ECOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

EPFL

Scala IDE as an Ajax Web Application

Projet de Semestre

Stephane Ruchet
Eté 2007

Tables des matières

Introduction 3

Description 4

- Apache 5
- Smotor 6
- Blommersit 6

Défis 7

- Un interpréteur multiutilisateurs 7
- Try Scala! 14
- Compatibilité Cross-Browsers 15

Suggestions 16

Conclusions 17

Introduction

Le web 2.0, un terme à la mode mais qui ne fait pas l'unanimité. Pourtant, un des principes les plus importants de cette mode est l'utilisation du web en tant que plate-forme. On cherche à proposer des applications web possédant un niveau d'interactivité équivalent à un logiciel classique. C'est l'idée qui est repris pour ce projet : proposer un environnement de développement pour le langage de programmation Scala développé à l'EPFL.

A terme, cette « application » pourrait s'enrichir et proposer un environnement complet pour celle ou celui qui désire de découvrir ce langage. Un tel service propose bien des avantages mais évidemment ne pourra pas remplacer un environnement de développement logiciel comme Eclipse pour la programmation plus complexe. Le concept est bien de faciliter la première prise en main de ce langage et de découvrir ses atouts sans devoir installer quoique ce soit. Cette application sera bien évidemment accessible n'importe où une machine est connectée à Internet et permettra, je le souhaite, à quiconque qui veut promouvoir Scala de pouvoir rapidement faire une démonstration.

Des applications comme celle-ci existent déjà pour le langage Ruby par exemple. Try Ruby[1] est l'exemple parfait : il permet en quelques minutes de découvrir ce langage et propose un tutoriel qui va dans ce sens. L'utilisateur n'a pas à se soucier d'une installation quelconque, de l'utilisation précise d'un logiciel, etc… . Ce type d'application comme celle mentionnée montre néanmoins quelques limitations comme l'écriture sur une seule ligne, ne pas avoir un aperçu des variables ou méthodes définies, l'impossibilité de modifier une méthode préalablement écrite. Des imperfections que j'essaie de combler avec ma proposition de « Scala Web Shell ».

J'espère avoir apporté la première brique d'une application qui sera enrichie par la suite. La possibilité de sauvegarder le code pour la prochaine visite, la découverte de ce langage à travers d'un tutoriel, un éditeur plus riche pour l'écriture de code sont quelques unes des fonctions qui trouveraient assurément leur place au sein de cette application.

Ma principale motivation de ce projet était de me dire que cette application pourra être utilisée par la suite par l'équipe du LAMP (je le souhaite du moins). Avoir en tête qu'on travaille sur un projet qui peut avoir un débouché est excitant et j'espère que de nouvelles fonctionnalités le renforceront par la suite.

[1] http://tryruby.hobix.com/

Description

Le but de ce projet est de créer un environnement de développement pour Scala, accessible sur Internet. Dans un premier temps il a fallu déterminer ce qui se trouvait à ma disposition et ce qui ne l'était pas.

L'interpréteur Scala est bien évidemment à ma disposition. Après avoir compris son fonctionnement, j'ai adapté le processus de création d'un interpréteur. Un point important lors de cette phase était de toujours garder en tête qu'il fallait accéder à la librairie NSC de Scala au plus haut niveau afin de maximiser la compatibilité avec des versions futures de l'interpréteur. De plus, un certain nombre de paramètres sont inutiles pour la version web de l'interpréteur (les arguments lors du lancement de l'interpréteur par exemple) et je ne les ai donc pas pris en compte.

Ce shell Scala pose d'autres problèmes dont je donne qu'une brève description ici. Ceux-ci sont tous liés au fait que l'interpréteur ne tourne plus sur une machine personnelle avec une seule personne qui y accède, mais est accédé par plusieurs utilisateurs à travers un navigateur. L'interpréteur a logiquement été conçu pour qu'une seule personne y accède à la fois donc il ne se soucie pas des partages des ressources, des variables déclarées, etc… . Chaque utilisateur devra donc avoir « son » propre interpréteur. Il faudra donc « lier » un interpréteur à l'utilisateur grâce aux sessions. Dans le même domaine, un mécanisme lié aux sessions devra vérifier que la session est toujours active et dans le cas négatif fermera l'interpréteur correspondant. Pour mieux comprendre, sur un bureau, lorsque nous voulons fermer une application nous procédons à sa fermeture, mais sur internet, un utilisateur peut simplement fermer son navigateur et il faudra bien que l'interpréteur qui y correspond soit aussi fermé. La sortie de l'interpréteur (de type « printwriter ») est statique alors qu'à chaque requête de l'utilisateur, le « chemin de retour » change et donc il faudra lier la sortie statique de l'interpréteur à l'entrée dynamique de la requête faite par l'utilisateur.

Finalement, l'interpréteur retourne le résultat et l'affiche dans la console. Nous désirons ajouter des fonctionnalités côté client et il faudra donc ajouter de l'information aux données qui arrivent de l'interpréteur afin de traiter ces données côté client. Un parseur sera donc utilisé pour enrichir la sortie de l'interpréteur.

Après une courte description des principaux défis de ce projet, une petite présentation des différentes architectures proposées est donnée.

Cette étape, bien qu'elle soit importante, n'est pas très compliquée en soit mais j'ai perdu énormément de temps à avoir les idées au claires et il m'est difficile d'expliquer les raisons de ce temps perdu. C'était la premières fois où je me retrouvais vraiment seul dans un projet et c'est une situation où je n'étais pas familier. Il m'a fallu du temps pour me mettre en route.

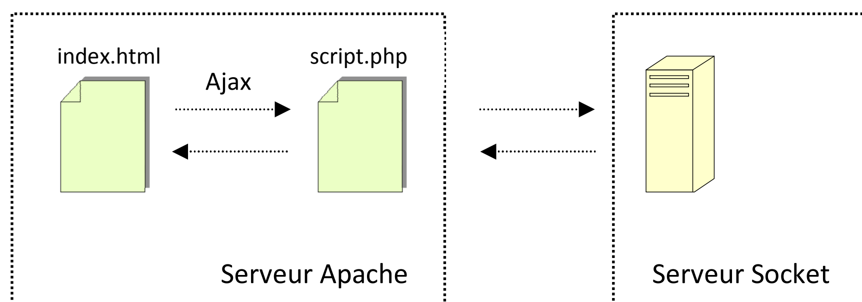
J'ai répertorié quatre différentes possibilités dont deux se basent sur le même principe, à savoir l'utilisation d'un Apache Http Server. La 3ème possibilité était l'utilisation de Smotor, un moteur de servlet Scala, développé par un membre du LAMP. Finalement la quatrième solution, celle choisie, était l'utilisation d'une librairie Http Server écrite par Rick Blommers[2].

[2] http://www.blommers-it.nl/libhttpd/home.php

Apache

Nul besoin d'expliquer en détail ce qu'est le projet Apache. Apache est un Serveur de pages web open source extensible (via des modules) et conforme aux protocoles standards, notamment HTTP 1.1. Sa robustesse et ses performances expliquent entre autres pourquoi plus de 70% des serveurs utilisent ce serveur. J'ai donc bien évidemment commencé par étudier mes possibilités avec Apache.

Mon idée de départ était de créer mon propre serveur socket en Scala. Il écouterait sur un port et lorsqu'une requête aboutissait, il la traite puis retourne la réponse. Les fichiers MultiServer.scala et MultiServerThread.scala que j'ai crée sont un exemple de cette idée. Toute la gestion du site à proprement parler aurait été faite par un serveur Apache et une requête Ajax aurait fait le lien entre ces deux « serveurs ». Si cette solution n'est pas très élégante, elle avait l'avantage à mes yeux de se baser sur le serveur Apache pour toute la gestion du site et que dans ce cas, l'utilisation d'une base de données (MySQL par exemple) ou encore l'utilisation de sessions à travers le langage de programmation PHP par exemple aurait été beaucoup plus rapide. J'ai rapidement découvert qu'une protection de la majorité des navigateurs allait compliquer la tâche. Afin de se protéger contre des attaques, les navigateurs empêchent un code JavaScript qui contient un XMLHttpRequest de communiquer avec des serveurs se situant sur un autre domaine. Si la page qui contient le XMLHttpRequest est chargée depuis le site monSite.com, l'XMLHttpRequest ne pourra que se connecter sur monSite.com et aucun autre. Toujours convaincu que cette solution était intéressante, j'ai écrit un petit script PHP [3] qui aurait fait le lien entre les deux serveurs :



C'est une solution qui a le mérite de fonctionner mais elle n'est pas très élégante : deux serveurs sont nécessaires, un fichier PHP de quelques lignes entre les deux, etc.

La deuxième solution aurait été d'avoir Apache avec Tomcat comme unique serveur en utilisant des servlets par exemple. C'est une solution tout à fait possible mais je ne l'ai pas étudié trop en profondeur. Pour ce projet, je n'avais besoin que d'une librairie qui me permette de bien évidemment gérer les requêtes et réponses, qu'il soit multithread, et finalement qu'il puisse gérer les sessions. Après ce constat, j'ai commencé à écrire mon propre serveur HTTP et j'ai implémenté la méthode GET [4] par exemple. L'implémentation de la méthode POST fut déjà plus délicate et j'avais décidé d'arrêter à ce moment là. Finalement, un serveur Apache me semblait trop lourd à mettre en

[3] script.php
[4] httpRequestHandler.scala

place pour les besoins que j'avais, et que créer sa propre libraire Serveur Http aurait été le sujet d'un autre projet de semestre.

Smotor

Un membre de l'équipe du LAMP a développé un moteur de Servlet Scala au doux nom de Smotor. N'ayant pas réussi à le faire fonctionner, on m'a déconseillé de l'utiliser pour la simple et bonne raison qu'il n'était pas encore tout à fait stable. Ayant comme unique documentation l'API de Smotor, j'ai préféré me tourner vers une autre solution.

Blommersit

Après une courte recherche sur Google, je suis tombé sur le site de Rick Blommers. Cette personne propose une librairie HTTP Server complète. Le projet est momentanément abandonné mais l'auteur de cette libraire projette de s'y remettre et de le proposer sur Sourceforge d'ici l'année prochaine.

Cette petite librairie de 300 Ko offre toutes les fonctions que j'avais besoin. J'ai utilisé pour ce projet une version non disponible « officiellement » (la version 1.03). C'est Mr. Blommers lui-même qui me l'a transmise après que j'ai détecté deux bugs et aider à la correction de ceux-ci (problème de performance lié au Keep-Alive URLs et un autre problème lié à son parseur d'URL). J'y ai apporté quelques modifications pour la gestion des sessions. Je me suis demandé si les performances seraient à la hauteur et j'ai demandé à Mr. Blommers ce qu'il en pensait par rapport à Apache. Il a répondu qu'on pouvait logiquement admettre qu'Apache soit plus rapide, mais mes craintes ont disparu lorsque j'ai constaté qu'une requête d'un client (envoi au server, traitement, retour au client) prenait entre 250 et 400ms.

Côté client, je me suis renseigné sur quelques Framework graphique comme Spry proposé par Adobe[5]. Bien que celui-ci soit très réussi, au fil du projet, l'utilisation d'une telle librairie ne se révélait pas nécessaire. La seule animation graphique que je propose est une simple « bulle » ne nécessitant pas d'effets graphiques particuliers. J'ai mis un accent particulier sur l'interface afin de le rendre le plus élégant que possible. Cela n'engage que moi, mais la première impression est la plus importante et j'ai travaillé dans ce sens en donnant à cette application une interface aussi réussie que possible.

[5] http://labs.adobe.com/technologies/spry/

Défis

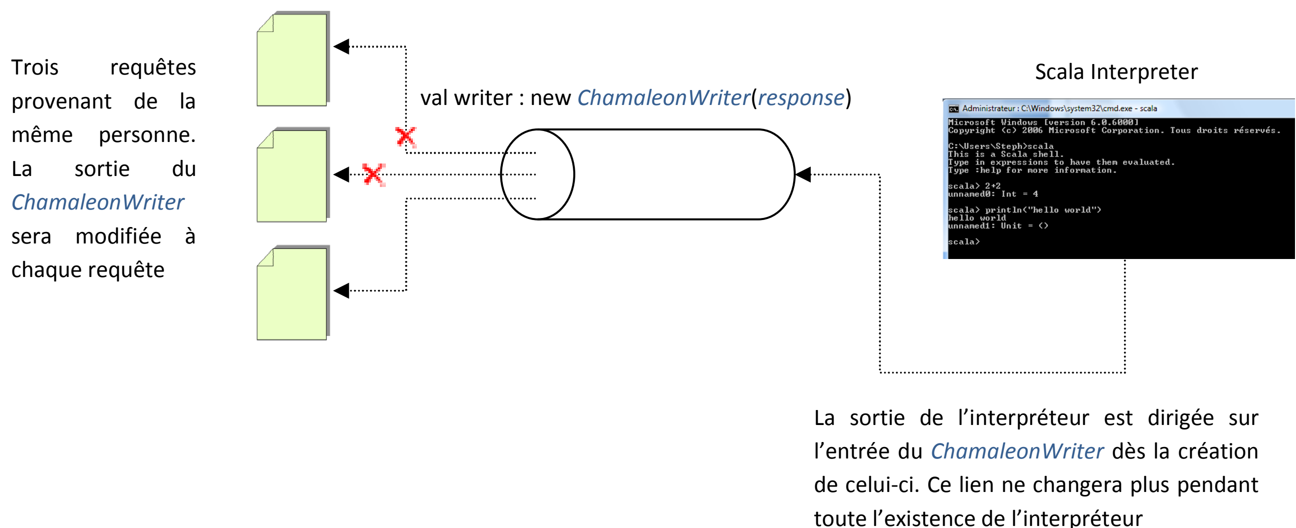
Brièvement exposé précédemment, je vais parler plus en détail d'un certain nombre de défis clés de ce projet et quelles ont été les solutions que j'ai implémentées.

Un interpréteur multiutilisateurs

Une application Internet se veut par définition être accessible par plusieurs utilisateurs simultanément. Plusieurs utilisateurs signifient plusieurs interpréteurs et un lien doit donc exister entre l'interpréteur et la session de l'utilisateur afin qu'il puisse utiliser une variable préalablement déclarée par exemple. Une autre raison évidente est l'utilisation des ressources mémoires : si les interpréteurs n'étaient pas fermés au bout d'un certain temps d'inactivité (la durée d'une session justement), nous consommerons rapidement toute la mémoire vive disponible. Il faut un mécanisme qui crée un interpréteur lors de la première requête, puis à chaque nouvelle requête (pour autant que la session n'ait pas expirée), que le même interpréteur soit utilisé et lors d'une inactivité d'une durée d'une session, que l'interpréteur soit fermé.

Pour gérer les sessions, j'ai modifié le fichier HttpSession.java et crée le fichier HttpSessionExpiryListener.java (qui contient la classe abstraite *HttpSessionExpiryListener*) dans la librairie libHttp. Le principe est plutôt simple : lors de la première requête, on ajoute à l'instance de la session actuelle l' « *expiryListeners* » du serveur qui est de type *HttpSessionExpiryListener*. Un « timer » débute dont le temps est égale à une session. Lors des requêtes suivantes, le timer est réinitialisé. Lorsqu'il est arrivé à terme, la méthode *ExpiryNotifier* (toujours dans HttpSession.java) est appelée. Celle-ci est plus « généraliste » qu'elle ne devrait pour mon utilisation exacte, mais j'ai préférée la rendre ainsi au cas où nous en aurions besoin (mais ce n'était pas nécessaire finalement). *ExpiryNotifier* va appeler la méthode *sessionExpired* (se trouvant dans le fichier server.scala) avec comme paramètre la session qui a expiré. La suite est logique : fermer l'interpréteur qui y correspond, enlever la clé de la hashmap *mapSession* et enlever le listener qui ne sert plus à rien. La boucle est fermée côté serveur, mais comment annoncer au client que sa session a expiré ? A moins d'utiliser des servlets, il n'est pas possible d'envoyer des données du serveur au client sans que lui s'attende à en recevoir. Le serveur ne peut donc pas l'avertir que l'interpréteur a été fermé et sa session détruite. La feinte est d'avoir un timer aussi côté client dont la valeur est la même que le timer côté serveur (fixé à 15 minutes). A chaque fois que le client envoie une requête au serveur (et uniquement dans ce cas) le timer est remis à zéro côté client (et côté serveur aussi). Il y a malheureusement un petit décalage entre les deux timers : la durée que prends la requête pour aller du client au serveur. Mais nous pouvons raisonnablement admettre que même quelques secondes sur 15 minutes restent négligeables.

Quand une personne envoie une requête, une nouvelle instance de la classe *InterpretCommand* est créée et un appel à la méthode *handleHttpRequest* est faite. Les paramètres à ma disposition sont l'objet *serveur* (identifie le serveur qui est utilisée), le *serverThread* (nécessaire pour les sessions), le paramètre *request* (le code qui a été envoyé par l'utilisateur) et finalement le *response* qui, pour visualiser, est le chemin de retour sur lequel on va retourner les données. Puisqu'une nouvelle instance de la classe *InterpretCommand* est créée à chaque requête, le chemin de retour n'est pas le même alors que la sortie de l'interpréteur ne change pas. Un mapping sera donc nécessaire lors de la réception de chaque requête. C'est la classe *ChamaleonWriter* qui va s'en charger :



Voici quelques lignes qui démontrent ce processus de « mapping » (du code a été enlevé afin de rendre cet exemple plus clair) :

```
def createInterpreterLoop { …
      Server.mapSession.update
            (st.getHttpSession(), new Pair(myInterpreterLoop, writer));
… }

val myInterpreterLoop: InterpreterLoop =
      Server.mapSession.get(st.getHttpSession()) match {
            case None =>
                  st.getHttpSession().addHttpSessionExpiryListener
                        (Server.expiryListener)
                 createInterpreterLoop(st, s, response)

            case Some(Pair(x, y)) =>      y.forward = response;
                                          x
      }
```

La hashmap *mapSession* a comme clé les sessions et comme valeurs une pair [*InterpreterLoop*, *ChamaleonWriter*]. On vérifie si la clé est dans la hashmap, si c'est le cas, cela veut dire que l'interpréteur existe déjà : on modifie la sortie du *ChamaleonWriter* à la nouvelle *response* puis en retourne le bon interpréteur. Si la clé n'existe pas, il faut créer l'interpréteur qui lui mettra à jour la

hashmap avec le bon *writer* (de type *ChamaleonWriter*) et appellera la méthode *addHttpSessionExpiryListener* de la session courante.

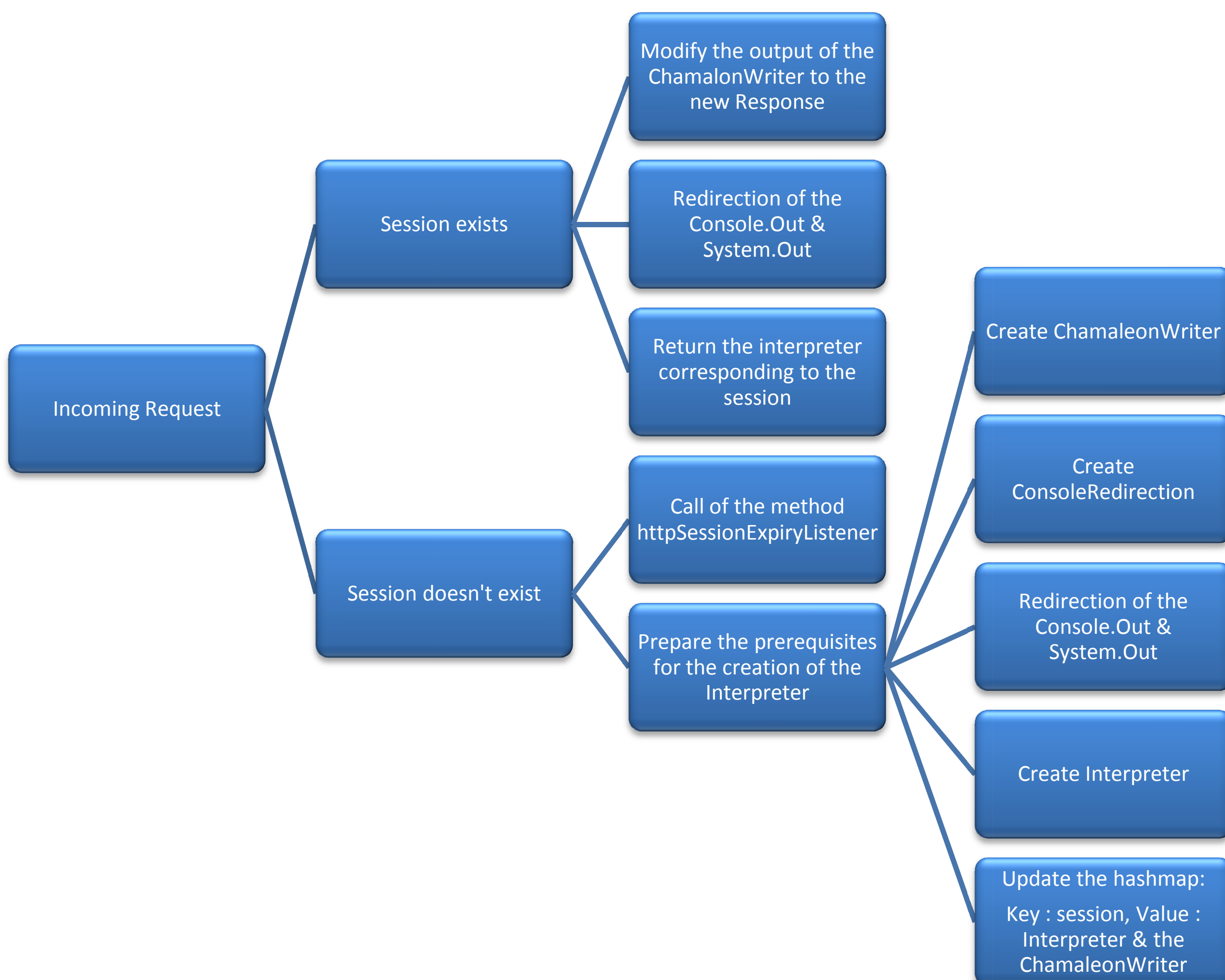
Le *ChamelonWriter* sera aussi utilisé pour la redirection du Console.Out et du System.Out.

```
val writer = new ChameleonWriter(response)
val redirection = new ConsoleRedirection(writer)

scala.Console.setOut(redirection)
System.setOut(new PrintStream(redirection))
```

Un peu de “wrapping” fut nécessaire pour pouvoir faire les redirections nécessaires. Notez que j’ai dû étendre la classe httpResponse définie dans la librairie Java à la classe Printwriter.

Maintenant qu’un aperçu a été donné concernant la création et la gestion de l’interpréteur, voici le déroulement lorsqu’une requête arrive :



A la fin de ce processus, nous avons un interpréteur (si si !), la sortie de l'interpréteur mappée sur le « *response* » de la requête et finalement les redirections de la Console et du Système. Nous pouvons maintenant traiter la requête.

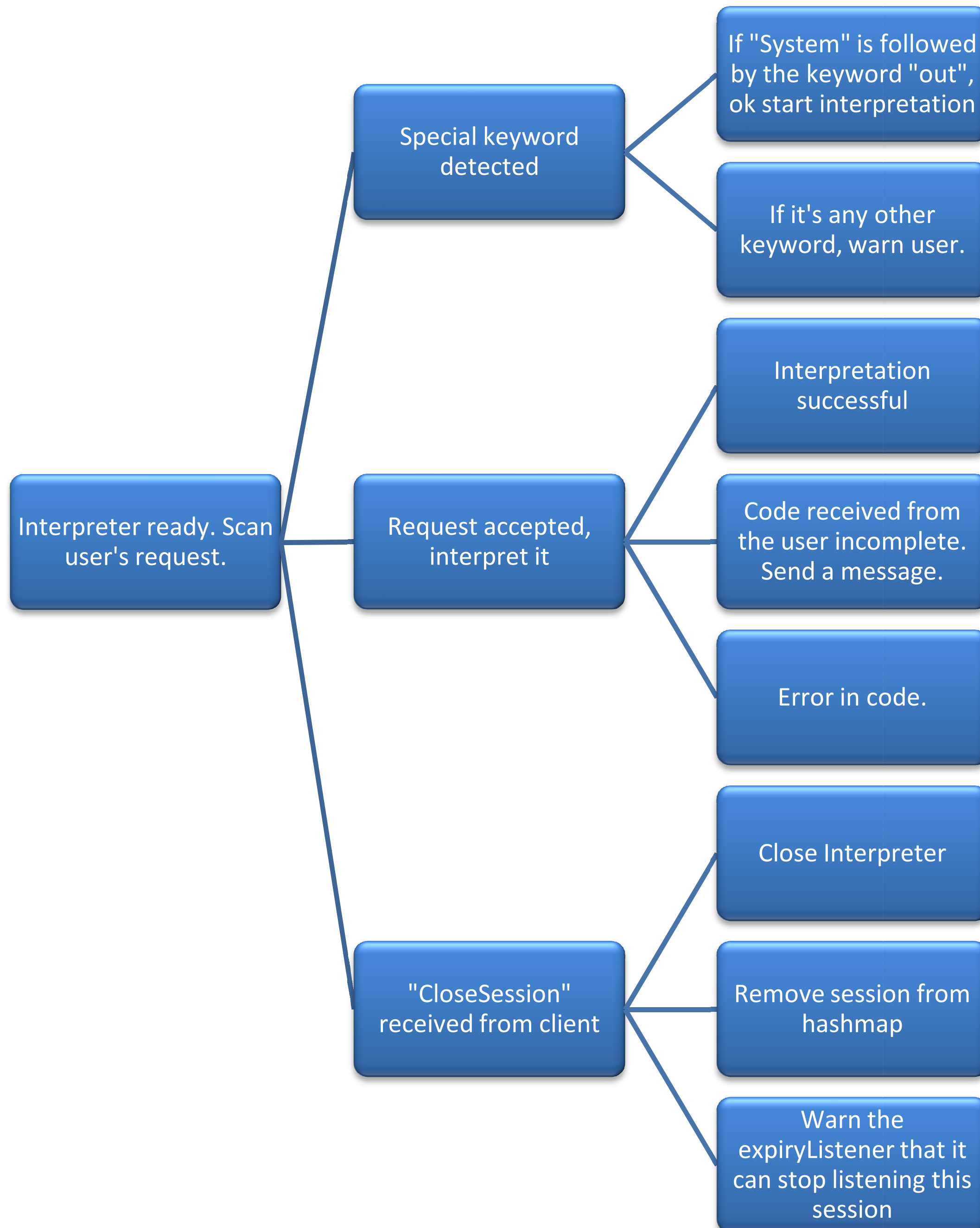
Comme mentionné au début de ce rapport, le but était d'utiliser la librairie NSC de Scala au plus haut niveau, afin que les modifications futures de cette librairie perturbent le moins possible le bon fonctionnement de la version web de l'interpréteur. Tout se passe principalement dans le fichier *InterpreterLoop.scala*. Jusqu'à récemment, j'utilisais la méthode *command* pour interpréter la requête utilisateur. Mais lorsqu'un utilisateur entrait une ligne syntaxiquement correct, mais non terminale (par exemple : def A(a : Int) =), l'interpréteur s'attends à ce que la « suite » arrive et donc dans ce cas, ne « flushait » pas le writer et donc côté client on attendait une réponse du serveur, côté serveur, il attendait la requête suivante : on avait un deadlock. J'ai donc du « descendre » à un niveau plus bas et utiliser la méthode *interpretStartingWith*. En réalité, je n'utilise pas cette méthode mais je l'ai copiée et modifiée un peu pour qu'elle corresponde à mes besoins.

C'est lors du cas « Incomplete » que la suite change par rapport à l'interpréteur de base. Ici, si nous sommes dans ce cas, il retourne un message au client qui traitera ce cas et ne fera plus rien, comme lors d'une erreur (None).

La question qui s'est posée ensuite était de savoir, en dehors de l'utilisation « normale » du serveur (l'interpréteur), quelles autres fonctionnalités cette application allait proposer. Mon raisonnement était le suivant : sur Internet, la mode aujourd'hui est d'économiser un maximum de bande passante entre le client et le serveur, et donc de minimiser l'interaction entre les deux. En d'autre termes, j'ai souhaité que le côté client fasse un maximum de tâches (filtrage de requêtes, stockage du code entré par l'utilisateur, stockage des noms des variables, définitions, etc…) et que le serveur ne soit utilisé que pour la partie « traitement » de la requête. Puisque le code côté client est accessible à qui le souhaite (bien que la compréhension de celui-ci soit une autre chose), cela pose néanmoins l'inconvénient qu'un utilisateur peut « voir » comment fonctionne l'envoi de requêtes et la réception des données. A ma défense, il ne me semble pas que cela ait un impact sur la sécurité. Le travail qui est fait côté client aurait très bien pu être fait côté serveur et je pense même qu'il aurait été plus simple de l'implémenter de cette façon. C'est un choix. Nous verrons néanmoins par la suite que le serveur propose tout de même une autre fonctionnalité. Finalement, il y a deux cas spéciaux que je traite : la demande de l'utilisateur de fermer la session (afin de commencer un nouvel interpréteur par exemple) et le filtrage de certains mots clés afin d'augmenter un peu la sécurité.

Je profite de ce dernier point pour parler d'un « petit » problème de sécurité : à part quelques filtrages basiques (côté client et côté serveur) que j'effectue afin d'éviter une utilisation abusive de l'interpréteur web, un utilisateur peut entrer du code qui permettrait par exemple d'écrire dans un fichier texte se trouvant sur la machine qui héberge le serveur (j'en ai fait l'expérience). Afin d'améliorer la sécurité, une possibilité serait d'utiliser une machine virtuelle qui empêche le serveur d'accéder à des ressources qui pourrait nuire au fonctionnement normal du serveur.

Voici le schéma donnant une vue globale des étapes du traitement d'une requête :



Une précision s'impose lorsque l'interpréteur « interprète » la requête. Dans le cas où la requête est acceptée ou qu'il y a une erreur, c'est bien l'interpréteur qui écrit directement dans le *response*.

La requête a maintenant été traitée, l'interpréteur va donc appeler le *ChamaleonWriter* pour qu'il écrive dans le *response*. C'est donc ici que je vais parser l'array de caractère reçu de l'interpréteur. Cette étape est cruciale et est difficile à mettre en place. L'interpréteur de base a été conçu pour retourner un array de caractère dénué de toute autre information supplémentaire. Si du côté client, je souhaite isoler les variables, définitions ou qu'un utilisateur puisse éditer une méthode précédemment entrée par exemple, le parseur devra retourner non plus un array de caractère, mais toute une structure qui permettra, côté client, de reconnaître dans quelle situation l'application se trouve.

Puisque c'est une application utilisant la technologie Ajax pour traiter les requêtes, le choix du format de cette structure est évident et donc repose sur le langage XML. Pourquoi n'avoir pas modifié à la base la sortie de l'interpréteur ? Pour la même raison lors de la création de celui-ci, j'ai voulu rester au niveau le plus élevé possible afin que si des modifications internes sont faites par la suite, l'interpréteur web pourra en profiter. Bien évidemment, le temps qui aura été nécessaire pour modifier en interne l'interpréteur ne m'aurait pas permis de terminer mon projet à temps.

Il a fallu imaginer et trouver toutes les syntaxes différentes que l'interpréteur pouvait fournir au *ChamaleonWriter*. Si certaines syntaxes étaient évidentes, d'autres l'étaient moins. Ainsi, si l'utilisateur a entré un code incorrect, il y a 3 sorties différents venant de l'interpréteur. Soit le String (je convertis l'array de caractère en string) commençait avec le token « <console> », soit avec « error » ou soit finalement avec le token « Warning ». Lorsque l'utilisateur déclare une variable mais sans lui attribuer un nom (simplement « 2 » et l'interpréteur retourne « unnamedX : … »), le premier terme du string varie.

La méthode nextToken est particulière, car elle ne retourne pas forcément le prochain token comme nous l'avons vu lors du cours de compilation. En étudiant les différentes syntaxes arrivant de l'interpréteur, certains caractères (le « { », le « (», etc…) n'ont pas été considéré comme des « séparateurs » de token comme l'espace ou le saut à la ligne. Ceci a surtout été nécessaire afin d'obtenir le type d'une définition :

```
def foo[T] (x : T) = x
```

L'interpréteur va retourner l' « array » suivant : `foo: [T >: ? <: ?](T)T` . Le mot « `foo` » représente le nom que nous avons donné à cette définition. La syntaxe habituelle est que le nom soit suivit du double point. Mais ce qui suit constitue tout son type, donc nextToken devra retourner `[T >: ? <: ?](T)T`. Un petit souci ici est que le double point est un token « séparateur » et fallait le prendre en compte.

Il a aussi été nécessaire de prendre en compte que plusieurs variables ou méthodes soient déclarées en même temps (var a, b = 5 par exemple). L'interpréteur sépare chaque déclaration par des sauts à la ligne : arrivé au token « 5 », si le parseur est arrivé à la fin de la ligne il recommence à la ligne suivante.

Lors d'une erreur, la console affiche un message expliquant l'erreur, suivit du code entré par l'utilisateur puis finalement le « chapeau » afin d'indiquer la position de l'erreur. Lors de mes essais, j'ai remarqué que l'interpréteur ne retournait pas tout ceci en une seule fois, mais en trois fois ! Si j'appliquais le même « parsing » que jusqu'à ici, côté client on aurait seulement reçu le message expliquant l'erreur. L'explication est la même que pour le problème de la synchronisation des sessions entre le client et le serveur : le serveur ne peux pas envoyer des données au client si celui-ci n'est pas « prêt » pour les recevoir. Quand le client envoie sa requête Ajax, il attends une réponse du serveur et dès qu'il l'a reçu, son état est « modifié » est une fonction traite les données reçues (sendReq.onreadystatechange = function() …). Lorsque le serveur envoie le code erroné, puis lors d'un troisième envoi, le « chapeau », le client n'est pas en « état » pour recevoir quoique ce soit. L'idée est la suivante : lorsque le parseur a détecté l'erreur dans le premier retour de l'interpréteur, il prépare la *response* et la stocke dans un buffer et avertit le *ChamaleonWriter* qu'il attend encore des données (keepParsing = true). Lors du troisième retour, on sait qu'il n'est composé que du « chapeau » et donc il suffit de compter le nombre d'espace avant celui-ci et on connaît la position où l'erreur se trouve. A ce moment là, on n'attend plus rien de l'interpréteur et on peut annoncer au *ChamaleonWriter* qu'il peut envoyer les données au client. Les autres défis se limitaient à mettre en place un parseur qui ne plante pas (encore heureux) : si une syntaxe à laquelle je n'ai pas pensé devait passer dans le parseur, il devrait retourner sous la forme d'un « <unknownsyntax> ».

Pour la redirection de la fonction println (Console.println ou System.out.println), j'ai remarqué que l'interpréteur retourne à chaque fois le string (qui se trouve entre les deux paranthèses) imprimé suivi de « unnamedXX: Unit = () » et j'utilise cette observation pour mettre en place l'action adéquate dans le ChamaleonWriter : lorsque *ConsoleRedirection* fait un appel à la méthode *write*, il appelle la méthode *write* du *ChamaleonWriter* mais avec un paramètre de type Boolean supplémentaire. Ce paramètre indique que l'écriture vient de la « console » et non de l'interpréteur. Il stocke les données dans un buffer puis un appel au *write* « conventionnel » du ChamaleonWriter (« unnamedXX: Unit = () ») est faite : le parseur débute avec un buffer non vide. La suite est logique : il retourne le contenu du buffer et la variable unnamedXX au client.

Par la suite, si quelqu'un désire imprimer des données côté serveur pour débugger par exemple, j'ai créé la classe *Logger* qui écrira dans un fichier les données qu'on lui demandera d'écrire. Attention à ne pas mettre un System.out.println dans la méthode *write* du *ChamaleonWriter* : vous aurez une magnifique boucle infinie !

Finalement il ne reste plus que la classe MultilineConsole à analyser. Je vais d'abord expliquer pour qu'elle raison elle a été crée. Comme nous verrons plus loin, l'utilisateur peut éditer une définition qu'il a déjà tapée par exemple. Lorsque l'utilisateur demande à éditer une définition, un pop-up va s'ouvrir. Ce pop-up contient le textarea dans lequel le code se trouvera. Au départ, je voulais qu'une fonction JavaScript se servant de l'élément DOM getElementById écrive dans le textarea. Si cela fonctionnait très bien dans IE 7, cela ne marchait pas dans Firefox (une fois n'est pas coutume). L'idée suivante était de récrire une fonction JavaScript dans lequel tout le code de la fenêtre pop-up s'y trouverait (insert.html) et que lors de son appel, il ajouterait le code puis ensuite écrirait le tout dans le pop-up. Solution qui fonctionne mais qui n'est vraiment pas très élégante : si une modification est apportée dans le fichier insert.html, celle-ci n'est pas faite dans cette fonction

puisque le code a été recopié. De plus, les performances n'étaient pas à la hauteur. Troisième solution : la classe MultilineConsole. Côté client on envoie une requête Ajax au serveur avec le code qu'il faudra insérer dans le textearea. Le serveur lit le fichier insert.html (donc les modifications sont pris en compte si nous modifions ce fichier), ajoute le code, puis retourne le tout au client qui lui ensuite ouvrira le pop-up et écrira le code dedans. Cette façon démontre bien que nous pouvons faire les choses soit du côté client, soit du côté serveur. J'ai toujours cherché la solution la plus performante et cette fois, c'est du côté serveur que les résultats étaient les meilleurs.

Try Scala!

Le développement côté client était à mes yeux une étape cruciale : si je voulais que cette application soit vraiment fonctionnelle, celle-ci devait être rapide, efficace et agréable à utiliser. J'ai sous-estimé le travail à faire et j'ai dû malheureusement abandonner certaines fonctionnalités que je voulais implémenter (ce sera pour la v2).

Une fois l'interface mise en place, la même question revenait toujours : quelles fonctionnalités implémenter afin de rendre l'utilisation de cette Web Console aussi agréable que possible ? Voici une liste non-exhaustive :

- Possibilité d'effacer juste la console
- Touches flèches montantes & descendantes afin de naviguer dans l'historique
- Montrer l'historique de l'utilisateur
- Si une ligne est incomplète (mais syntaxiquement correct), proposer l'ouverture du pop-up avec le code dedans
- Affichage du code d'une définition ou d'un type (class, trait, etc…)
- Possibilité d'éditer les définitions et les différents types
- Quelques filtrages de mots clés
- Pouvoir éditer plusieurs lignes de code

C'est sûrement ce dernier point qui a été le plus difficile à mettre en place. A partir du moment que je proposais cette fonction de « multiconsole », de nombreux défis se sont manifestés : que se passe-t-il lorsque un utilisateur entre deux définitions à la suite (ce n'est pas le même cas quand une définition se trouve à l'intérieur d'une autre) car il n'existe aucun moyen côté serveur pour séparer le code de la première définition de la deuxième si on souhaite qu'ensuite l'affichage du code d'une définition soit correct. Que se passe-t-il lorsqu'un utilisateur mélange définitions et variables ou utilise des syntaxes plus exotiques comme « var a , b = 5 ». Il existe plusieurs façons d'écrire le même code (une définition : tout sur une ligne, utilisation des crochets {}, le signe « = », ...) qui sont l'une des richesses du langage de programmation Scala mais qui complique logiquement la tâche de séparer correctement le code. J'ai découvert un comportement étrange de l'interpréteur Scala qui a ajouté du piment dans mon code : supposons que nous déclarions une variable A, suivit d'une définition B, suivit d'une variable C. Je m'attendais à ce que l'interpréteur traite les données dans le

même ordre qu'il les a reçu, mais en réalité la variable C sera traitée avant la définition B ! D'autres défis s'ajoutent à cette liste comme lorsqu'un utilisateur entre une variable A, mais ensuite crée une classe avec le même nom ! Il faut s'attendre aussi à ce que l'utilisateur ajoute des sauts à lignes ou des espaces blancs qui compliquent la tâche du « parseur » côté client. Finalement, si celui-ci n'arrivent pas à gérer le code qu'on lui donne, un message avertira l'utilisateur afin qu'il puisse continuer en modifiant son code.

Compatibilité Cross-Browsers

Ce fut un défit de taille que de rendre l'interpréteur compatible avec les 4 navigateurs suivants : Internet Explorer 7, Firefox 2.x, Opera 9.x et Safari (pour satisfaire quelques utilisateurs du LAMP). Une grosse erreur que je ne ferais plus jamais : coder d'importantes portions de code sans vérifier directement avec les quatre navigateurs cités ci-dessus. Tout le développement a été fait sur Firefox et quelle joie cela a été quand j'ai découvert que plus rien ne fonctionnait sur les autres navigateurs. Il n'y a pas vraiment de méthode pour rapidement trouver la panne à part revenir en arrière ou en isolant le code. Je n'énumérerais pas tous les problèmes que j'ai rencontrés mais je vais commenter les plus importants.

Comme vous pouvez le lire ici[6], Internet explorer ne traite pas les nœuds textes vides comme Firefox les traite. On ne sait pas pourquoi c'est ainsi, mais c'est un fait. Une fois que ce problème connu et qu'une astuce existe (car ce n'est pas toujours le cas), c'est vite réglé. C'est lorsqu'on n'est pas au courant qu'on s'arrache les cheveux. Un autre problème fut l'utilisation de la méthode insertAdjacentHTML avec Internet Explorer qui ne réagissait pas du tout comme il devrait. Pour contourner ce problème, j'ai trouvé un script sur Internet (ie_methods.js) qui m'assurait la compatibilité avec IE. Puisque je parle de script, j'ai aussi utilisé le script prototype.js qui est très connu chez les développeurs Web. Mais je n'utilise même pas 10% de cette librairie est donc une amélioration serait de prendre (ou de créer) seulement les fonctions qui sont vraiment nécessaire et de plus avoir besoin d'un libraire JavaScript de 75Ko. Un autre bug très pénible à trouver était le bouton « Send Code » dans le MultilineConsole. Pourtant c'est le même principe que sur la page principale. J'ai trouvé finalement que le « hack » href=''#'' était le fautif et en regardant sur le W3C le href n'est pas obligatoire et je l'ai donc enlevé. Si cette modification a réglé le problème sous Opera, il reste un problème avec IE : le premier « Send Code » affiche un « wait » dans la console principale alors que les envois suivants marchent sans problème ! Il a suffit que je change ce bouton pour le « button » normal et tout est entré dans l'ordre !

J'ai fait valider le site en HTML Transitional et en CSS afin de maximiser les chances qu'il soit compatible avec les navigateurs. Si je n'observe plus de problème avec IE et Firefox, un bug d'affichage des boutons se produit avec Opera : en effet, il faut rafraichir 3 à 4 fois la première page afin que les boutons s'affichent correctement et je n'ai pas trouvé la raison de ce problème. Un dernier « bug » avec Safari : par défaut Safari ajoute une bordure bleue dans les champs lorsqu'ils

[6] http://www.w3schools.com/dom/dom_mozilla_vs_ie.asp

sont activés. Des sites prétendent qu'il est possible de surpasser cette fonctionnalité mais je n'ai pas réussi à l'implémenter.

C'est dommage que le support d'Internet Explorer dans une version inférieure à la 7 ne soit possible. Heureusement avec le temps ces versions devraient disparaître…

Suggestions

J'ai mentionné au début de ce rapport que le site de Try Ruby me plaisait bien. Il est bien fait et est agréable à utiliser. Sans trop m'avancer, je pense que j'en ai fait de même avec Try Scala. Mais sur le site de Try Ruby, il propose une fonctionnalité fort sympathique : un tutoriel permettant à l'utilisateur, en 15 minutes, de se faire la main sur Ruby. Non seulement le tutoriel est bien fait (avancement automatique des pages lorsque l'utilisateur entre le bon code), mais c'est une fonctionnalité bien pratique car il faut avouer que la personne qui ne connaît pas Scala ne pourra pas immédiatement découvrir ce magnifique langage à moins de tâtonner.

Une autre fonctionnalité qui aurait un intérêt serait de stocker le code que l'utilisateur souhaiterait garder pour une utilisation ultérieure. Cela sous-entend donc l'utilisation d'une base de données avec code d'identification. JDBaccess[7]aurait pu être utilisé avec MySQL ou encore JDBC[8] avec MySQL.

Le multiconsole peut encore bien évoluer. J'ai cherché une façon simple de coloration syntaxique « à la volée » dans le textarea mais ce n'était pas concluant. C'est sûrement possible avec du JavaScript. On pourrait imaginer ensuite lors de chaque frappe que des mots clés soit proposés, ou des boutons qui écriraient la syntaxe pour la création d'une classe (comme le fait Eclipse par exemple). Il n'y a pas vraiment de limite à part l'imagination. Le point que je regrette le plus peut-être est que je n'ai pas eu le temps de gérer plusieurs onglets dans le MultilineConsole.

L'historique est peut-être le seul point que je n'ai pas eu le temps de bien finir. On aurait pu imaginer également la coloration syntaxique, un bouton « copier vers presse-papier », un bouton pour imprimer, une meilleure interface.

[7] http://jdbaccess.com/
[8] http://java.sun.com/javase/technologies/database/

Conclusions

Lorsque j'ai postulé pour ce projet, je savais que cela serait un défi pour moi et je n'ai pas été déçu. Je le souhaitais ainsi pour que je progresse plus rapidement. J'ai d'ailleurs demandé de faire ce projet seul. Si les débuts ont été difficiles, j'ai pu, avec l'aide & la patience de Gilles Dubochet (que je remercie au passage) surmonter les problèmes et au fil du temps me sentir plus à l'aise dans ce projet. Les projets à l'EPFL se font généralement à plusieurs et dès qu'un problème surgit, on demande au copain d'à côté d'intervenir. Ici ce n'était pas le cas et c'était autant plus difficile qu'il fallait gérer tous les aspects de cette application.

Ce projet m'a permis de mieux comprendre la programmation orienté-objet, de découvrir le fonctionnement des EventsListeners, d'utiliser les writers ainsi que toutes ses variantes, d'imaginer comment la vie doit être difficile pour les webmasters de site complexe et finalement de réellement avoir du plaisir à me battre avec le JavaScript. Au fil des semaines, j'ai remarqué que mon code était souvent trop « stricte » et que je n'envisageais pas toutes les possibilités. C'est un défaut que j'ai cherché à améliorer en proposant une console qui envisageait toutes les possibilités, bien que je sois conscient que des améliorations pourraient encore être faites.

Cette Web IDE pour Scala que j'ai développé trouvera, je le souhaite, une place sur le site officiel de Scala ou du moins qu'elle soit reprise par quelqu'un afin de l'améliorer. Les fonctions de bases y sont implémentées et j'ai beaucoup insisté pour donner une interface qui soit à la hauteur de ce langage. Il permet l'écriture de code dans le terminal ou encore dans le MultiConsole pour autant que le code n'ait pas une syntaxe trop exotique. Elle permet de rapidement écrire en Scala et de tester des petits morceaux de code sans se soucier d'une quelconque installation de plugin ou de la compatibilité du navigateur par exemple. La gestion des sessions fonctionnent, elle consomme peu de ressources et est rapide (les requêtes prennent entre 250 et 400ms). Des modifications ultérieures dans l'interpréteur ne devraient pas perturber le fonctionnement de la console web à moins si la sortie est modifiée et dans ce cas là, il faudra modifier le parseur côté serveur mais rien n'aura besoin d'être changé côté client.

Je remercie encore Gilles Dubochet pour son temps et ses explications et également le LAMP pour avoir piraté ma machine. Je suis content d'être arrivé au bout de ce projet et si vous désirez améliorer ou corriger certains aspects de cette application, n'hésitez pas à me contacter ! Try Scala en attendant !