Rapport de stage chez eBusiness Information

eBusiness Information
43-45 Avenue Carnot
94230 Cachan

Gatling - Test de charge optimisé

Stagiaire :
Romain Sertelon

Tuteur de stage :
François-Pierre Chalopin

Université Paul Cézanne – Aix-Marseille III
Master SIS – Sciences de lInformation et des Systèmes
Filière Professionnelle – Spécialité Informatique – Parcours Génie Logiciel

Période de stage : du 1er mars 2011 au 31 août 2011

Remerciements

Je souhaite remercier messieurs Olivier Duvoid, François-Pierre Chalopin et Fabrice Reby de m'avoir accepté en tant que stagiaire dans leur entreprise.

Je souhaite également remercier Stéphane Landelle, grâce à qui j'ai pu travailler sur Gatling.

Enfin je voudrais également remercier tout le personnel du groupe Excilys qui m'a accueilli chaleureusement et aidé tout au long de mon stage.

Sommaire

Remerciements **i**

Glossaire **vii**

1 Introduction **1**
1.1 eBusiness Information et Excilys . . . 1
1.1.1 La Charte Excilys . . . 1
1.1.2 Le groupe Excilys . . . 2
1.2 Formation Java/JEE . . . 2
1.2.1 Être stagiaire chez eBusiness Information . . . 2
1.2.2 Capico . . . 2
1.2.3 Contenu de la formation . . . 3
1.3 Ma mission . . . 4

2 Description du problème **7**
2.1 Les tests en génie logiciel . . . 7
2.1.1 Généralités . . . 7
2.1.2 Principe des tests de charge . . . 7
2.2 Motivations du projet . . . 8
2.2.1 Outils Existants . . . 8
2.2.2 Limitations des outils actuels . . . 8
2.2.3 Gatling . . . 10

3 Problématiques **11**
3.1 Cadre d'utilisation de Gatling . . . 11
3.2 Ecriture de scénarios . . . 11
3.2.1 Interface de l'outil . . . 11
3.2.2 Utilisation d'un DSL . . . 12
3.3 Performances . . . 12
3.3.1 Modèle actuel . . . 12
3.3.2 Le modèle d'acteur . . . 13
3.3.3 Le modèle de Gatling . . . 14
3.4 Langage de Gatling . . . 15
3.5 Rapports de simulation . . . 16
3.5.1 Types de rapports . . . 16
3.5.2 Format des rapports . . . 16
3.5.3 Génération des rapports . . . 17

4 Conception de Gatling **19**
4.1 Architecture de Gatling . . . 19
4.1.1 Entrées & sorties . . . 19
4.1.2 Application modulaire . . . 20
4.1.3 Structure des dossiers . . . 20
4.2 Modules de l'application . . . 21
4.2.1 Gatling App . . . 21
4.2.2 Gatling Core . . . 22
4.2.3 Gatling HTTP . . . 23
4.2.4 Gatling Statistics . . . 23
4.3 Le DSL . . . 25
4.3.1 Analyse d'un test de charge . . . 25
4.3.2 Définition du DSL . . . 26
4.3.3 Fonctionnalités supplémentaires du DSL . . . 27

Conclusion **31**

Table des illustrations

1.1 Droits et devoirs pour les acteurs du service équitable 1
1.2 Interface de Capico 3
1.3 Patrick's Bank - Visualisation du détail d'un compte 4

2.1 Interface de JMeter 9
2.2 Interface de LoadUI 10

3.1 Composition d'un acteur 13
3.2 Enchainement d'acteurs 14
3.3 Fonctionnement de la preuve de concept 14
3.4 Fonctionnement d'un moteur de template 17

4.1 Entrées et sorties de Gatling 19
4.2 Architecture de Gatling 20
4.3 Structure de dossiers de Gatling 21
4.4 Menu de sélection de la simulation à démarrer 21
4.5 Composants de Gatling Core 22
4.6 Composants de Gatling HTTP 23
4.7 Composants de Gatling Statistics 24
4.8 Génération de statistiques 24
4.9 Exemple de statistiques générées 25

Glossaire

Denial of Service Attaque informatique visant à rendre inopérant un serveur en lui envoyant un très grand nombre de requêtes par seconde.

CPU *Central Processing Unit*. La CPU, ou microprocesseur, est une puce informatique capable d'effectuer des calculs. C'est le "cerveau" de l'ordinateur.

AJAX *Asynchronous Javascript And XML*. AJAX est le regroupement sous un seul nom, de différentes technologies permettant de créer des interfaces web plus riches et d'offrir une expérience utilisateur plus agréable.

HTTP *HyperText Transfer Protocol*. HTTP est le protocole utilisé par les navigateurs internet et les serveurs web afin de communiquer.

LDAP *Lightweight Directory Access Protocol*. LDAP est une norme pour les annuaires informatiques.

DSL *Domain Specific Language*. Un DSL ou Langage spécifique à un domaine, permet de décrire des entités grâce à un langage créé uniquement dans ce but.

IDE *Integrated Development Environment*. Un environnement de développement intégré est un logiciel qui permet au développeur de créer facilement des applications.

PoC *Proof of Concept*. Une preuve de concept est une application très simple visant à valider un concept avant de l'utiliser à plus grande échelle.

JVM *Java Virtual Machine*. La JVM est une application qui permet l'exécution de programmes compilés en bytecode, le plus souvent écrits en Java.

REST et RESTful *REpresentational State Transfer*. Architecture d'applications fonctionnant autour de ressources et utilisant les verbes HTTP afin d'agir sur ces ressources.

Chapitre 1

Introduction

1.1 eBusiness Information et Excilys

1.1.1 La Charte Excilys

eBusiness Information a été créée en l'an 2000 dans le but de répondre à des problématiques métier diverses grâce à la mise en place de solutions informatiques robustes, tout en mettant en valeur le savoir-faire humain qui amène à ces solutions.

En effet, pour ses fondateurs, le service informatique est un métier réalisé par des hommes qui doivent avoir de plus en plus de compétences. En 2002, ils imaginent une notion nouvelle dans le monde du service informatique : le **Service équitable**[1]. Inspiré par le commerce équitable, cette notion sera inscrite dans une charte : la **Charte Excilys**. Le service équitable consiste en la création d'un cercle vertueux [1] profitant à tous les acteurs du service. Tout comme le commerce équitable, cette notion vise à profiter aux personnes qui produisent le service plutôt qu'aux intermédiaires seuls.

La **Charte Excilys** définit donc un ensemble de droits et de devoirs entre les trois acteurs du service : le client, l'entreprise et le consultant comme montré figure 1.1.

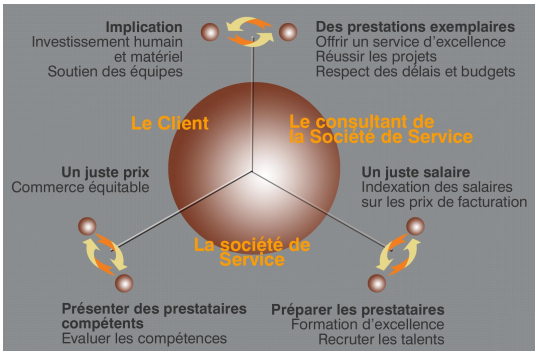

FIGURE 1.1 – Droits et devoirs pour les acteurs du service équitable

L'application des règles de cette charte était un pari risqué, mais un pari réussi. En effet, après plusieurs années de fonctionnement, eBusiness Information a prouvé qu'un modèle

1. L'entreprise facture le client plus cher que la moyenne. Cette facturation profite au consultant qui fournit donc un travail de qualité. Ce travail de qualité satisfait le client. Ainsi, les acteurs concernés par le service rendu y trouvent tous leur compte.

basé sur les compétences humaines peut fonctionner. De plus, les résultats de l'entreprise sont extrêmement encourageants : relations durables avec les clients, aucunes dettes, aucun investisseur extérieur et aucun projet raté.

1.1.2 Le groupe Excilys

Le groupe Excilys est né de la volonté de reproduire ce modèle dans d'autres sociétés de service qui souhaitent elles aussi centrer leur métier sur l'excellence ; c'est d'ailleurs du mot excellence que vient le nom **Excilys**.

Depuis, plusieurs entreprises ont rejoint le groupe, elles sont aujourd'hui au nombre de sept : Adlys, Altendis, eBusiness Information, Edvance, Equitalis, SS2J et Visual3X. Chacune de ces sociétés fournit des services en informatique mais avec des domaines de compétences complémentaires et une image de qualité qui lui est propre.

1.2 Formation Java/JEE

1.2.1 Être stagiaire chez eBusiness Information

Excilys [2] emploie de nombreux stagiaires chaque année ; par exemple, pendant l'année 2011, c'est plus d'une vingtaine de stagiaires qui a été accueillie chez Excilys. L'entreprise mise beaucoup sur les jeunes informaticiens en fin d'études afin d'embaucher des consultants compétents ; ce qui lui permettra de continuer à offrir un service de qualité.

Dans cette optique, un grand soin est apporté au confort des stagiaires : hébergement disponible, salaire confortable, et, surtout, une formation de six semaines au début du stage afin de leur permettre de monter en compétence sur de nombreuses technologies. Le fait que les stagiaires soient en colocation dans le même immeuble, et qu'ils travaillent dans le même open space permet de créer rapidement des liens et cela participe à une émulation générale, ainsi qu'à une bonne ambiance.

A la suite de la formation, chaque stagiaire se voit proposer un sujet de stage différent, et parfois, une mission chez un client qui se poursuit fréquemment après la fin du stage.

1.2.2 Capico

Afin de former leurs stagiaires et d'offrir à leurs consultants la possibilité de se former facilement sur diverses technologies utilisées dans le domaine Java/JEE, les gérants d'eBusiness Information ont investi dans le développement d'un outil de formation en ligne nommé **Capico** [3]. Cet outil contient des cours (audio ou non) ainsi que des exercices et travaux pratiques avec leurs corrigés, la figure 1.2 représente l'interface (côté élève) de Capico.

Aujourd'hui, Capico est devenu bien plus qu'un outil de formation interne. En effet, avec le temps, et grâce aux investissements en R&D d'Excilys, ce logiciel est devenu une plateforme d'e-coaching qui propose des cours gratuits du CP au CM2, en français et mathématiques[2] et qui est récemment entré en compétition dans un appel d'offres du département des Hauts-de-Seine afin de moderniser ses collèges.

2. Dans le reste de ce rapport, eBusiness Information et Excilys, bien que représentant deux entités juridiques différentes, désigneront l'entreprise où j'ai réalisé mon stage.

3. Capico pour **Capi**talisation des **co**nnaissances

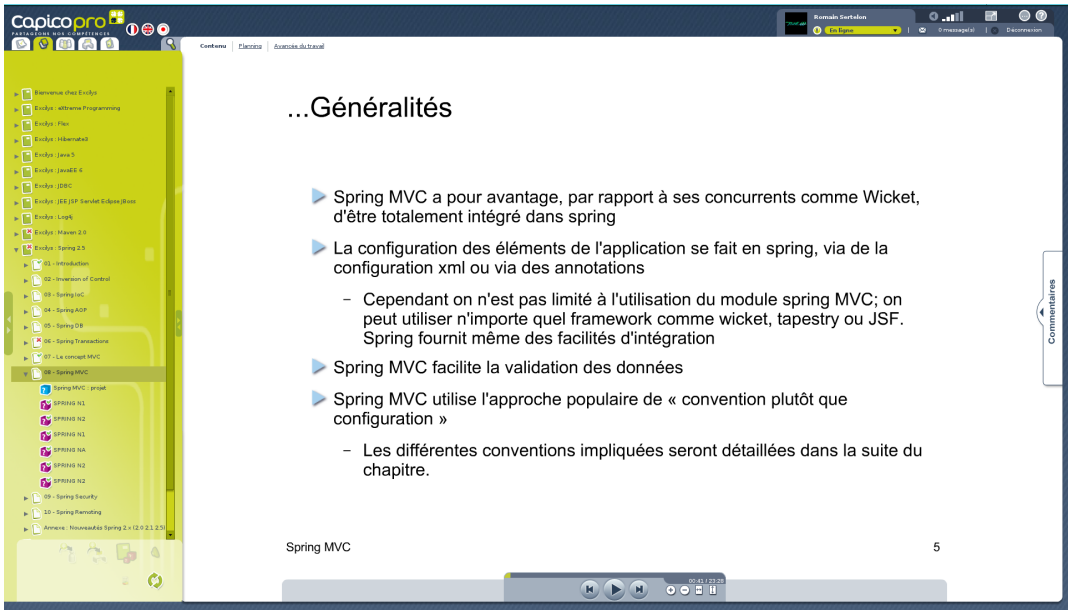

FIGURE 1.2 – Interface de Capico

1.2.3 Contenu de la formation

Cours sur Capico

La période de six semaines de formation commence par des cours sur Capico couvrant diverses technologies mais aussi des notions sur les méthodes agiles, en particulier **eXtreme Programming**. Les technologies abordées sont les suivantes :

- **UML / Java 5** - Un rappel sur Java et le langage de formalisation UML ;
- **JEE JSP+Servlets / Java EE 6** - Un rappel sur Java Enterprise (EJB, Servlet, JSP, JSF) ;
- **JDBC / Hibernate** - Un cours sur JDBC et Hibernate, un des ORM Java les plus utilisés en entreprise ;
- **Log4j / Slf4j** - Un cours sur la journalisation en Java ;
- **JUnit** - Un cours sur le framework JUnit qui permet de faire du test unitaire ;
- **Spring** - Un cours sur le framework Spring (IoC, MVC, etc.) ;
- **Maven** - Un cours sur un gestionnaire de dépendances de plus en plus utilisé au sein de la communauté Java ;
- **Subversion / Git** - Un cours sur les deux gestionnaires de sources les plus populaires ;
- **Flex** - Un cours expliquant les bases du développement avec Flex.

Ces cours offrent un tour d'horizon des technologies les plus utilisées par les clients de l'entreprise, ainsi que les technologies utilisées au sein du projet Capico. En effet, chaque stagiaire travaille sur le projet Capico avant d'être affecté à une mission ou à un projet différent, dans le but de mettre en pratique ce qu'il a appris sur un projet concret.

Réalisation d'un projet

La formation, est complétée par la mise en pratique des cours grâce à la réalisation d'une application d'e-banking dans le cadre des méthodes agiles. Notre groupe de six stagiaires à

réalisé son application [4] sur cinq semaines, soit cinq itérations en développant par binôme tel que préconisé par la méthode eXtreme Programming. La figure 1.3 représente une page du site que nous avons réalisé.

FIGURE 1.3 – Patrick's Bank - Visualisation du détail d'un compte

Nous avons pu ainsi nous confronter aux technologies que nous n'avions jusqu'alors pas pratiquées : Intégration Continue avec Jenkins, utilisation de Google code, de Maven, etc.

1.3 Ma mission

A la fin de ma période de formation, j'ai commencé par travailler sur Capico pendant deux semaines. Puis Stéphane Landelle, directeur technique d'eBusiness Information, m'a proposé de travailler avec lui sur une idée dont il avait déjà réalisé une **Preuve de Concept**. Cette idée consiste en la création d'un nouvel outil de test de montée en charge – ou injecteur HTTP – nommé Gatling[3].

Dès lors, j'ai été en charge de concevoir, puis d'implémenter les fonctionnalités que l'outil doit proposer. L'objet de ce rapport consiste en la présentation de la réalisation de ce nouvel outil.

Le chapitre 2 définira ce qu'est un test de charge et motivera la création d'un nouvel outil permettant d'automatiser ce type de tests. Le chapitre 3 décrira l'analyse des fonctionnalités

4. Disponible sur Github : http ://github.com/BluePyth/patricks-bank

que Gatling devra implémenter afin de trouver des solutions aux divers problèmes soulevés par la création d'un tel outil. Enfin, le chapitre 4 détaillera la conception de Gatling et listera la plupart des fonctionnalités disponibles dans sa version actuelle, la 1.0.0.M3.

Chapitre 2

Description du problème

2.1 Les tests en génie logiciel

2.1.1 Généralités

Afin de tester le bon comportement d'une application ou de l'un de ses composants, il existe différentes techniques. Généralement, on distingue :

- **Les tests unitaires**. Ils permettent de vérifier le bon fonctionnement d'un composant, par exemple le comportement d'une méthode d'une classe dans certaines conditions. Il faut donc en créer un grand nombre afin d'avoir une couverture de code maximale. Ces tests ne permettent de valider qu'un seul composant d'une application.
- **Les tests fonctionnels**. Ceux-ci sont effectués à une échelle plus grande que les tests unitaires. En effet, ils valident le comportement d'un ensemble de composants réalisant une fonction particulière. Les tests fonctionnels permettent de s'assurer que l'application apporte bien les fonctionnalités attendues.

On peut donc, grâce à ces deux types de tests, s'assurer du bon fonctionnement, d'un point de vue fonctionnel, de l'application.

Cependant, il existe une autre sorte de tests qu'il faut parfois réaliser, ce sont les tests de performance. Ces tests concernent particulièrement les applications à architecture **client-serveur** qui risquent d'être fortement sollicitées. On distingue deux types de tests de performances :

- **Les tests de charge**. Ils permettent de mesurer des performances en terme de temps de réponse du serveur, en fonction du nombre d'utilisateurs connectés et des types de requêtes envoyées. On utilise ce type de test pour s'assurer que les clients pourront obtenir leur réponse dans un laps de temps suffisament court.
- **Les tests de stress**. Ces tests là permettent de pousser le système au-delà de ses limites et de s'assurer qu'il est robuste. On réalise ces tests pour s'assurer qu'un serveur ne tombera pas sous une attaque de type **Denial of Service** par exemple.

2.1.2 Principe des tests de charge

Simulation d'utilisateurs

Afin de tester la résistance d'une application et de l'architecture qui l'héberge, on utilise des outils de test qui permettent de simuler des milliers d'utilisateurs.

Pour simuler ces utilisateurs, on définit des scénarios d'utilisation. Chaque scénario représente un comportement utilisateur typique. On peut par exemple, simuler l'utilisation d'un site web par un expert − utilisation de recherche avancée, comparaisons multiples, etc. − ou par un débutant − recherche simple, clic sur les publicités, etc.

Ces scénarios sont ensuite exécutés par ces outils un certain nombre de fois, en parallèle, afin d'imposer une charge au serveur testé. Une fois l'exécution des scénarios terminée, l'outil fourni en général des informations sur leur déroulement : temps de réponse moyen, nombre de requêtes par seconde, etc.

Test de charge brute

Une autre approche du test de charge consiste à déterminer le nombre maximal de requêtes par seconde que l'application peut supporter. Ce qui peut être utile d'un point de vue technique, mais qui ne donne aucune idée du nombre d'utilisateurs simultanés qui pourront utiliser cette application.

2.2 Motivations du projet

2.2.1 Outils Existants

Etant donné que l'idée de faire des tests de charge existe depuis longtemps, un certain nombre d'outils logiciels destinés à exécuter ce type de tests est déjà disponible sur la JVM. Voici une liste des principaux outils open source et/ou gratuits existants lors de l'écriture de ce rapport.

JMeter

JMeter[4] est un projet de la fondation Apache qui permet de faire du test de charge. Il est assez connu dans la communauté Java et offre une interface graphique, ainsi que des modules lui permettant de tester des serveurs avec différents protocoles, ainsi que des modules de visualisation des résultats des tests. La construction des scénarios se fait via une interface graphique, sous forme d'arbre, comme montré Figure 2.1.

LoadUI

LoadUI[5] est un projet de la société SmartBear (anciennement Eviware) qui a créé le logiciel SoapUI[6] utilisé pour tester des webservices de manière fonctionnelle. LoadUI a été créé suite à une forte demande de la communauté d'utilisateurs de SoapUI et intègre donc logiquement ce dernier. Son interface utilise JavaFX et est très soignée, à l'inverse de JMeter comme le montre la Figure 2.2.

2.2.2 Limitations des outils actuels

Les outils disponibles pour faire du test de charge fonctionnent bien mais ont quelques lacunes.

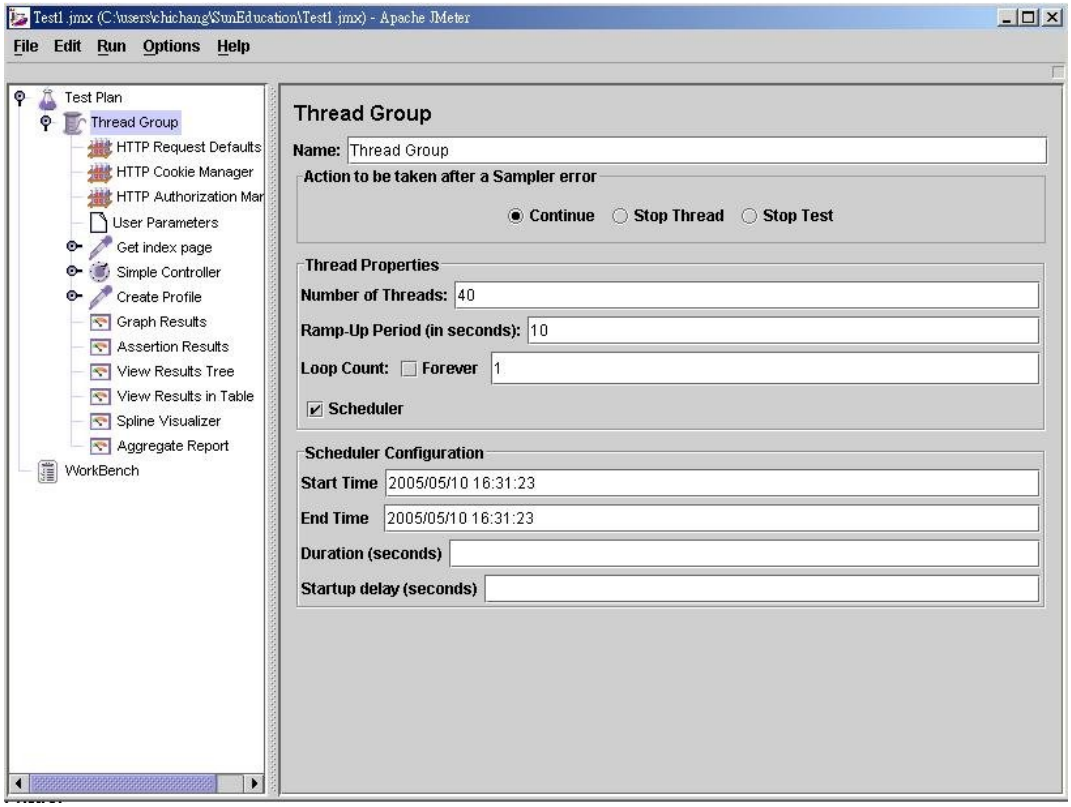

FIGURE 2.1 – Interface de JMeter

Interface Graphique

Les outils tels que JMeter ou LoadUI ne proposent qu'une façon de créer des scénarios et de les exécuter : via une interface graphique. Il n'y a aucun moyen de passer outre et si l'on veut modifier les fichiers sources des scénario à la main − si l'on n'a pas accès au logiciel, par exemple −, cela se révèle complexe car les fichiers n'ont pas été prévus pour être lisible par l'homme, mais pas des ordinateurs.

Vocabulaire utilisé

De plus, ces interfaces graphiques contiennent beaucoup de notions que l'utilisateur doit maîtriser en plus de l'application à tester. Par exemple, pour LoadUI, on trouve des *load generators*, des *runners*, etc. Les développeurs de JMeter, quant à eux, ont proposé un vocabulaire plus proche de l'implémentation de leur application que du domaine visé par l'application : *Thread Groups*, *Controllers*, *Listeners*, etc.

Performances

Afin d'effectuer des tests de charge significatifs, il faut être capable de simuler un grand nombre d'utilisateurs ou de requêtes par seconde. Les outils existants consomment beaucoup de mémoire et de puissance CPU et proposent une solution afin de remédier à ce problème : la distribution des tests sur plusieurs machines. Cela induit donc des coûts supplémentaires. De plus, cela oblige le testeur à surveiller ces différentes machines afin de vérifier que le test

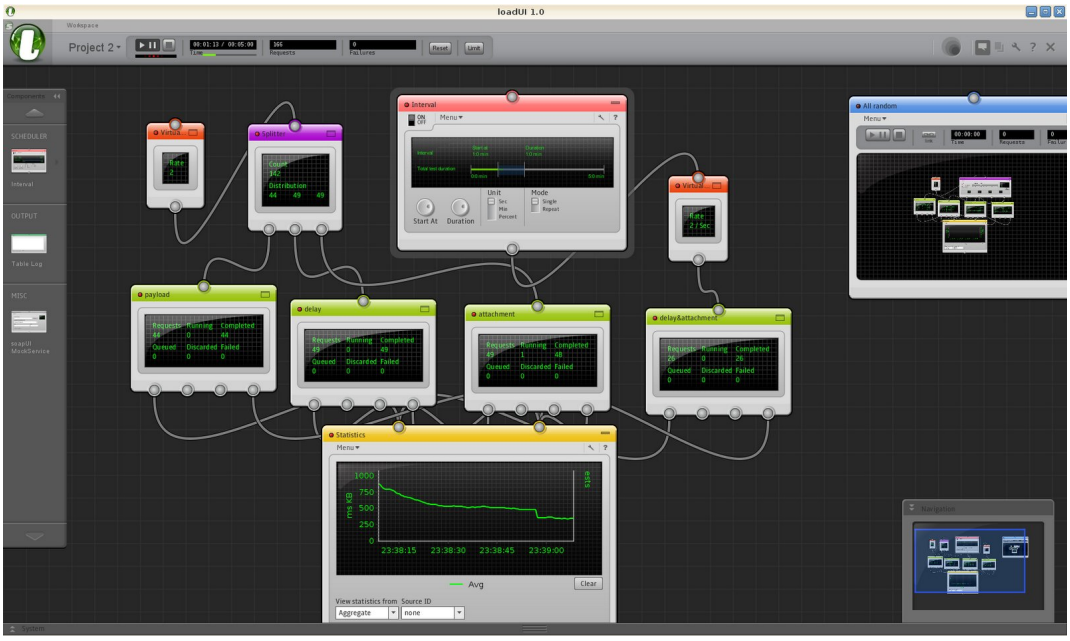

FIGURE 2.2 – Interface de LoadUI

lancé soit vraiment significatif. En effet, si un des agents de test plante ou ralentit pour une raison quelconque, le test effectué ne fournira pas les résultats attendus.

2.2.3 Gatling

Le projet Gatling naît donc d'une volonté d'aller au-delà de ces limitations et de proposer à ses futurs utilisateurs un logiciel :

- **Performant**. Gatling permettra d'effectuer des tests complexes et lourds en utilisant le moins de ressources possible, grâce à l'optimisation de ses performances. Ceci étant la raison principale motivant sa création.
- **Simple d'utilisation**. Gatling offrira une interface utilisateur simple et rapide à prendre en main.
- **Accessible**. Gatling utilisera un vocabulaire proche du domaine du test de charge.

Chapitre 3

Problématiques

3.1 Cadre d'utilisation de Gatling

La cible première de Gatling est l'ensemble des applications web modernes. Cela signifie que l'outil doit pouvoir envoyer des requêtes sur le protocole HTTP ou HTTPS et fournir la possibilité de simuler des requêtes AJAX [1]. Dans les évolutions suivantes, il est prévu que Gatling supporte les applications réalisées avec le framework GWT ou avec Flex. Il est également prévu d'y ajouter des modules permettant de tester des serveurs de données, des annuaires LDAP, etc.

Afin de garantir des résultats exploitables par le testeur et ses dirigeants, Gatling simulera, à l'aide de scénarios définis par le testeur, l'utilisation de l'application ciblée. La difficulté de ce genre de test réside dans la rédaction des scénarios, qui doivent être bien conçus pour représenter au plus près la réalité.

Si l'écriture des scénarios est laissée aux testeurs, Gatling tentera de simplifier au maximum cette écriture. Cela permettra au testeur de ne se concentrer que sur le contenu de ses scénarios, plutôt que sur la façon dont il devra les saisir.

3.2 Ecriture de scénarios

L'écriture des scénarios doit donc être fluide, et la plus naturelle possible pour le testeur. Les éléments décisifs qui rendent cela possible sont l'interface de l'outil et la représentation des scénarios.

3.2.1 Interface de l'outil

L'interface de l'outil doit être la plus simple possible afin de ne pas rendre son utilisation trop complexe. Elle peut être graphique, comme les outils présentés plus haut dans ce rapport, ou alors en ligne de commande.

Gatling étant amené à être exécuté sur des serveurs sans interface graphique, l'interface par ligne de commande sera privilégiée. Celle-ci pourrait rebuter les utilisateurs peu expérimentés, cependant, le test de charge est souvent réalisé par des personnes qui connaissent bien l'environnement informatique, soit parce qu'ils développent − ce sont parfois les développeurs eux-mêmes qui font ce travail −, soit parce qu'ils administrent des serveurs.

1. AJAX signifie *Asynchronous Javascript And XML* : Javascript et XML Asynchrones

3.2.2 Utilisation d'un DSL

Le choix d'utiliser la ligne de commande limite considérablement les possibilités de création d'un scénario. En effet, cette contrainte induit de ne pas fournir d'interface graphique comme le font JMeter et LoadUI.

Ainsi, il faut se tourner vers une représentation textuelle des scénarios. Cela obligerait donc le testeur à apprendre la formalisation des scénarios qui a été choisie par Gatling. Cette solution va à l'encontre d'une simplification de l'écriture des scénarios.

Pourtant, il existe un moyen efficace de s'affranchir de l'apprentissage de la formalisation : les **DSL**[2]. Ce sont des langages composés d'appels de méthodes successifs. Ainsi en choisissant le nom des méthodes, on peut faire en sorte d'écrire des phrases. Leur utilisation apporte plusieurs avantages :

— **Un langage naturel**. Un DSL bien conçu permet d'écrire de façon quasiment naturelle ce que l'on veut représenter. En voici un exemple simple dans le cadre des mathématiques :

```
// Create a point named A with coordinates (2, 6)
createPoint "A" withAbscissa 2 andOrdinate 6
```

— **Utilisation des fonctions de base d'un langage**. Etant donné qu'un DSL est écrit avec un langage de programmation et compilé (ou exécuté en tant que script), on peut naturellement y utiliser toutes les fonctionnalités apportées par le langage si besoin. En effet, supposons que l'on veuille créer un point avec une ordonnée avec une valeur égale à deux fois la valeur de l'abscisse, on pourra écrire :

```
val x = 3
// Create a point named A with coordinates (x, 2x)
createPoint "A" withAbscissa x andOrdinate 2*x
```

— **Autocomplétion**. L'utilisation d'un langage de programmation afin d'écrire des DSL permet aussi d'utiliser les fonctionnalités que peut offrir un IDE. En effet, l'IDE pourra aider à l'écriture du code utilisant le DSL en proposant les mots disponibles à l'utilisateur lors de l'écriture du code.

L'utilisation d'un DSL permettra donc aux utilisateurs de Gatling de bénéficier d'une façon simple et efficace d'écrire leurs scénarios.

3.3 Performances

3.3.1 Modèle actuel

Dans la section 2.2.2, la mise en cause des performances des outils de test actuels a été évoquée. En effet, lorsque ces outils veulent simuler des utilisateurs, ils adoptent comme modèle 1 utilisateur = 1 thread[3]. Cette approche apparait la plus naturelle, et permet de simuler les temps de pause utilisateur en mettant directement les threads en sommeil.

Cette façon de procéder donne généralement des résultats satisfaisants lorsque le nombre d'utilisateurs n'est pas très élevé et le travail de chaque thread relativement simple. Dès que le nombre d'utilisateurs croît ou que le scénario demande un traitement des réponses important,

2. *Domain Specific Language* Langage spécifique à un domaine
3. Un thread est un fil d'exécution dans une application

l'exécution de la simulation devient imprévisible. Les temps de pause peuvent parfois varier du simple au double, et le scénario simulé perd ainsi de sa valeur, car il ne correspond pas à ce que souhaitait le testeur.

3.3.2 Le modèle d'acteur

La programmation concurrente est généralement faite en utilisant des threads. On doit donc faire attention à la synchronisation des méthodes, bien veiller à ce que l'on n'ait pas de situation de compétition entre les threads, et tout faire pour éviter les dead-locks. Programmer en utilisant des threads peut donc vite devenir complexe.

Pour pallier cette complexité, on peut utiliser un modèle dit **modèle d'acteur**. Ce modèle, comme l'indique Wikipedia[7],

> "est un modèle mathématique qui définit les acteurs comme les seules entités nécessaires au calcul concurrent. En réponse à un message, un acteur effectue un traitement, crée d'autres acteurs et envoie d'autres messages."

Ainsi, la programmation concurrente à l'aide d'acteurs est basée sur l'envoi de messages et des traitements asynchrones. Un acteur est composé d'une **unité de calcul** et d'une **messagerie**, comme illustré dans la figure 3.1.

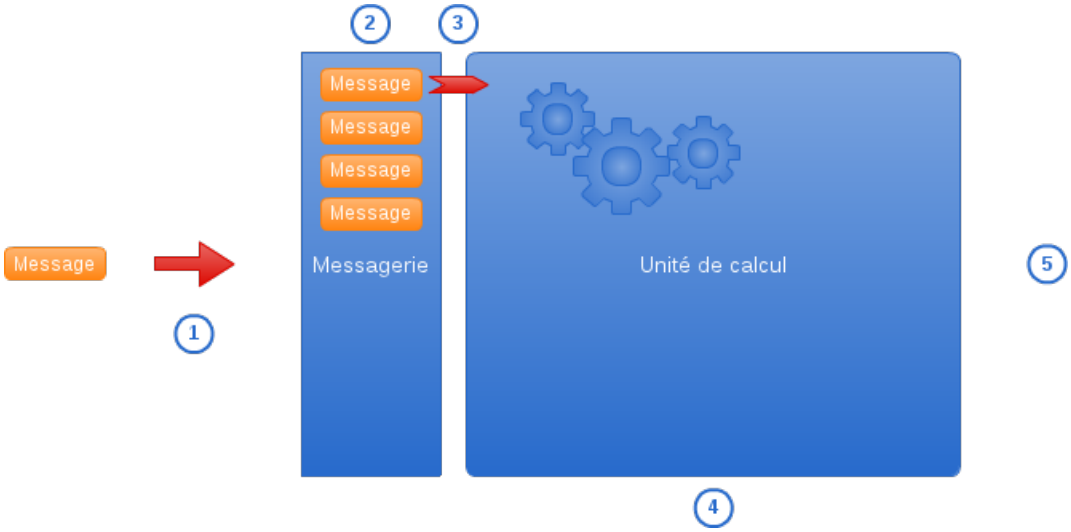

FIGURE 3.1 – Composition d'un acteur

Le fonctionnement de ce modèle est le suivant :

1. Un message est envoyé à l'acteur. Ce message est un ordre donné à l'acteur, il contient, si besoin, les paramètres nécessaires au calcul que l'acteur doit effectuer ;
2. Le message arrive dans la messagerie de l'acteur, il est stocké dans une file d'attente ;
3. Lorsqu'il y a un message dans la messagerie, l'acteur le traite, il est ensuite supprimé de la file d'attente ;
4. Le traitement de l'ordre contenu dans le message est effectué par l'unité de calcul ;
5. Enfin, l'acteur peut passer directement au message suivant, ou envoyer des messages à d'autres acteurs avant de passer au message suivant comme illustré par la Figure 3.2.

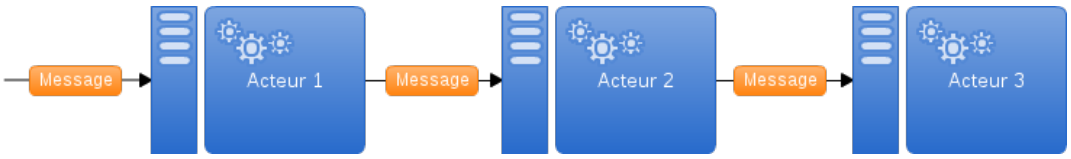

FIGURE 3.2 – Enchainement d'acteurs

3.3.3 Le modèle de Gatling

Gatling doit donc utiliser le modèle le plus performant afin de garantir les performances qu'il veut offrir. Afin de démontrer que l'approche par le modèle d'acteurs est la plus performante, des tests ont été faits pour comparer JMeter et une preuve de concept utilisant ce modèle.

Fonctionnement de la preuve de concept

La preuve de concept utilise donc le modèle d'acteur afin de simuler l'exécution d'un scénario. La figure 3.3 illustre la manière dont est utilisé le modèle d'acteur.

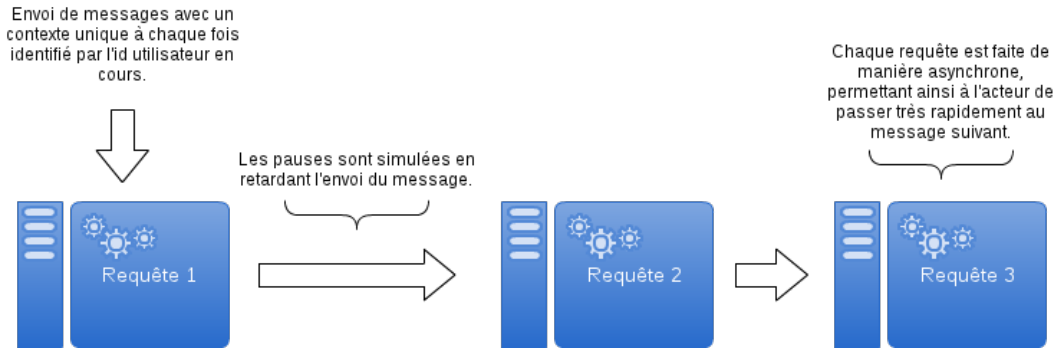

FIGURE 3.3 – Fonctionnement de la preuve de concept

Chaque étape d'un scénario est représentée par un acteur qui se charge uniquement d'effectuer la requête demandée à cette même étape. Ainsi, les acteurs sont chainés, chaque acteur envoie un message à l'acteur suivant lorsqu'il a fini, afin que la requête suivante puisse être exécutée. Afin de lier les acteurs entre eux, on utilise un contexte qui est transmis à chaque fois et qui permet de savoir à tout instant pour quel utilisateur la requête est effectuée. Ce fonctionnement sera repris dans le moteur de simulation de Gatling.

Le protocole de test

Les tests ont été effectués avec :

— Un PC avec Processeur 4 coeurs 2.67GHz, JVM lancée avec 2Go de mémoire alloués (Heap) dédiés à l'utilisation de l'outil. L'exécution de la simulation a été faite avec 4Go de RAM au départ pour les deux outils, puis à 5Go, étant donné que JMeter consomme beaucoup de RAM ;

— Un serveur web apache tournant sur une autre machine servant une seule page de 100ko ne contenant que du texte ;

— Un scénario type : L'utilisateur appelle la page un certain nombre de fois, en effectuant une pause entre chaque appel. Ce nombre de pauses induit une durée de simulation incompréssible de 10mn04s ;
— Deux types d'opérations sur la réponse du serveur [4] ont été effectués :
  — Recherche par **expression régulière** ;
  — Recherche par **XPath**.

Conclusions des tests

Les résultats de ces tests, consignés dans le tableau 3.1, indiquent que le modèle d'acteur est bien plus performant que le modèle utilisé par JMeter. En effet, JMeter consomme plus de mémoire vive et la simulation sur JMeter est parfois bien plus longue qu'avec la preuve de concept. De plus, en surveillant l'activité des coeurs du processeur, on remarque qu'avec la preuve de concept, l'utilisation des coeurs est régulière et oscille autour de 55%, alors que pour JMeter, les coeurs sont très sollicités, avec des plafonnements réguliers à 100%.

Utilisateurs	RAM	Opération	Durée PoC	Durée JMeter	Diff.
2000	4Go	RegExp	10'08"	10'21"	13"
3000	4Go	RegExp	10'11"	12'01"	1'50"
2000	4Go	XPath	10'12"	13'08"	2'56"
2000	5Go	XPath	10'14"	10'45"	31"
3000	5Go	XPath	10'20"	18'36"	8'16"

Table 3.1 – Résultat des tests modèle d'acteurs vs threads

L'utilisation du modèle d'acteur pour le moteur de simulation de Gatling est donc un bon choix et présage un gain de performances non négligeable pour des scénarios complexes qui demandent des opérations gourmandes sur les réponses reçues.

3.4 Langage de Gatling

Une question subsiste : en quel langage Gatling doit-il être codé ? Afin de réaliser Gatling, nous avons besoin d'un langage qui ait :

— **Une bibliothèque d'acteurs performante et assez mature**. En effet, ce modèle, bien que découvert il y a déjà longtemps [5], n'est pas encore très répandu dans le monde informatique professionnel.
— **Une syntaxe autorisant la création de DSL simplement**. Il n'existe pas beaucoup de langages qui permettent de s'affranchir des éléments de syntaxe qui rendent le code moins naturel tels que ., {, }, (,), etc.
— **Une compatibilité avec la JVM**. En effet, eBusiness Information travaillant dans les technologies Java/JEE, et la plupart des systèmes d'information d'entreprise fonctionnant avec Java, il est naturel de proposer un logiciel qui fonctionnera sans avoir à installer des dépendances supplémentaires.

4. Le fait d'effectuer une opération sur la réponse du serveur permet de tester la robustesse de l'outil face à de telles demandes lorsque leur nombre augmente.
5. Ce modèle a été découvert en 1973[8]

Le langage **Scala**[9] a ainsi été choisi. En effet, ce langage répond aux exigences exprimées précédemment et propose même une compatibilité avec toutes les bibliothèques Java ! Cela permet donc de bénéficier d'un langage fonctionnel et objet, tout en ayant accès aux bibliothèques connues de Java. Le listing 3.1 montre un exemple de syntaxe Scala.

```
object HelloWorld {
  def main(args: Array[String]) {
    println("Hello world!")
  }
}
```

Listing 3.1 – Exemple de code Scala

La bibliothèque pour le modèle d'acteurs se nomme **Akka**[10]. Ses créateurs ont rejoint l'entreprise TypeSafe qui édite le langage Scala, la bibliothèque sera donc maintenue et bénéficiera de l'expertise des créateurs de Scala. De plus, elle est Open Source.

3.5 Rapports de simulation

3.5.1 Types de rapports

Comme tout bon outil de test, Gatling devra fournir au testeur un moyen de visualiser facilement ce qu'il s'est passé durant le test, entre autres :

— Le nombre d'utilisateurs actifs à n'importe quel instant de la simulation ;
— Le temps de réponse des différentes requêtes durant le test ;
— Le nombre de requêtes envoyées par seconde ;
— ...

Afin de montrer ces résultats aux testeurs, il existe plusieurs options :

— **Une version texte**. Cette version ne contient pas d'images et peut être visualisée avec n'importe quel éditeur de texte. Cependant, le déchiffrage de ce type de rapports, lorsqu'il y a des dizaines de milliers de requêtes effectuées dans une simulation, peut s'avérer très complexe, et peu utile aux testeurs.
— **Une version graphique**. Cette version est sans doute la plus simple à interpréter et permet de visualiser rapidement le comportement de l'application durant les tests.
— **Une version brute**. Cette version, bien que plus compliquée à lire qu'une version texte offre l'avantage de pouvoir être importée dans des logiciels de type tableur (Microsoft Excel, LibreOffice Calc, Google docs, etc.)

Gatling devra donc offrir des rapports sous forme graphique et sous forme brute, type fichiers CSV.

3.5.2 Format des rapports

Afin de visualiser les données sous forme graphique, il existe, là aussi, plusieurs solutions. On peut faire un module logiciel type application lourde, générer des fichiers PDF [6], ou générer des rapports au format HTML.

L'interface de l'outil étant en ligne de commandes, il est logique d'écarter la création d'un module logiciel. Il reste donc la génération de fichiers PDF ou HTML.

6. *Portable Document format*

Les fichiers HTML ont l'avantage d'être visualisables sur n'importe quel ordinateur possédant un navigateur web, on touche ainsi à tous types d'appareils : ordinateurs, smartphones, tablettes. Les fichiers PDF, eux, nécessitent l'installation d'un logiciel tiers.

De plus, les fichiers HTML peuvent être rendus dynamiques, autorisant, plus tard, la visualisation des données en temps réel, ce qui n'est pas possible avec le format PDF qui est statique et ne peut être généré qu'à la fin du test.

Les tests de charge demandant une infrastructure particulière afin d'être menés à bien, il est certain que Gatling sera amené à être utilisé sur un serveur dédié à cet usage. C'est pour répondre à ceci que Gatling offrira une interface web dans le futur, faisant ainsi des rapports HTML une solution privilégiée.

La génération de rapports PDF offre par contre un bon complément au rapport sous forme HTML, car comme son nom l'indique : il est portable. La production de rapports au format PDF est donc prévue dans les futures versions de Gatling.

3.5.3 Génération des rapports

La génération de rapports au format HTML peut être réalisée simplement en créant la page morceau par morceau grâce à du code. Cependant, cette technique n'est pas la plus conseillée. En effet, cela induit un temps de développement plus long en général à cause de la difficulté à maintenir des pages HTML de ce type.

Pour pallier cela, on utilise un moteur de template nommé Scalate[11], le moteur de template écrit en Scala et utilisé dans le framework Play ! par exemple. Le fonctionnement d'un moteur de template est illustré figure 3.4.

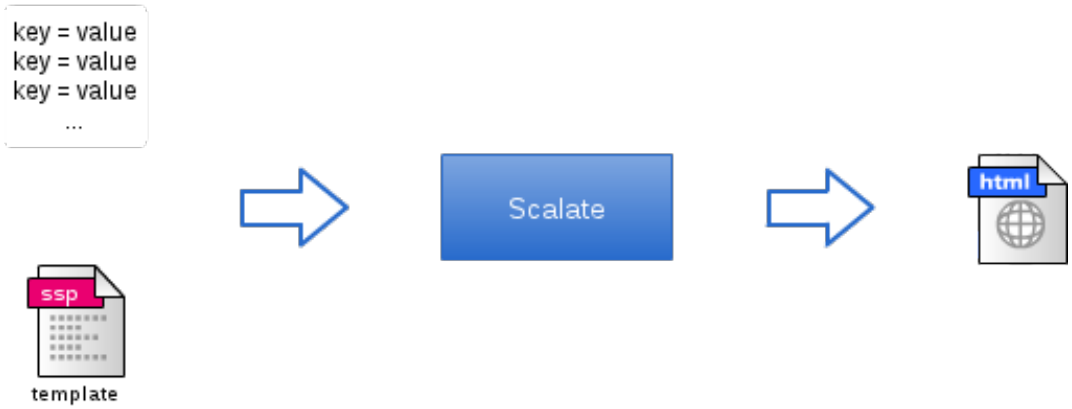

FIGURE 3.4 – Fonctionnement d'un moteur de template

L'utilisation d'un moteur de template permettra aussi d'ajouter des fonctionnalités de ce type au DSL, afin de permettre aux testeurs de se servir de templates Scalate dans leurs scénarios.

Chapitre 4

Conception de Gatling

4.1 Architecture de Gatling

4.1.1 Entrées & sorties

Gatling, comme tout programme informatique, accepte des données en entrée et renvoie des données en sortie. Les entrées sont créées par l'utilisateur de l'outil, tandis que les sorties sont destinées à être interprétées par ce même utilisateur. La figure 4.1 représente les entrées et sorties de Gatling, elles sont composées de :

- Entrées
  - **Scripts**. Les scripts sont des fichiers scala qui contiennent les scénarios et simulations que le testeur veut exécuter.
  - **Seeds**. Les seeds [1] contiennent les données qui seront utilisées par les feeders [2] dans les scénarios.
  - **Bodies**. Les Bodies sont les corps envoyés avec les requêtes de type POST ou PUT.
- Sorties
  - **Rapports**. Les rapports générés au format HTML et également exportés en CSV

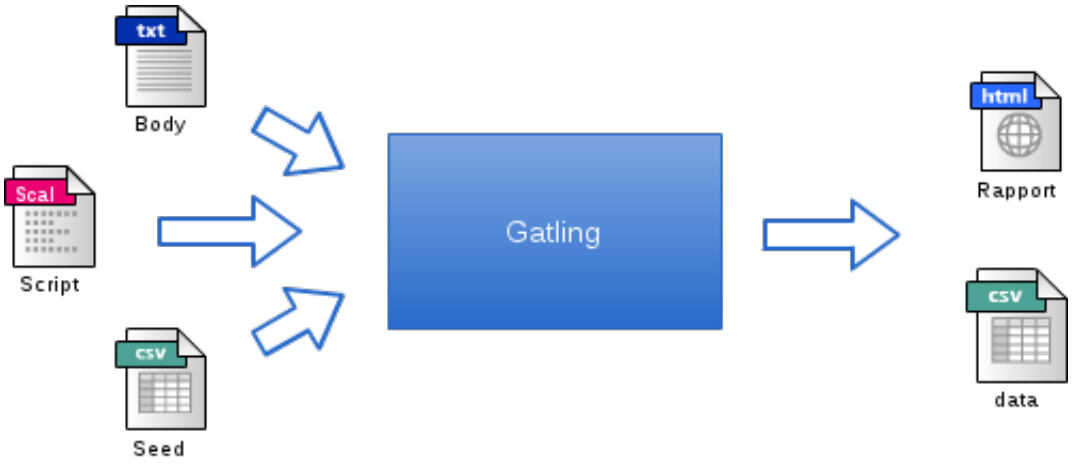

FIGURE 4.1 – Entrées et sorties de Gatling

1. Graines en français
2. Voir section 4.3.3 p.28

4.1.2 Application modulaire

Gatling est découpé en différents modules interagissant les uns avec les autres. Ce découpage permet de créer des APIs[3] internes, ce qui facilite l'ajout de fonctionnalités similaires comme, par exemple, l'ajout du support d'un protocole autre que HTTP. La figure 4.2 montre les modules principaux nécessaires au test d'applications web standard et présents dans la première version de l'application :

- **Gatling App**. C'est l'interface avec l'utilisateur de l'application, en l'occurence par la ligne de commande.
- **Gatling HTTP**. Ce module fournit les classes permettant de supporter le protocole HTTP.
- **Gatling Statistics**. Ce module fournit les classes permettant de générer les rapports.
- **Gatling Core**. Ce module contient toutes les fonctionnalités au coeur de l'application.

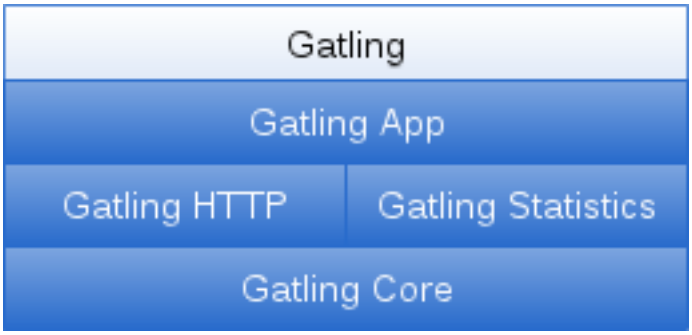

FIGURE 4.2 – Architecture de Gatling

4.1.3 Structure des dossiers

Afin de permettre à l'application comme à l'utilisateur de savoir où se trouvent les fichiers nécessaires à l'exécution d'une simulation, une structure de dossiers a été créée. Cette structure est représentée ci-après par la figure 4.3.

Le dossier **assets** contient des fichiers utiles à la génération des rapports. Les dossiers **bin** et **lib** contiennent les bibliothèques ainsi que l'exécutable qui permet de lancer Gatling. Le dossier **conf** contient le(s) fichier(s) de configuration de Gatling. Le dossier **results** contient les résultats de tests, rangés dans des dossiers portant comme nom la date et l'heure d'exécution du test.

Enfin, le dossier **user-files** contient les différents fichiers que l'utilisateur crée pour écrire ses scénarios. Le dossier **scenarios** contient les scripts écrits avec le DSL de Gatling. Le dossier **request-bodies** contient les corps de requête à être envoyés tels quels, alors que le dossier **template** contient des templates pouvant être complétés lors de l'exécution du scénario avant d'être envoyés. Le dossier **seeds** quant à lui contient les jeux de données utilisés par les feeders.

3. *Application Programmable Interface*. Les APIs sont des interfaces utilisables par d'autre logiciels afin d'utiliser celui qui les fournit

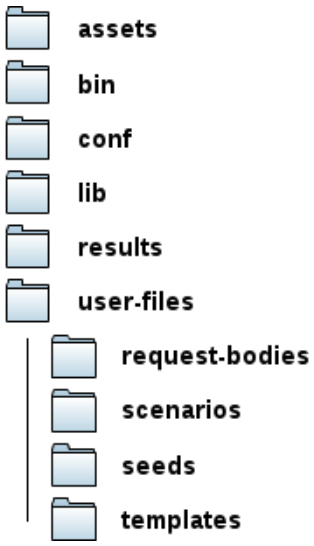

FIGURE 4.3 – Structure de dossiers de Gatling

4.2 Modules de l'application

4.2.1 Gatling App

Ce module permet de proposer un menu au testeur, et ensuite, de charger dynamiquement les scripts correspondant à la simulation qui doit être exécutée. Une fois la simulation déroulée, la génération des rapports est exécutée et l'application s'arrête une fois que c'est terminé. Le menu proposé à l'utilisateur est montré sur la figure 4.4.

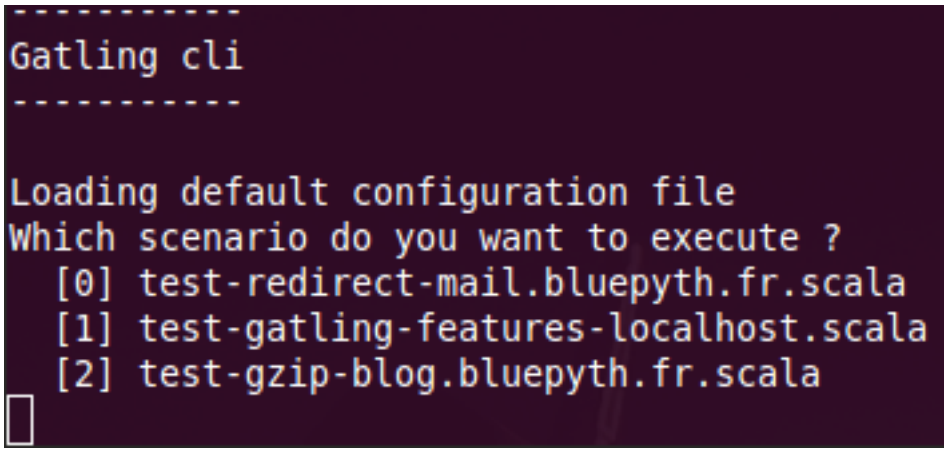

FIGURE 4.4 – Menu de sélection de la simulation à démarrer

4.2.2 Gatling Core

Présentation

Ce module contient toutes les fonctionnalités au coeur de Gatling, et sert aussi de base aux autres en fournissant une API commune à tous les modules, notamment en ce qui concerne l'action **Request** qui servira de base pour tous les protocoles implémentés à l'avenir. Le schéma de la figure 4.5 présente les divers composants de ce module.

FIGURE 4.5 – Composants de Gatling Core

Composants

Les composants de Gatling Core sont les suivants :

Scenario Il contient la définition d'un scénario et fournit l'API que doivent implémenter les modules qui souhaitent proposer une écriture de scénario qui leur est propre. Comme dans le reste de l'application, les composants **builder** servent à décrire le DSL, par souci de clarté, les autres ne sont pas représentés.

Action Au coeur du moteur, les actions sont en fait des acteurs au sens vu section 3.3.2.

Feeder Ce composant permet de lire les seeds que l'utilisateur a créées. Comme on le voit, il est possible de récupérer des informations depuis des fichiers CSV, TSV et SSV.

Provider Les providers sont les classes à la base des captures et des assertions. En effet, le fait d'avoir des providers dans Core permet de n'implémenter qu'une seule fois la recherche, mais de pouvoir l'utiliser dans tous les modules. Pour l'instant, il y a trois types de providers : RegExp (Expression Régulière), XPath et Range qui permet de vérifier qu'un nombre est bien compris entre deux valeurs [4].

Context Ce composant permet de stocker ou récupérer des valeurs du contexte.

Config Composant permettant de lire la configuration utilisateur qui se trouve dans le fichier gatling.conf.

Writer Ce composant enregistre l'évolution des simulations dans un fichier.

Processor Cette API permet de définir des processors. Un processor est une unité qui agit sur la réponse. On y retrouve typiquement les captures et assertions, mais d'autres pourraient être ajoutés dans le futur.

Runner Cette API permet elle de définir la façon dont va s'exécuter le scénario.

4. Ce provider permet de vérifier le statut des réponses entre 200 et 299 par exemple pour les réponses HTTP valides.

4.2.3 Gatling HTTP

Présentation

Ce module permet d'utiliser Gatling pour tester des applications qui utilisent le protocole HTTP afin de communiquer, ce sont majoritairement des applications web. Les verbes HTTP supportés par ce module sont PUT, POST, GET et DELETE, ce qui rend possible le test d'applications RESTful et des webservices REST. Le schéma de la figure 4.6 présente les divers composants de ce module.

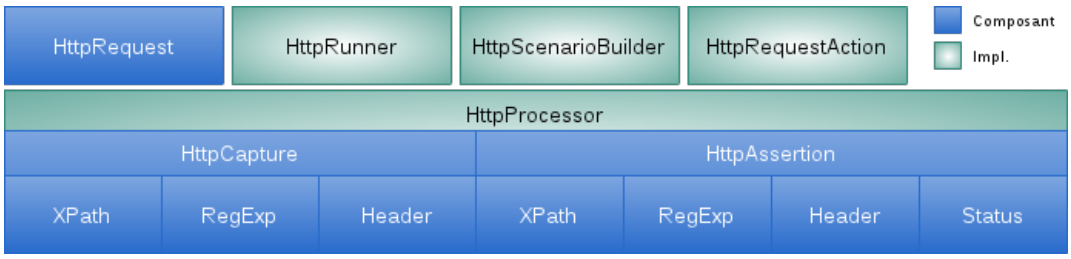

FIGURE 4.6 – Composants de Gatling HTTP

Composants

Les composants de Gatling HTTP sont les suivants :

HttpRequest Ce composant est un wrapper de la classe Request du client HTTP utilisé : AsyncHttpClient[12]. Elle permet, entre autres, de définir le DSL permettant de créer les requêtes HTTP.

HttpRunner Ce composant, comme expliqué dans la section 4.2.2, programme l'exécution des scénarios.

HttpProcessor Ce composant définit les captures et assertions qui pourront être utilisés sur les requêtes HTTP du scénario.

4.2.4 Gatling Statistics

Présentation

Ce module permet de générer les rapports de simulation et de calculer les statistiques nécessaires à ces mêmes rapports. Sa composition est présentée figure 4.7. Il existe actuellement trois entités différentes pour lesquelles des statistiques sont calculées : les sessions, les requêtes d'un point de vue global et des détails pour chaque requête.

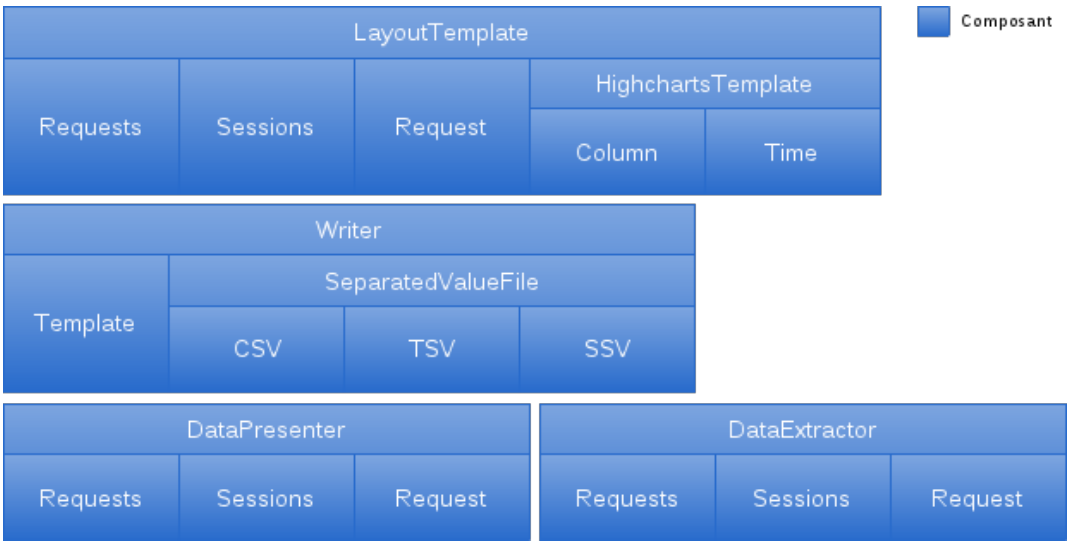

FIGURE 4.7 – Composants de Gatling Statistics

Fonctionnement

Les statistiques sont générées selon le modèle de la figure 4.8. Lors de l'exécution du scénario, le moteur enregistre les résultats de chaque requête dans un fichier. Ce fichier est ensuite analysé par un **extracteur de données** qui récupère et formate les données utiles à la génération d'un graphique. Ces données sont ensuite mise en forme par les **presenters** et enfin, un fichier de rapport est créé au format HTML grâce à l'utilisation de templates, comme vu section 3.5.3. En parallèle de la génération de rapports au format HTML, des fichiers au format TSV sont créés.

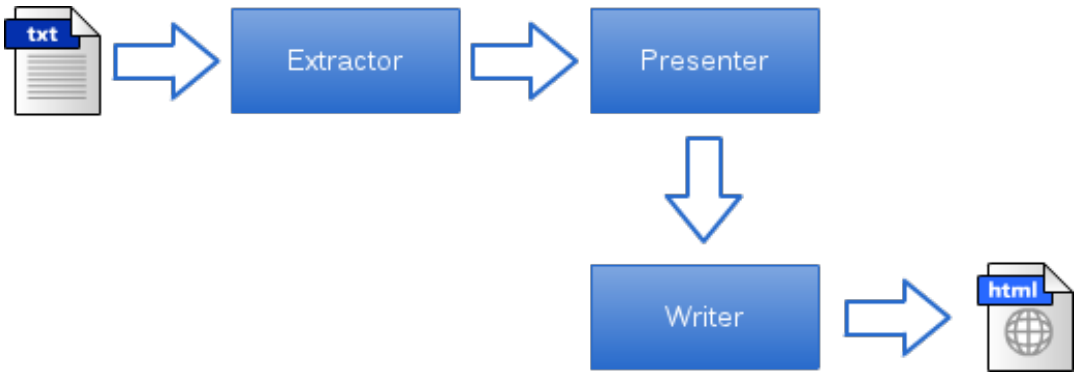

FIGURE 4.8 – Génération de statistiques

La figure 4.9 représente le résultat obtenu pour le détail d'une requête. On distingue deux graphiques : le premier indique le temps de réponse en fonction du temps, tandis que l'autre indique le nombre de requêtes par temps de réponse. Entre les deux graphiques, des données statistiques sont affichées telles que l'écart-type, la moyenne, la valeur maximale et minimale du temps de réponse, etc.

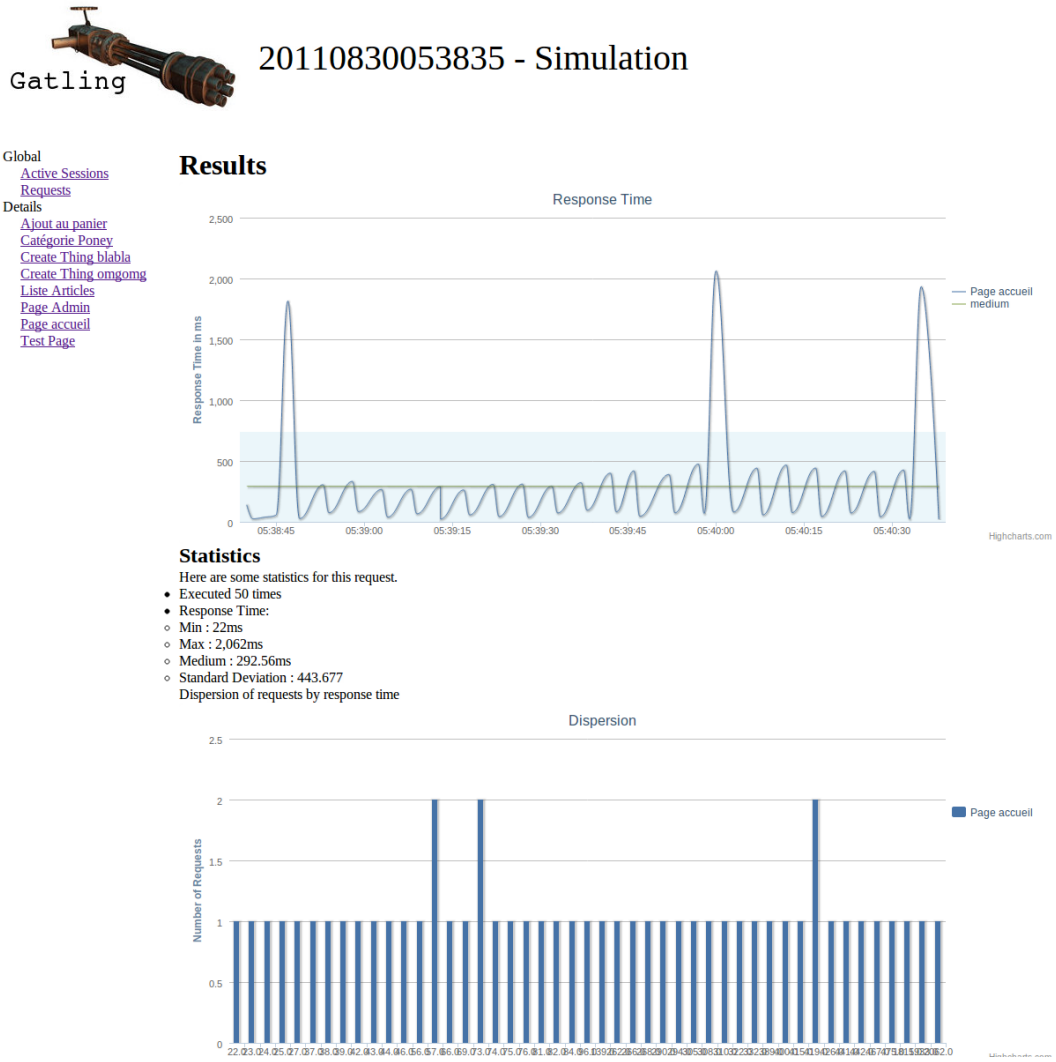

FIGURE 4.9 – Exemple de statistiques générées

4.3 Le DSL

4.3.1 Analyse d'un test de charge

Afin de créer un DSL simple à utiliser, une analyse été faite afin de repérer les éléments que contiennent un scénario et une simulation.

Définition d'un scénario

Afin de définir un scénario d'utilisation, la personne qui l'écrit doit pouvoir spécifier :
- Le nom du scénario ;
- Les requêtes faites par l'utilisateur, et pour chaque requête :
  - Le type de la requête (HTTP pour l'instant) ;
  - Le nom de la requête (pour facilement identifier la requête dans les statistiques) ;
  - Le verbe HTTP utilisé (POST, PUT, GET ou DELETE) ;
  - Les paramètres de la requête ;

— Le corps de la requête si possible ;
— Les pauses qui simulent le temps de lecture de l'utilisateur.

Exécution d'une simulation

Une fois les scénarios écrits, il faudra les exécuter. Le testeur doit pouvoir configurer cette exécution. Voici les éléments configurables :

— Le nombre d'utilisateurs simultanés pour ce scénario ;
— La rampe de démarrage des utilisateurs. Ceci permet de simuler un arrivage progressif des utilisateurs sur le site par exemple ;
— Le démarrage du scénario. En effet, plusieurs scénarios peuvent être exécutés durant une simulation, on peut donc spécifier à quel moment de la simulation le scénario doit débuter.

4.3.2 Définition du DSL

Les éléments que l'analyse a permis de découvrir représentent la base du DSL. Pour chaque élément, vous trouverez un exemple du DSL qui correspond. Le logiciel étant Open Source et destiné aux entreprises du monde entier, le DSL est en anglais.

Déclaration d'une requête HTTP

Le DSL permet de déclarer et configurer facilement des requêtes HTTP. Le code du listing 4.1 montre quelques unes des diverses possibilités offertes.

```
// Faire une requete de type GET sur l'url http://google.fr
get("http://google.fr")

// Faire une requete de type GET sur
// l'url http://google.fr?q=ma+recherche
get("http://google.fr") withQueryParam ("q", "ma+recherche")

// Faire une requete de type POST sur l'url http://monurl.com
// avec le corps : '{"mon_object": {"attribut": "valeur" }'
// et des headers specifiant le contenu : JSON
post("http://monurl.com")
    withBody """{ "mon_object" : { "attribut" : "valeur" }"""
    asJSON

// Faire une requete de type PUT sur l'url http://monurl.com
// avec le parametre de formulaire 'name' = 'John'
put("http://monurl.com") withParam ("name", "John")
```

Listing 4.1 – Déclaration d'une requête HTTP

Déclaration d'un scénario

L'exemple suivant, montré dans le listing 4.2, illustre la façon dont un scénario peut être créé via le DSL de Gatling. Un scénario basique permettant de tester une application web

contient des requêtes de type HTTP, ainsi que des pauses permettant de simuler le temps de lecture de l'utilisateur.

```
// Declarer un scenario nomme 'S1', constitue d'une requete
// sur le site de Google, attendre 1 seconde puis executer
// une requete de recherche.
scenario("S1")
  doHttpRequest("Accueil", get("http://google.fr"))
  pause(1)
  doHttpRequest("Recherche", get("http://google.fr")
    withQueryParam ("q", "keyword")
```

Listing 4.2 – Déclaration d'un scénario simple

Déclaration d'une simulation

La dernière étape, une fois le scénario écrit, consiste à le configurer pour l'exécuter. Le listing 4.3 montre de quelle façon est déclarée la configuration puis la demande d'exécution de la simulation.

```
// On declare un scenario
val stdUser = scenario("Standard") do...

// On cree une configuration pour ce scenario avec
// * 5 utilisateurs concurrents
// * 0 seconde de rampe
// * 0 seconde de delai au demarrage
val stdUserConf =
  configureScenario(stdUser)
    withUsersNumber 5
    withRamp 0
    startsAt 0
val execution = runSimulations(stdUserConf)
```

Listing 4.3 – Déclaration d'une simulation au complet

4.3.3 Fonctionnalités supplémentaires du DSL

Afin de simplifier l'écriture des scénarios et d'augmenter les possibilités du DSL, des fonctionnalités supplémentaires ont été ajoutées.

Captures et assertions

Afin de valider les requêtes, le testeur pourra utiliser des assertions sur la réponse HTTP, permettant de vérifier son code, ainsi que la présence de certains **headers** ou des expressions dans le corps de la réponse. Cela permettra par exemple de valider une requête de login avant d'exécuter le reste du scénario.

Les assertions sont basées sur les captures qui permettent de faire des recherches par expression régulière ou par XPath. Ces captures sont exécutées sur la réponse HTTP reçue

par le moteur de simulation. Chacun de ces **processeors** est disponible indépendamment de l'autre dans le DSL, le listing 4.4 montre comment ils sont utilisés.

```
scenario("S1")
  doHttpRequest(
    "Accueil",
    get("http://www.google.fr"),
    // On stocke "ou" dans "name"
    regexp("""Ab(.*)t""") in "name",
    // On verifie qu'on est bien sur google
    assertRegexp("""<img alt="(.*)" """, "Google")
  )
```

Listing 4.4 – Utilisation des captures et assertions

Contexte

Un contexte est propre à un seul utilisateur simulé par un scénario. La présence de ce contexte permet au testeur de faire passer des infos de requêtes en requêtes.

Le testeur pourra extraire des données des réponses HTTP afin de les stocker dans le contexte, cela se fera en utilisant des captures (ou des assertions), les valeurs seront ensuite accessibles dans le reste du scénario. Le listing 4.5 montre l'utilisation du contexte dans le cadre d'un scenario simple.

```
scenario("S1")
  doHttpRequest(
    "Set Value",
    get("http://google.fr"),
    regexp("""<img alt="(.*)" """) in "pagename")
  pause(1)
  doHttpRequest(
    "Use value",
    get("http://google.fr")
      withQueryParam ("q", FromContext("pagename")))
```

Listing 4.5 – Utilisation du contexte

Feeders (ou Seeders)

La simulation d'un grand nombre d'utilisateurs peut nécessiter l'utilisation d'un nombre aussi grand de comptes différents sur l'application testée. Pour permettre l'utilisation d'un compte différent par utilisateur simulé, Gatling propose des Feeders [5]. Ces éléments permettent d'utiliser une source de données telles qu'un fichier de type CSV pour y récupérer des informations à utiliser dans le scénario. Le listing 4.6 montre l'utilisation de ces Feeders dans le cadre d'un scénario simple.

```
// Declaration du Feeder
userCredentials =
  new CSVFeeder("user_cred", List("login", "password"))
```

5. ou Seeders, le nom n'étant pas encore figé

```
scenario("S1")
  doHttpRequest(
    "Login",
    post("http://mywebsite.com/login")
      // Declaration du Feeder pour cette requete
      withFeeder userCredentials
      // Utilisation des parametres du feeder
      withParam "login"
      withParam "password"
```

Listing 4.6 – Utilisation des Feeders

Conclusion

Gatling offre une nouvelle approche du test de charge grâce à des performances optimisées, une façon d'écrire les scénarios plus naturelle et un langage résolument proche du vocabulaire utilisé dans le domaine du test de charge. Publié sous licence Apache 2, Gatling est donc Open Source et disponible à l'adresse http ://github.com/excilys/gatling. La version actuelle pose des bases solides pour le développement de la suite des fonctionnalités prévues dans la roadmap, et j'espère qu'elle saura percer et devenir un outil incontournable pour le test de charge basé sur des scénarios d'utilisation.

Ce stage fut très intéressant et extrêmement enrichissant. La formation proposée m'a permis de découvrir de nouvelles technologies. La conception de Gatling m'a initié au domaine du test de charge que je ne connaissais pas, et m'a permis de participer à la communauté Open Source et de découvrir un nouveau langage de programmation.

L'implication de l'entreprise dans le développement personnel de ses collaborateurs m'a permis de découvrir des communautés telles que le Paris Java User Group, l'association JDuchess et le Paris Android User Group ; je suis également devenu Oracle Certified Java Programmer. Ravi d'intégrer cette société, je pourrai continuer à m'investir dans le développement de Gatling sous la bannière Excilys :-)

Bibliographie

[1] Excilys. Site web du groupe excilys. https ://www.excilys.com/.

[2] Excilys. Capico primaire. http ://www.capico.fr.

[3] Excilys. Gatling sur github. http ://github.com/excilys/gatling.

[4] Fondation Apache. Jmeter. http ://jakarta.apache.org/jmeter.

[5] SmartBear. Loadui. http ://loadui.org/.

[6] SmartBear. Soapui. http ://www.soapui.org/.

[7] Communauté Wikipedia. Modèle d'acteur. http ://fr.wikipedia.org/wiki/Modèle_d'acteur.

[8] Communauté Wikipedia. Modèle d'acteur [en]. http ://en.wikipedia.org/wiki/Actor_model.

[9] TypeSafe. Site du langage scala. http ://www.scala-lang.org.

[10] TypeSafe. Site de la bibliothèque akka. http ://akka.io.

[11] Développeurs de Scalate. Site de scalate. http ://scalate.fusesource.org.

[12] Jean François Arcand. Async http client sur github. https ://github.com/sonatype/async-http-client.