

I - Tests de performances

1. Rôles des tests et bénéfices attendus

Objectifs des tests de performances et de montée en charge :

- **Arbitrer des choix de technologies** (*comparer les perfs de plusieurs technologies et choisir la plus efficace*)
- **Qualification technique** (*vérifier que le minimum attendu est au moins assuré pour garantir un fonctionnement normal / corriger l'application en cas de problème(s)*)
- **Optimisation des ressources** (*détecter et enlever certains goulots d'étranglement pour ne pas sous utiliser les autres ressources matérielles / ne pas gaspiller*).
- **Anticiper** sur des besoins de **redimensionnement** (*nécessité d'agrandir un cluster ?,*)

Différents types de mesures et de tests

Type de mesures et de tests	Moment(s) adéquat(s)	Objectifs / caractéristiques
Profilage (<i>en interne dans une application ou dans un serveur d'applications</i>)	Développement Intégration (dans serveur d'application)	Détecter mauvaise programmation (algorithmes lents, mémoire mal gérée, ...)
Mesures externes des rendements/capacités et de temps de réponse	Développement Qualification Production	Vérifier un fonctionnement correct et/ou déterminer le seuil de saturation
Simulation de charge et <i>courbe de réponses</i>	Qualification	Pas besoin de stresser le serveur en production !

Éléments requis pour la mise en œuvre des tests

- *Simuler/Extrapoler correctement:*
Proche réalité/infrastructure de production
ou bien extrapolations justes/contrôlées.
- *Bien mesurer:*
Savoir utiliser les logiciels de mesures
- *Diagnostiquer correctement:*
**Savoir interpréter les résultats (avoir des
éléments de comparaison)**
- *Résoudre les problèmes: Savoir comment agir .*

Règle (assez générale) des 80/20

- Environ **20% d'efforts** pour obtenir **les
premiers et principaux résultats (80%)**
- **80% d'efforts** pour difficilement obtenir
les derniers résultats (20%)

2. Types de tests (principes , intérêts)

2.1. Différents types d' analyses (vue externe, sondes, ...)

L'analyse des performances comporte divers aspects que l'on peut ranger dans les catégories suivantes:

- Les **mesures non intrusives**: Il n'y a pas de sonde à l'intérieur du serveur. Les traitements ne sont donc pas alourdis mais on ne peut pas connaître les réglages internes à optimiser.
Exemple: **test de montée en charge, Analyse du temps de réponse , analyse du débit supporté (throughput / TPS)**.
- Les **mesures intrusives**: On paramètre le serveur (et/ou l'application) de façon à ce qu'il(s) nous envoie(nt) régulièrement des mesures détaillées sur tel ou tel aspect. Ces sortes de sondes ont tendances à alourdir les traitements donc à fausser légèrement les mesures mais on peut en contrepartie bien cerner ce qu'il est nécessaire d'optimiser (ex: mémoire ou taille d'un pool de connexions, ...).

P.M.I. signifie *Performance Monitoring Infrastructure* .

2.2. Activation des mesures et conséquences

L'activation des mesures (sondes) a tendance à alourdir les traitements du serveur et donc à fausser les mesures elles mêmes (*principe d'incertitude d'Einsenberg*).

On aura donc tout intérêt à effectuer que des mesures ponctuelles et bien ciblées.

3. Profiling (application / serveur)

Profilage (interne) d'une application

Principe: *récolter* via des "*sondes*" tout un tas de *mesures internes précises* de façon à *associer/affecter* précisément des "*coûts*" (*en temps d'exécution ou ...*) aux différentes *fonctions* et sous fonctions du code.

--> **objectif**: *repérer* les parties "*traitements longs*" et "*attentes*" et *essayer de les expliquer et/ou optimiser*.

NB: *il faut avoir quelques éléments de comparaison pour pouvoir interpréter les résultats (bon ou pas ?)*

4. Mesures externes (non intrusives)

Mesures de capacités et de temps de réponse

Principe: effectuer des *mesures à l'extérieur de l'application* (et/ou du serveur logiciel) de façon à connaître ses *capacités (temps de réponse , % d'utilisation CPU , ...)* .

(Exemple: envoyer des requêtes SQL vers un SGBDR(Oracle/Mysql) ou bien des requêtes HTTP vers une application Web (Php ou Asp ou Java_EE) et mesurer le nombre de secondes (ou ms) qui s'écoulent en moyenne avant d'obtenir une réponse.)

Certaines mesures ne sont pertinentes que si le serveur travaille vraiment normalement (déjà entièrement initialisé et bien sollicité par des clients simultanés).

5. Simulation et test de charge

Simulation de charge

Principe: *Envoyer* via un logiciel spécialisé un **paquet de requêtes à un rythme assez élevé** vers un serveur logiciel de façon à **simuler la charge induite par un nombre bien défini de clients simultanés**.

Il s'agit en quelque sorte de "stresser" un peu un certain serveur pour observer son comportement et ses capacités.

Objectif: effectuer différents "tirs" en faisant varier le nombre de requêtes simultanés et synthétiser les mesures résultantes (temps de réponse, ...) au sein d'une courbe de réponses. *Charge maxi supportée = avant saturation*

Pour effectuer des mesures à un rythme assez régulier et pour que l'outil de stress ne fausse pas les mesures, il faut impérativement faire tourner l'utilitaire de test de montée en charge (ex: JMeter) sur un ordinateur différent de celui où s'exécute le serveur à tester.

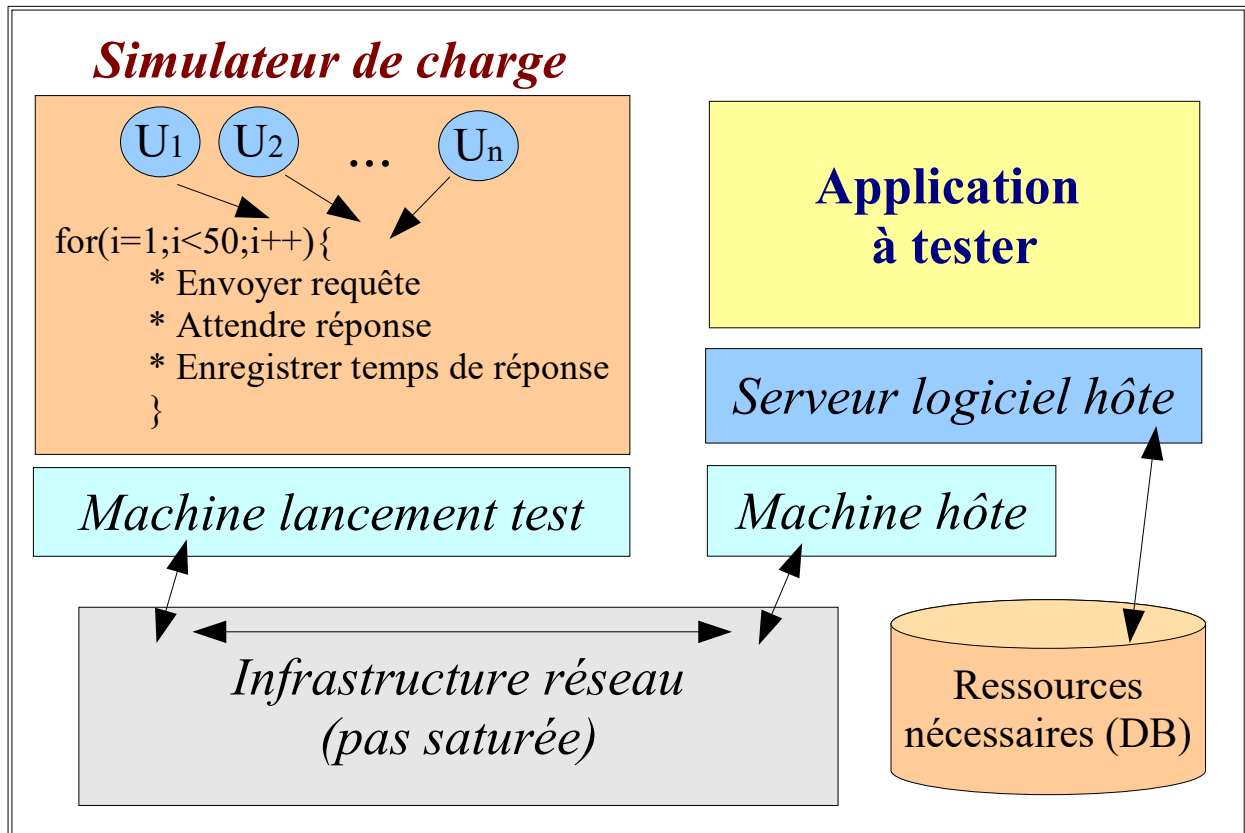
Dans une configuration de test idéale, chacun des éléments suivants doit fonctionner sur une machine séparée:

- JMeter ou équivalent (test de montée en charge , analyse des rythmes et temps des réponses).
- Le serveur d'application (Tomcat ou JBoss ou ...).
- Le SGBDR (MySQL , Oracle , ...) avec les bases de données applicatives.

Etant programmé en java (et devant donc gérer des phases de ramassage de la mémoire [**Garbage Collector**]" l'utilitaire **JMeter** peut quelquefois manquer de régularité et les mesures seront alors assez approximative .

Finalement , pour obtenir des mesures réalistes , il est vivement conseillé d'introduire une certaine variabilité au sein des requêtes envoyées au serveur pour que ce dernier ne se contente pas de nous renvoyer tout le temps le même résultat (éventuellement en cache ou construit d'une façon trop bien optimisée).

6. Topologie de la plateforme de test (charge)



NB: Certains produits sophistiqués sont capables de :

- déclencher simultanément des "tirs de requêtes" émis depuis plusieurs machines clientes
- synthétiser globalement des résultats obtenus .

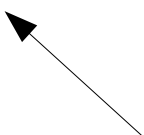
Ceci permet quelquefois d'ajouter un peu plus de réalisme et est avant tout utile si les requêtes chargent beaucoup le réseau ou si le simulateur de charge est installé sur une machine qui n'est pas assez puissante pour solliciter suffisamment le serveur (ou le cluster de serveurs).

7. Configuration des requêtes nécessaires à la simulation

Requêtes pour simulation de charge

- Bien formulées (*selon protocole, ...*)
- Techniquement acceptées (*avec cookies HTTP, champs cachés, ...*)
- Dans le bon ordre (*index, page1, pageN*)
- Avec arguments variés
- ...

Mise au point par *enregistrement d'une séquence/session réelle* et/ou *configuration pas à pas*



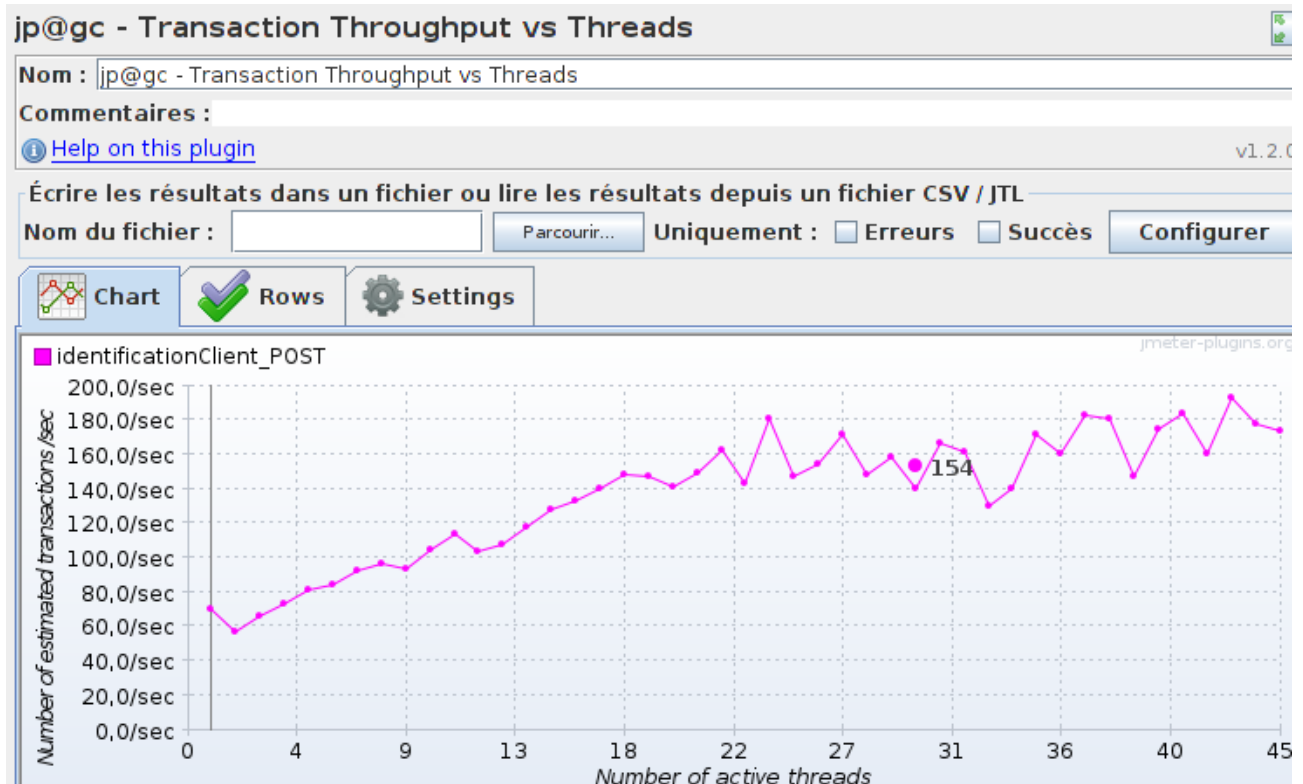
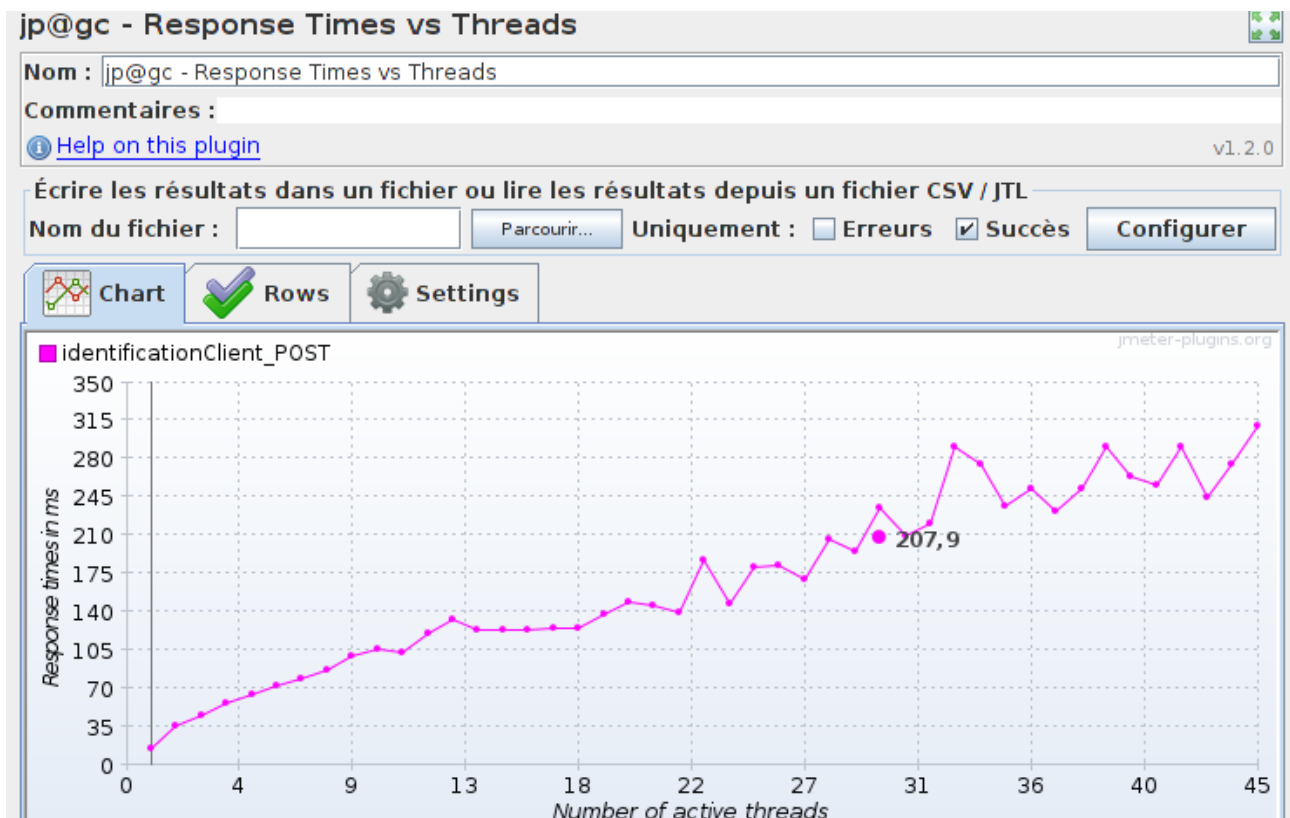
==> Pas si simple (voir même assez complexe avec certains outils).

8. Synthèse des résultats obtenus

Devoir faire soit même un rapport (avec courbe de réponse selon la charge) ou bien paramétrer un automatisme qui le génère automatiquement constitue l'une des différences importantes entre un produit d'entrée de gamme et un produit sophistiqué .

L'utilitaire open source de référence "**JMeter**" de la *fondation Apache* comporte un paquet de plugins dit "*extra*". Les **plugins** "*TransactionPerSecond vs Thread*" et "*ResponseTime vs Thread*" permettent d'obtenir simplement des rapports graphiques synthétiques (si l'on paramètre bien un démarrage progressif du nombre de threads dans le groupe d'unité d'exécution).

Exemples :



Dans l'exemple ci dessus , on visualise assez bien un **seuil de saturation** à partir d'environ 20 threads actifs en parallèle (associé à environ 150 TPS) .

9. Présentation de JMeter

JMeter est un produit **open source** de la communauté **Apache**.

JMeter est capable de **simuler une certaine charge** tout en **récoltant des mesures essentielles** (temps de réponses , débits,) .

JMeter est développé en **java** et est donc:

- multiplateforme (windows , linux, ...)
- soumis à certains cycles de "GC/ramasse-miettes" qui ont tendance à introduire un certain niveau d'approximation dans les mesures recueillies (==> ordre de grandeur correct mais précision limitée) .
- Simple à utiliser (interface graphique perfectionnée).

Principales fonctionnalités du produit "JMeter":

- requêtes HTTP , SOAP , ...
- requêtes SQL (JDBC), ...
- requêtes LDAP , FTP , TCP, ...
- requêtes JMS,

- résultats en graphes , tableaux, rapports,
- vérifications via assertions, comparaisons, sauvegardes fichiers,

- Timer, threads(clients simultanés simulés)
- Pré et post-processeurs (extraction de valeurs via reg-expr, ...)
- Variables, boucles , ...

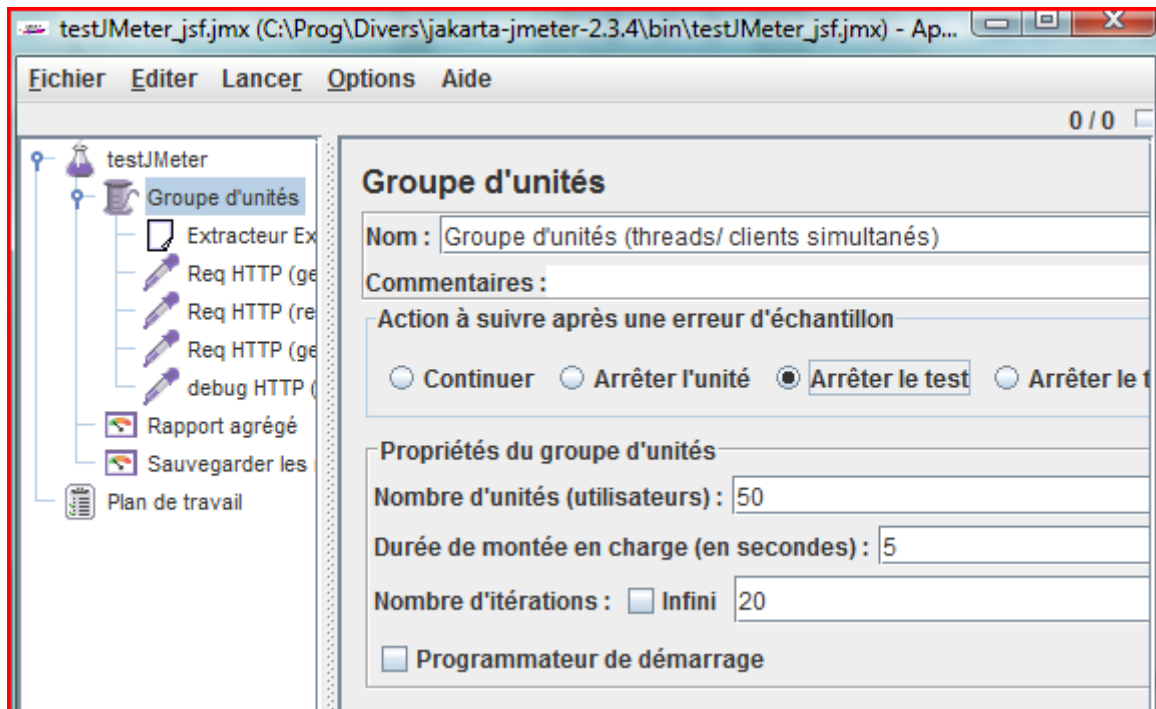
- Gestion automatique des cookies HTTP (utile entre autres pour identifier les sessions des clients simultanés simulés)
- ...

10. Configuration élémentaire de JMeter

Commencer par renommer le plan de test (ex: MyApp ou TestXY)

10.1. Configuration d'un groupe d'unités (clients simultanés)

Ajouter / Groupe d'unités [Add / Thread group]



10 Threads = 10 clients simultanés

Période de chauffe [*Ramp-up period*] = 5

==> les threads démarrent un par un toutes les $5 / 10 = 0,5$ s.

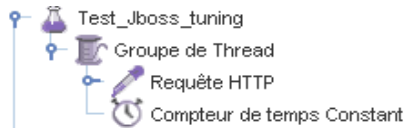
Une fois démarré, chaque thread enverra 20 requêtes (*Loop Count*). Entre chaque requête, chacun des threads attendra par défaut 0 ms.

Pour paramétrer (si besoin) une pause éventuelle (ex: **1000 ms**) entre les requêtes consécutives envoyées par un même thread il faut alors incorporer (sous le groupe de Thread) un **Compteur de temps fixe** [*Timer*].



10.2. Configuration d'une requête à lancer (plusieurs fois)

Sous ce "Groupe d'unités" ==> **Ajouter / Echantillon / Requête HTTP**



Requête HTTP

Nom: Requête HTTP / Conversion

Serveur Web

Nom ou IP du Serveur: localhost

Numéro de Port: 8080

Requête HTTP

Protocole: Méthode: ☒ GET ☐ POST

Chemin: deviseWeb/convlt.jsp ☐ Rediriger Automatiquement ☒ Suivre

Envoyer les Paramètres Avec la Requête:

Nom:	Valeur
somme1	100
monnaie1	Euro
monnaie2	Yen

10.3. configuration de la récupération des résultats (synthèse, courbe, ...)

==> Sous la "Requête HTTP" ou bien sous "le groupe d'unités"

==> **Ajouter / Récepteur / Résultats Graphiques**

==> **Ajouter / Récepteur / Rapport Agrégé (tableau récapitulatif)**

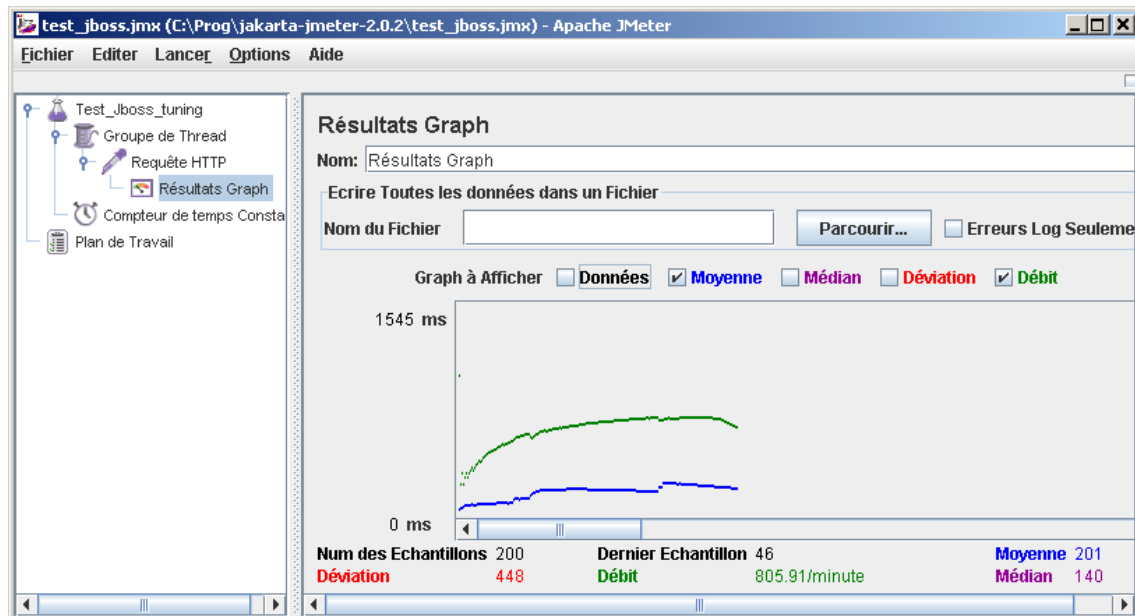
10.4. (re-)lancement du test et observation des résultats

Sauvegarder le "Test Plan" .

Ré-initialiser à zéro des courbes et résultats via "**Lancer / Nettoyer tout** [*Run / Clear All*]"

Activer la simulation de montée en charge via "**Lancer / Démarrer** [*Run / Start*]"

==>



Rapport Agrégé

Nom: Rapport Agrégé

Ecrire Toutes les données dans un Fichier

Nom du Fichier: Parcourir... ☐ Erreurs Log Seulement

URL	Count	Average	Min	Max	Error%	Rate
Requête HTTP	40	41	0	140	0,00%	4,6/sec
TOTAL	40	41	0	140	0,00%	4,6/sec

11. Très important: tester tout d'abord le test !!!

Si l'on ne vérifie pas l'exactitude des paramétrages au niveau des tests, on peut alors confondre une réponse normale avec une réponse de type "message d'erreur" retournée par le serveur. Des temps de réponses liés à des réponses négatives (messages d'erreurs) ne sont évidemment pas significatifs.

Pour être certain que la réponse retournée par le serveur est correcte/normale, il suffit de procéder de la manière suivante (durant la phase de mise au point):

- 1) diminuer (temporairement) au maximum le nombre de requêtes qui seront envoyées:

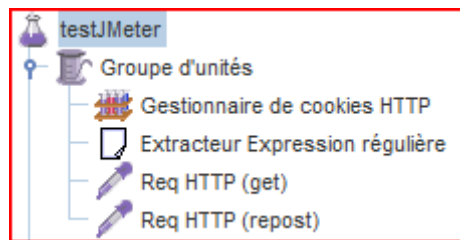
- 2) ajouter un récepteur de type "*Sauvegarder les réponses vers un fichier*"

- 3) **lancer le test et vérifier le contenu du ou des fichier(s) généré(s)**
- à refaire à chaque nouveau re-paramétrage des tests (pour vérifier une non-régression)
- 4) une fois la phase de mise au point terminée, il suffit de **rendre inactif** le récepteur de type "*Sauvegarder les réponses vers un fichier*" pour ne pas ralentir inutilement les tests. En jouant ainsi sur le menu contextuel "activer/désactiver", on peut basculer rapidement d'un mode "debug" vers un mode "test réaliste" et vice-versa. Il faudra également penser à ré-augmenter les nombres de boucles et d'utilisateurs simultanés au sein du groupe d'unité après avoir achever la phase de "debug".

12. Gestion correcte des sessions HTTP (cookies)

De façon à ce que JMeter se comporte exactement comme un navigateur internet au niveau de la gestion des sessions utilisateurs, il faut **absolument ajouter** la configuration "**Gestionnaire de cookies HTTP**" en dessous du niveau "groupe d'unités" .

Rien d'important n'est à paramétrer à ce niveau, mais le "**Gestionnaire de cookies HTTP**" doit simplement être présent .



13. Successions de requêtes http liées entre elles

Un utilisateur utilise généralement une application web en passant par une suite logique de pages (accueil , menu1, menu2 , sélection/recherche , résultats , action , statut ,).

Ces différentes pages activées sont souvent liées entre elles par un ou plusieurs champs cachés (véhiculant un id "utilisateur" ou bien "un état de session" ou "...").

Ces "id" sont générés dynamiquement et ne sont donc pas constants (différents pour chaque utilisateur et pour chaque session).

JMeter offre heureusement un moyen pour "récupérer et ré-injecter" ces "id" véhiculés sous forme de champs cachés **via un processeur d'extraction d'expressions régulières** (ou bien Xpath).

13.1. Principe du "repost-hidden"

repost_hidden.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>repost_hidden</title>
</head>
<%
String chX=request.getParameter("x");
String chLast=request.getParameter("last");
int x=1;
int last=1;
if(chX!=null) x=Integer.parseInt(chX);
if(chLast!=null)
    try{ last=Integer.parseInt(chLast);}
    %>
```

```

    catch (NumberFormatException ex) { last=0; }
    int y=last*x;
    %>
    <body>
        <form>
            x=<input type="text" name="x" /> <br/>
            <input type="hidden" name="last" value="<%=y%>" />
            <input type="submit" value="mult by last value"/>
        </form>
    </body>
</html>
x=<%=chX%><br/>
last=<%=chLast%><br/>
y=last*x=<%=y%> (new last)
</body>
</html>

```

Cet exemple de page jsp montre l'utilisation d'un champ caché permettant de mémoriser une certaine valeur entre plusieurs appels consécutifs émis depuis le même utilisateur.

Résultats des ré-appels successifs:

<input type="text" value="x=3"/> <input type="button" value="mult by last value"/>	<input type="text" value="x=2"/> <input type="button" value="mult by last value"/>	<input "="" type="text" value="x="/> <input type="button" value="mult by last value"/>
x=null last=null y=last*x=1 (new last)	x=3 last=1 y=last*x=3 (new last)	x=2 last=3 y=last*x=6 (new last)

Nouvelle valeur de y = ancienne valeur multipliée par le x saisi .
L'ancienne valeur (last) est véhiculée via un champ caché (aller/retour "serveur-client-serveur").

```
<input type="hidden" name="last" value="1 , 3 puis 6"/>
```

Test JMeter associé:



Paramétrage de la première requête (en mode GET)

Requête HTTP

Nom : Req HTTP (get)

Commentaires :

Serveur web

Nom ou adresse IP : localhost Port : 8080

Requête HTTP

Protocole (défaut http) : Méthode : GET Encodage du c

Chemin : testJmeter/repost_hidden.jsp

☒ Rediriger automatiquement ☐ Suivre les redirections ☒ Connexions persistantes

Envoyer les paramètres avec la requête :

Nom :	Valeur :

Paramétrage de l'extracteur d'expression régulière (post response processor) :

Extracteur Expression régulière

Nom : Extracteur Expression régulière

Commentaires :

Champs réponse à cocher:

☒ Corps ☐ Corps (non échappé) ☐ Entêtes ☐ URL ☐ Code de

Nom de référence : last

Expression régulière : <input type="hidden" name="last" value="(.)"/>

Canevas : \$1\$

Correspond au num. (0 pour Aléatoire) : 1

Valeur par défaut : 0

==> la valeur du champ caché est ainsi récupérée dans la variable "last" ==> `${last}`

Cette valeur est ensuite ré-injectée dans les requêtes ultérieures:

Requête HTTP

Nom : Req HTTP (repost)

Commentaires :

Serveur web

Nom ou adresse IP : localhost Port : 8080

Requête HTTP

Protocole (défaut http) : Méthode : POST Encodage :

Chemin : testJmeter/repost_hidden.jsp

☒ Rediriger automatiquement ☐ Suivre les redirections ☒ Connexions persistantes

Envoyer les paramètres avec la requête :

Nom :	Valeur :
x	2
last	\${last}

13.2. Cas concret du framework JSF

Dans le monde Java, les deux frameworks "WEB" les plus utilisés sont STRUTS et JSF. Le framework **JSF** (Java Server Faces) aujourd'hui en standard dans JEE5 utilise systématiquement la logique du "repost-hidden" avec un champ caché technique appelé "**ViewState**" :

exemple :

```
<form ...> .... <input type="hidden" name="f_SUBMIT" value="1" />
<input type="hidden" name="javax.faces.ViewState" id="javax.faces.ViewState"
value="C6+wc1LPa1t7RmBc0gq2x70OkLeUW00e0N6oI3nSK68L7t2CDtFOklqNvxrafMIhcH
TeV/zfHwOSd9Hr8zDSgA==" /> </form>
```

Extracteur Expression régulière

Nom : Extracteur Expression régulière

Commentaires :

Champs réponse à cocher :

☒ Corps ☐ Corps (non échappé) ☐ Entêtes ☐ URL ☐ Code de réponse ☐ Message de réponse

Nom de référence : jsfViewState

Expression régulière : <input type="hidden" name="javax.faces.ViewState" id="javax.faces.ViewState" value="(.*?)"/>

Canevas : \$1\$

Correspond au num. (0 pour Aléatoire) : 0

Valeur par défaut : 0

expression régulière :

Pour JSF ancien :

```
<input type="hidden" name="javax.faces.ViewState" id="javax.faces.ViewState" value="(.*?)"/>
```

Pour JSF récent :

```
<input type="hidden" name="javax.faces.ViewState"
id="j_id__v_0:javax.faces.ViewState:1" value="(.*?)"/>
```

Le contenu de la variable *jsfViewState* est ensuite ré-injectée dans les requêtes ultérieures :



Requête HTTP

Protocole (défaut http) : Méthode : **POST**

Chemin : testJmeter/page1.jsf

☒ Rediriger automatiquement ☐ Suivre les redirections ☒ Connexion

Envoyer les paramètres

Nom :	
f.a	2
javax.faces.ViewState	\${jsfViewState}
f.username	didier
f.b	3
f_SUBMIT	1
f.doAddition	additionner

avec

Encodage	Inclure égale ?
<input type="checkbox"/>	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
<input type="checkbox"/>	<input checked="" type="checkbox"/>

pour le paramètre *javax.faces.ViewState* qui comporte quelquefois dans sa valeur un caractère "+" ne devant pas être transformé en caractère espace (" ") par les mécanismes d'encodage HTML.

NB:

- Le paramètre f_SUBMIT (valant "1") est un autre champ caché des mécanismes JSF.
- Il ne faut pas oublier d'indiquer le paramètre lié au bouton poussoir (<h:commandButton /> JSF / <input type="submit" ../> HTML) ici "f.doAddition" valant "additionner" .
- Une gestion des sessions (via un gestionnaire de cookies HTTP ou ...) est absolument nécessaire pour que JSF fonctionne correctement.

Remarque importante (pour JSF):

De façon à ce que les noms des champs du formulaire html soient intelligibles (ex: "f.a" , "f.username") il faut absolument que le développeur précise une valeur au niveau de l'attribut id facultatif des balises de JSF:

exemple:

```
<%@ page language="java" contentType="text/html"%>
<%@ taglib prefix="f" uri="http://java.sun.com/jsf/core"%>
<%@ taglib prefix="h" uri="http://java.sun.com/jsf/html"%>
```

```

<html><head><title>page1 (jsf)</title></head>
<body>
<f:view>
  <h3> page1.jsp (jsf) avec h:form et re-post automatique en mode post
</h3>
  <h:form id="f">
    username (sessionScope) = <h:inputText id="username"
                                value="#{myJsfbBean.mySession.username}" /> <br/>
    a (requestScope) = <h:inputText id="a" value="#{myJsfbBean.a}" /> <br/>
    b (requestScope) = <h:inputText id="b" value="#{myJsfbBean.b}" /> <br/>
    <h:commandButton id="doAddition" value="additionner"
                      action="#{myJsfbBean.additionner}" />
  </h:form>
</f:view>
</body>
</html>

```

Sans l'information `id="..."`, les champs du formulaire ont des identifiants par défaut qui sont très peu compréhensibles: `<input id="j_id_jsp_85527204_1:j_id_jsp_85527204_3" name="j_id_jsp_85527204_1:j_id_jsp_85527204_3" type="text" value="0" />` !!!!

14. Configurations diverses

14.1. Test de charge (SQL) vers un serveur de base de données

NB: **JMeter** peut également effectuer des tests de montée en charge sur un SGBDR (MySQL, Oracle, ...). Il faut penser à ajouter le driver JDBC adéquat au CLASSPATH de JMeter [*Archives ".jar"* à déposer dans *jakarta-jmeter-2.x.y\lib*]. Ce type de tests (**JDBC Request**) est très pratique pour effectuer des ajustements au niveau des pools de connexions JDBC.

14.2. Variété au niveau des requêtes pour plus de réalisme

Si l'on déclenche en boucle une requête avec toujours les mêmes valeurs de paramètres en entrée, le serveur peut éventuellement optimiser ses traitements internes (via des caches) et retourner extrêmement rapidement la même réponse. Pour mieux simuler des clients simultanés envoyant des requêtes un peu différentes on a alors besoin d'injecter des valeurs variables dans les requêtes.

L'introduction d'un **contrôleur logique de type forEach** (en tant que *parent d'une requête*) permet de répéter celle-ci avec une valeur de paramètre variable (exemple: `${forEachOutputVar}`).

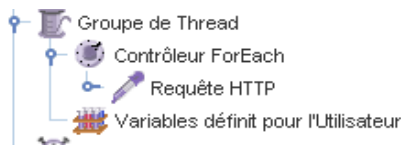
Requête HTTP

Protocole: Méthode: ☒ GET ☐ POST

Chemin: ☐ Rediriger Automatiquement ☒ Suivre

Envoyer les Paramètres Avec la Requête:

Nom:	Valeur
log_level	\${forEachOutputVar}



Contrôleur ForEach

Nom:

Préfix du paramètre en entrée

Nom du paramètre en sortie

La boucle ForEach sera effectuée sur les variables suivantes (*_1, _2, _3, ..., _n*) :

Variables définit pour l'Utilisateur

Nom:

Variables définit pour l'Utilisateur

Nom:	Valeur
forEachInputVar_1	TRACE
forEachInputVar_2	WARN

Le Listener "**voir l'arbre des résultats**" est dans ce cas très pratique pour contrôler les requêtes et les réponses .

14.3. Plugins utiles

JMeter peut être enrichi par certains plugins.

Le site officiel de JMeter comporte quelques paquets de plugins.

Le zip "**JmeterPlugins-Extras-1.2.0.zip**" (qu'il suffit d'extraire (en ajout) dans le répertoire de JMeter) comporte quelques plugins intéressants (dont "**TPS vs Threads**" et "**ResponseTime vs Threads**").

NB : bien penser à cliquer sur "*configurer*" et cocher "*nombre d'unités actives*"

et dimensionner au mieux la période de chauffe (ramp up period) du groupe d'unités pour que les threads puissent démarrer progressivement (via "plugin "*Stepping Thread Group*" ou ...).

14.4. JMeter à la source:

Pour télécharger **JMeter** ou approfondir certains de ses aspects , l'url de départ est :

<http://jmeter.apache.org/>

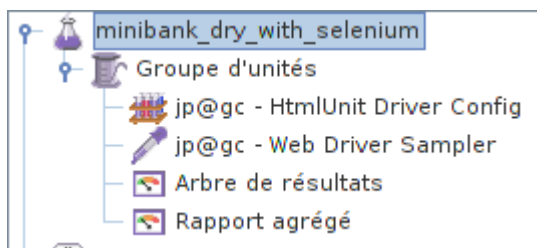
14.5. Plugin "JMeterPlugins-WebDriver-1.2" pour "selenium"

En téléchargeant et extrayant le contenu de *JmeterPlugins-WebDriver-1.2.0.zip* dans le répertoire de JMeter, on peut installer un plugin permettant de déclencher simplement depuis JMeter une séquence "selenium".

L'intérêt principal de ce plugin tient dans le fait qu'une séquence sélénium peut être simplement enregistrée via le plugin "selenium-ide" pour le navigateur "firefox". (voir annexe "selenium")

Mode opératoire :

- Installer si nécessaire le plugin "selenium-ide" dans firefox et utiliser celui-ci pour **enregistrer une séquence d'utilisation d'une suite de pages WEB** (navigation, soumission de formulaire, ...).
- Rejouer la séquence pour la vérifier et **exporter** le test sous les formats "*Java / Junit4 / WebDriver*" et "*python2 / unittest/ WebDriver*"
- Au sein d'un projet de test **jMeter**, configurer une arborescence classique ressemblant à celle-ci:



NB : on peut (au choix) utiliser "*firefox Driver Config*" ou bien "*HtmlUnit Driver Config*" (en

prenant soin de supprimer les anciens ".jar" selon la documentation du plugin).

- La configuration la plus importante se situe au niveau de "**web Driver Sampler**". Il faut renseigner un script (en langage javascript) permettant de piloter un navigateur web ("firefox" ou "chrome" ou "..." ou l'invisible "htmlUnit"). Ce script peut en grande partie se rédiger en effectuant un copier/coller partiel de l'enregistrement "java" ou "python" du test selenium et en **adaptant** la syntaxe.

Exemple de script attendu par le plugin "Web Driver" de JMeter :

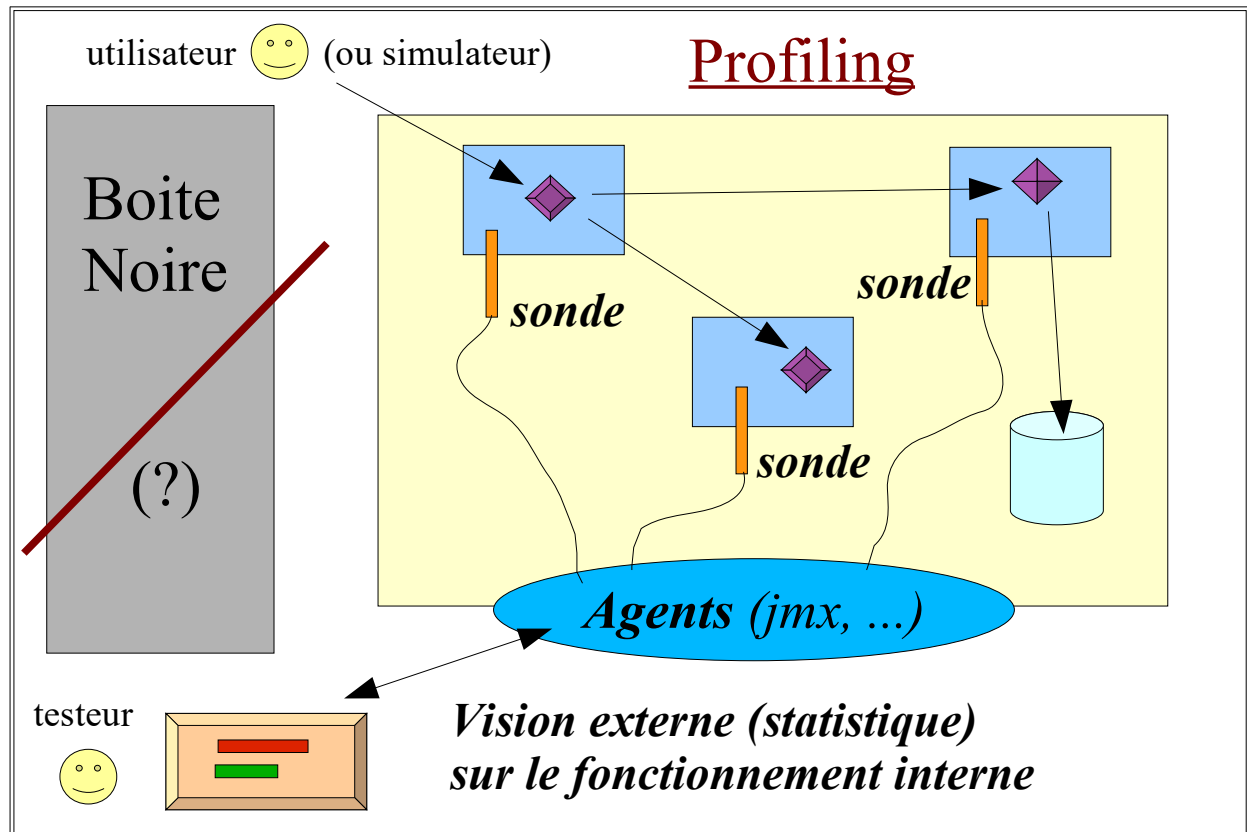
```
WDS.browser.get('http://localhost:8080/minibank-dry/')
WDS.sampleResult.sampleStart() //Start capturing the sampler timing
WDS.browser.findElementByLinkText("identification client").click()
WDS.browser.findElementById("identification:numClient").clear()
WDS.browser.findElementById("identification:numClient").sendKeys(["1"])
WDS.browser.findElementById("identification:btnIdentification").click()
WDS.sampleResult.sampleEnd() //Stop the sampler timing
if(WDS.browser.getTitle() != 'listeComptes.jsp') { // Verify the results:
    WDS.sampleResult.setSuccessful(false)
    WDS.sampleResult.setResponseMessage('Expected title to be listeComptes.jsp')
}
```

exemple d'enregistrement/export partiel (au format "python") effectué par selenium-IDE :

```
driver.get(self.base_url + "/minibank-dry/")
driver.find_element_by_link_text("identification client").click()
driver.find_element_by_id("identification:numClient").clear()
driver.find_element_by_id("identification:numClient").send_keys("1")
```

Attention : la version java/javascript de sendKeys() attend un tableau de string → ajouter des `[]`.

15. Principe du "profiling"



16. JaMon (Java Api for Monitor)

Il existe depuis longtemps une API open source java spécialisée dans les mesures des performances. Cette api s'appelle "**JaMon**" (*Java Api for Monitor*). Elle est simple à utiliser et peut facilement être associée à d'autres technologies java (spring, ejb3 , ...).

Nb : il existe une autre api java du même genre (*JMeasurement*) . Cependant *JMeasurement* semble être plus complexe et moins au point que *JaMon* .

Nb2 : l'api "**Commons-Monitoring**" (Counter, Gauge, Stopwatch, ...) de la fondation **Apache** est plus récente et pour l'instant un peu plus light/limitée. **Commons-Monitoring** reprend cependant petit à petit la même logique et les mêmes fonctionnalités que jamonapi et deviendra peut être le futur standard.

16.1. Intérêt d'une api compatible avec les api de logs

Le principal intérêt d'une api de mesures de performances (telle que jamon) réside dans le fait de pouvoir être utilisé à tous les stades (développement , intégration, recette , production) sans dépendre d'un environnement spécifique (pas de version d'eclipse imposée).

16.2. Utilisation élémentaire de l'api "Jmon"

Placer la librairie **jamon-2.79.jar** dans le *classpath* d'une application java

(exemple: dans *WEB-INF/lib*) .

```
import java.util.Iterator;
import com.jamonapi.Monitor;
import com.jamonapi.MonitorFactory;

public class TestAppJAMon {
    public static void main(String[] args) {
        Monitor mon=null,m=null;
        for (int i=1; i<=10; i++) {
            mon = MonitorFactory.start("myFirstMonitor");
            try { Thread.sleep(100 + i);
            } catch (Exception e) { e.printStackTrace();
            }
            mon.stop();
        }
        System.out.println(mon.toString());
        /*m=MonitorFactory.getMonitor("myFirstMonitor", "ms.");
        // m.reset();
        System.out.println(m);*/
        /*Iterator it = MonitorFactory.iterator();
        while(it.hasNext()){
            Monitor mm = (Monitor) it.next();
            System.out.println(mm);
        }*/
        System.out.println(MonitorFactory.getReport()); //tableau html
    }
}
```

Mesures affichées :

JAMon **Label=myFirstMonitor, Units=ms.**: (LastValue=112.0, Hits=10.0, Avg=105.9, Total=1059.0, Min=101.0, Max=112.0, Active=0.0, Avg Active=1.0, Max Active=1.0, First Access=Wed Nov 07 07:32:04 CET 2012, Last Access=Wed Nov 07 07:32:05 CET 2012)

```
<table border='1' rules='all'>
<th>Label</th><th>Hits</th><th>Avg</th><th>Total</th><th>StdDev</th><th>LastValue</th>
<th>Min</th><th>Max</th><th>Active</th><th>AvgActive</th><th>MaxActive</th>
<th>FirstAccess</th><th>LastAccess</th><th>Enabled</th><th>Primary</th>
<th>HasListeners</th><th>Label</th>
<tr><td>myFirstMonitor,
ms.</td><td>10.0</td><td>105.9</td><td>1059.0</td><td>3.21282153600563</td><td>112.0</td>
<td>101.0</td><td>112.0</td><td>0.0</td><td>1.0</td><td>1.0</td><td>Wed Nov 07
07:32:04 CET 2012</td><td>Wed Nov 07 07:32:05 CET
2012</td><td>true</td><td>false</td><td>false</td><td>myFirstMonitor, ms.</td></tr>
</table>
```

Label	Hits	Avg	Total	StdDev	LastValue	Min	Max
myFirstMonitor, ms.	10.0	105.9	1059.0	3.21282153600563	112.0	101.0	112.0

...

Remarque: la mesure "Active" (et les variantes associées "MinActive", "MaxActive", "AvgActive") correspond à la notion de "nombre d'appels simultanés".

"Active" vaut souvent "0" ou "1" en mode mono-thread et peut valoir "2" ou plus en mode multi-thread (ex: si fonctionnement au sein de tomcat avec beaucoup d'utilisateurs simultanés).

Dépendance maven pour jamonapi :

```
<dependency>
  <groupId>com.jamonapi</groupId>
  <artifactId>jamon</artifactId>
  <version>2.79</version>
</dependency>
```

16.3. Activation et désactivation des mesures

La méthode `MonitorFactory.setEnabled(true or false)` permet d'activer ou désactiver globalement toutes les mesures de jamon.

En mode "désactivé / enabled=false" les appels à `monitor.start()` et `monitor.stop()` ne font quasiment rien et l'on récupère ainsi un maximum de puissance CPU pour les fonctionnalités métiers de l'application. Ceci peut être quelquefois pratique en mode "production" .

Etant donné que l'appel à `MonitorFactory.setEnabled()` peut être effectué lors de l'exécution de l'application (au runtime) , l'activation/désactivation des mesures peut ainsi être pilotée par un administrateur (par exemple via la page `jamonadmin.jsp` de `jamon.war`) .

Pour paramétrer l'activation de seulement certaines mesures (avec plus de finesse) , il faudra au cas par cas régler certains paramètres techniques (url pour applications du filtre web , pointcut spring aop , ...).

16.4. Intégration au sein de Spring 2 ou 3 (pour mesurer les temps d'exécution de la partie "back-office" (services + dao))

```
package org.mycontrib.generic.profiler;
import org.aspectj.lang.ProceedingJoinPoint;

public interface GenericProfilerAspect {
    public abstract Object doProfiling(ProceedingJoinPoint pjp)
        throws Throwable;
}
```

```
package org.mycontrib.generic.profiler;
import org.aspectj.lang.ProceedingJoinPoint;
import com.jamonapi.Monitor;
import com.jamonapi.MonitorFactory;

/**
 * JamonGenericProfilerAspect est une classe d'aspect pour Spring-AOP
 * Elle enregistre les temps d'exécution des méthodes via l'api open source "JaMon" .
 * (pour récupérer les statistiques --> on peut partir de MonitorFactory.getMonitor("signature methode", "ms.");
 * ou de MonitorFactory.iterator() ou encore de MonitorFactory.getReport());
 *
 * @author Didier DEFRANCE
 *
 * exemple de configuration Spring :
 *
 * <bean id="jamonGenericProfilerAspectBean"
 *     class="org.mycontrib.generic.profiler.JamonGenericProfilerAspect"></bean>
 * <aop:config>
 *     <aop:pointcut id="execution_methodes_generic_dao"
 *         expression="execution(* org.mycontrib.generic.persistance.*.*(..))" />
 *     <aop:pointcut id="execution_methodes_dao"
 *         expression="execution(* tp.myapp.minibank.impl.persistance.dao.jpa.*.*(..))" />
 *
 *     <aop:aspect id="myProfilerAspect" ref="jamonGenericProfilerAspectBean">
 *         <aop:around method="doProfiling" pointcut-ref="execution_methodes_generic_dao" />
 *         <aop:around method="doProfiling" pointcut-ref="execution_methodes_dao" />
 *     </aop:aspect>
 * </aop:config>
 * */

public class JamonGenericProfilerAspect implements GenericProfilerAspect {
    public Object monitor(ProceedingJoinPoint pjp) throws Throwable {
        Monitor monitor = MonitorFactory.start(pjp.getSignature().toShortString());
        Object objRes = pjp.proceed();
        monitor.stop();
        return objRes;
    }
}
```

exemple de résultat (getReport() / html) :

Label	Hits	Avg	Total	StdDev
GenericDao.getEntityById(..), ms.	4.0	15.75	63.0	31.5
GenericDao.persistNewEntity(..), ms.	1.0	156.0	156.0	0.0

NB : L'exemple ci-dessus date de 2013.

depuis 2014 , les nouvelles versions **2.76** et **2.79** de jamon intègrent maintenant une classe "com.jamonapi.aop.spring.**JamonAspect**" qui est à peu près équivalente à la classe *JamonGenericProfilerAspect* de la page précédente (JamonAspect n'implémente pas l'interface GenericProfilerAspect).

Exemple de configuration spring :

```
<import resource="profilingSpringConf.xml" /> <!-- dans applicationContext.xml -->
```

profilingSpringConf.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans .....>

  <bean id="jamonAspectBean" class="com.jamonapi.aop.spring.JamonAspect" />

  <aop:config>
    <aop:pointcut id="execution_methodes_dao"
      expression="execution(* tp.myapp.minibank.impl.persistance.dao.jpa.*(..))" />
    <aop:pointcut id="execution_methodes_services"
      expression="execution(* tp.myapp.minibank.impl.domain.service.*(..))" />

    <aop:aspect id="myJamonAspect" ref="jamonAspectBean">
      <aop:around method="monitor" pointcut-ref="execution_methodes_dao" />
      <aop:around method="monitor" pointcut-ref="execution_methodes_services" />
    </aop:aspect>
  </aop:config>
</beans>
```

16.5. Intégration au sein d'un module EJB3 (alternative à Spring) :

META-INF/ejb-jar.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd">
  <assembly-descriptor>
    <!-- Default interceptor that will apply to all methods for
all beans in deployment -->
    <interceptor-binding>
      <ejb-name>*</ejb-name>
      <interceptor-class>
        com.jamonapi.aop.JAMonEJBInterceptor
      </interceptor-class>
    </interceptor-binding>
  </assembly-descriptor>
</ejb-jar>
```

16.6. Utilisation de l'api "JaMon" sur la partie JDBC/SQL

NB: le pseudo driver jdbc "JAMonDriver" (dont le code est à l'intérieur de jamon.jar) peut être utilisé pour **enregistrer automatiquement des mesures de performances sur les requêtes SQL** . Celles ci sont à récupérer via MonitorFactory.(getReport() ou ...)

Paramétrages :

Le nom complet du pseudo driver jdbc de JaMon est `"com.jamonapi.proxy.JAMonDriver"`

Si l'url jdbc ordinaire est de type `jdbc:mysql://localhost/minibank_db`

et si le driver jdbc ordinaire est `"com.mysql.jdbc.Driver"`

alors l'URL jdbc à fournir pour jamon est

`jdbc:jamon:mysql://localhost/minibank_db?jamonrealdriver=com.mysql.jdbc.Driver`

Exemple de configuration JDBC complète utilisant le pseudo-driver "JAMonDriver" :

```
<bean id="myDataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <!-- <property name="driverClassName" value="com.mysql.jdbc.Driver" />
  <property name="url" value="jdbc:mysql://localhost/minibank_db" /> -->

  <property name="driverClassName" value="com.jamonapi.proxy.JAMonDriver" />
  <property name="url" value=
"jdbc:jamon:mysql://localhost/minibank_db?jamonrealdriver=com.mysql.jdbc.Driver" />

  <property name="username" value="root" />
  <property name="password" value="root" />
</bean>
```

Exemple (partiel) de résultats :

Label	Hits	Avg	Total	StdDev	LastValue	Min	Max
MonProxy-SQL-PreparedStatement: insert into Client (ref_adressePrincipale, dateNaissance, email, nom, password, prenom, telephone) values (?, ?, ?, ?, ?, ?, ?), ms.	1.0	15.0	15.0	0.0	15.0	15.0	15.0
MonProxy-SQL-Statement: delete from CLIENT, ms.	1.0	16.0	16.0	0.0	16.0	16.0	16.0
MonProxy-SQL-Type: All, ms.	19.0	13.157894736842104	250.0	24.56796278330048	0.0	0.0	78.0
MonProxy-SQL-Type: delete, ms.	5.0	34.4	172.0	40.33360881448621	0.0	0.0	78.0
MonProxy-SQL-Type: insert, ms.	8.0	9.75	78.0	11.76860229593982	15.0	0.0	32.0
MonProxy-SQL-Type: select, ms.	6.0	0.0	0.0	0.0	0.0	0.0	0.0

16.7. Utilisation de l'api "JaMon" sur la partie Web (via filtre web)

De façon à récupérer automatiquement des mesures de performances sur la partie web (fabrication des pages html via servlet ou jsp) on peut activer un filtre web prédéfini de jamon qui interceptera automatiquement les requêtes http et qui générera des mesures sur les temps d'exécutions .

Ce filtre web (dont le code est dans jamon.jar) se configure dans **WEB-INF/web.xml** de la façon suivante :

```
<web-app>
...
  <filter>
    <filter-name>JAMonFilter</filter-name>
    <filter-class>com.jamonapi.JAMonFilter</filter-class>
  </filter>






  <filter-mapping>
    <filter-name>JAMonFilter</filter-name>
    <url-pattern>/*</url-pattern>  <!-- ou ... -->
  </filter-mapping>
</web-app>
```

Comme d'habitude avec jamon, les mesures sont à récupérer via **MonitorFactory.(getReport()** ou ...).

Etant donné que l'on situe alors forcément dans un environnement web java (ex : tomcat) , on pourra indirectement récupérer (et afficher) les mesures via **jamonadmin.jsp** (de **jamon.war**) ou bien via un équivalent (copie adaptée).

NB : Dans le cas ou l'application jamon.war est installée à coté de notreAppliWeb.war dans tomcat , il faudra prévoir une installation de jamon.jar dans les librairies partagées de tomcat (tomcat/lib ou ...).

Exemple de résultats (ici affichés via une copie de **jamonadmin.jsp**) :

JAMon Action	Mon Proxy Action	Output	Range/Units	Display Columns	Col Filter			
Refresh	No Action (currently=TTTF)	HTML	AllMonitors	Basic Cols Only	###			
Modify	Label ↑	Hits	Avg	Total	StdDev	LastValue	Min	Max
	/minibank-jpa/pages/identificationClient.jsf, ms.	2	66	132	72	117	15	117
	/minibank-jpa/pages/listeComptes_href.jsf, ms.	2	16	32	0	16	16	16
	/minibank-jpa/pages/operations_get.jsf, ms.	2	54	107	1	53	53	54
	/minibank-jpa/pages/virement.jsf, ms.	2	40	81	35	65	16	65
	/minibank-jpa/pages/welcome.jsf, ms.	5	3	15	7	15	0	15

Dans la plupart des cas, c'est un même thread (de tomcat ou ...) qui exécute :

- le code d'un servlet ou d'une page jsp (et les éventuels suppléments struts ou jsf)
- le code d'un service spring (ou ...) en arrière plan
- le code jdbc/sql pour accéder à la base de données

Le filtre web fournira donc des temps d'exécution globaux (front-office/ihm + back-office).

Ces mesures devront théoriquement être assez proches des mesures de "temps de réponse" que l'on peut récupérer via des outils externes de type "JMeter" .

Attention, attention :

Une application java/web démarre généralement très lentement car beaucoup d'aspects dynamiques nécessitent un grand nombre d'éléments à initialiser :

- pages JSP à transformer en servlet et à compiler
- configuration Spring/Hibernate à initialiser (introspection , aop , ...)
- ajustement des zones mémoires (allocations dans le "heap") .
-

Lorsque tout est bien initialisé , une application java fonctionne ensuite très rapidement.

Tout ceci induit donc une période initiale de chauffe où les mesures ne sont pas du tout significatives. Conseil : effectuer un "reset" des mesures après la période de chauffe pour ne pas comptabiliser les mesures non significatives dans les moyennes .

16.8. Temps théoriques à calculer :

tR : temps de réponse observé dans jMeter

tHttp : temps que les requêtes et réponses HTTP passent à traverser le réseau

tGlobalJee : temps d'exécution complet (jee+sql) récupéré par le filtre web de JaMon

tWeb : temps d'exécution purement "ihm web" (fabrication des pages html sans compter les sous-temps back-office)

tReqSQL : temps d'exécution (SQL+JDBC) récupéré par le pseudo driver jdbc de jaMon

tTraitService : temps d'exécution des services métiers (code java Spring ou EJB avec éventuels DAO et avec éventuelles conversions <<entity>>/<<dto>>) sans compter le temps d'exécution "SQL+DataBase"

tGlobalBackOffice : temps complet de la partie back-office (service + sql / db) (se mesure via intercepteur AOP Spring ou EJB3)

Temps récupérés par mesures directes de jaMon (ou JMeter):

tR , tGlobalJee , tReqSQL , tGlobalBackOffice

Temps récupérés par calculs indirects:

tTraitService = **tGlobalBackOffice** - **tReqSQL**

tWeb = **tGlobalJee** - **tGlobalBackOffice**

tHttp = **tR** - **tGlobalJee**

et **tempGlobalPurementJava** = **tGlobalJee** – **tReqSQL**