- **The Pirate Bay Away From Keyboard**
  http://watch.tpbafk.tv/
- **Datalove**
  http://datalove.me/
- **SPF Justice Commission des jeux de hasard** Liste noire des sites de jeux illégaux
  http://www.gamingcommission.be/opencms/opencms/jhksweb_fr/gamingcommission/news/news_0001.html
- **RTBF Médias** La liste noire des sites web bloqués mise en cause
  http://www.rtbf.be/info/medias/detail_la-liste-noire-des-sites-web-bloques-mise-en-cause?id=8001587
- **Node.js**
  http://nodejs.org/
- **Wikipédia** Node.js
  http://fr.wikipedia.org/wiki/Node.js
- **Benchmark** JavaScript V8 vs. Java 7
  http://benchmarksgame.alioth.debian.org/u32/javascript.php
- **V8 JavaScript Engine**
  http://code.google.com/p/v8/
- **Express**
  http://expressjs.com/
- **Connect-assets**
  https://github.com/adunkman/connect-assets
- **Node-mysql**
  https://github.com/felixge/node-mysql
- **YouTube** Felix Geisendörfer: Faster than C? Parsing Node.js Streams!
  http://youtu.be/Kdwwvps4J9A
- **Request**
  https://github.com/mikeal/request
- **Marked**
  https://github.com/chjj/marked
- **Async**
  https://github.com/caolan/async
- **Underscore**
  http://underscorejs.org
- **MD5**
  https://github.com/pvorb/node-md5
- **Node-toobusy**
  https://github.com/lloyd/node-toobusy
- **CoffeeScript**
  http://coffeescript.org

- **Jade**
  http://jade-lang.com
- **Stylus**
  http://learnboost.github.io/stylus
- **Bootstrap**
  http://twitter.github.io/bootstrap
- **Flat-UI**
  http://designmodo.github.io/Flat-UI
- **jQuery**
  http://jquery.com
- **DataTables**
  http://www.datatables.net
- **Validate**
  https://github.com/engageinteractive/validate
- **Heroku**
  https://www.heroku.com
- **AlwaysData**
  https://www.alwaysdata.com
- **MySQL**
  http://www.mysql.fr
- **ClearDB**
  http://www.cleardb.com
- **MongoDB**
  http://www.mongodb.org
- **MongoLab**
  https://mongolab.com

A COMPLETER

7. Glossaire

The Pirate Bay :

Scalabilité :

Benchmark :

Middleware :

A COMPLETER