



# Tests de charge et fonctionnels avec *JMeter*

---

David THIBAU - 2020  
david.thibau@gmail.com



# Agenda

---

## Introduction

- Les tests de charges
- Installation JMeter
- Présentation de l'interface

## Concepts de base

- Éléments d'un plan de test
- Enregistrement des scénarios
- Variables et fonctions
- Éléments générant des variables

## Tests de performances

- Compteurs de temps
- Outils pour la validation
- Lancement des tests en mode batch
- Génération de rapport

## Tests fonctionnels

- Introduction
- Types d'assertions
- Récepteurs assertions

## Pour aller + loin

- Scripting
- Intégration continue
- Tests distribués



# Introduction

---

**Tests de performance/charge**

Installation JMeter

Interface JMeter



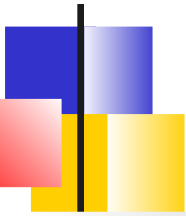
# Introduction

---

*Un test de performance est un test dont l'objectif est de déterminer la performance d'un système informatique. [Wikipedia]*

Plus concrètement, mesurer les temps de réponse ou le débit d'un système en fonction de sa sollicitation.

=> Définition très proche de celle de test de charge




# Variantes des tests de charge

---

Test de dégradations des transactions : Simuler l'activité transactionnelle d'**un seul** scénario fonctionnel

Test de stress : Simuler l'activité maximale attendue tous scénarios fonctionnels confondus en **heures de pointe** de l'application

Test de robustesse, d'endurance, de fiabilité : Simuler une charge importante d'utilisateurs sur une **durée relativement longue**



# Variantes des tests de charge (2)

---

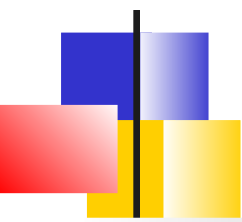
Test de capacité, test de montée en charge :

Simuler un nombre d'utilisateurs **sans cesse croissant** de manière à déterminer quelle charge limite le système est capable de supporter

Test aux limites : Simuler en général une activité **bien supérieure** à l'activité normale

Test de Benchmark : Comparaisons de logiciel, matériels, architectures,...

Tests de non-régression des performances, Tests de Composants, Tests de Volumétrie des donnée



# Spécification ou Plan de test

---

Le **plan de tests** est l'expression du besoin de la campagne de tests. C'est le premier livrable qui contient généralement :

- la présentation du projet
- les objectifs,
- le modèle de charge,
- le type de tests à réaliser,
- les scénarios fonctionnels à tester accompagnés des jeux de données nécessaires
- un planning



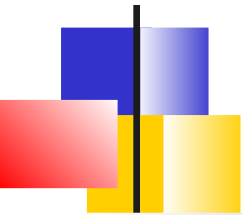
# Modèle de charge

---

La modélisation de la charge consiste à estimer la **charge représentative de l'activité réelle ou attendue** de l'application

- Cette modélisation s'estime à partir d'un modèle d'usage : nombre d'utilisateurs simultanés, nombre de processus métier réalisés, périodes d'utilisation, heures de pointe...
- Elle contient un nombre d'utilisateurs à simuler, leur répartition sur les différents scripts (scénarios fonctionnels), leurs rythmes d'exécution respectifs.
- Accessoirement, le modèle peut tenir compte des profils de montée ou descente de charge des groupes d'utilisateurs





# Bilan des tests

---

Le bilan des tests est un autre livrable qui comporte généralement trois vues:

- **Vue Métier** : Nombre de clients de l'application connectés, nombre de processus métier réalisés.
- **Vue Utilisateur** : Temps de réponse des transactions, taux d'erreur.
- **Vue Technique** : Consommations des ressources systèmes (CPU, Mémoire, I/O), consommations applicatives (métriques applicatives telles que sessions, attentes dans les pools threads/connexions, etc...).



# Méthodologie

---

- Tester de façon large, puis de façon approfondie
- Tester dans un environnement contrôlé

L'environnement de test doit être dans la mesure du possible identique à l'environnement de production, à défaut à une échelle étalonnée reconnue fiable permettant d'établir des abaques

=> 3 étapes :

- Analyse de Référence
- Tests Préliminaires
- Test de Charge à Grande Échelle



# Analyse de référence

---

- Définir le système à tester, les processus métier, et les objectifs (métiers, techniques, économiques)
- Définir les scénarios, par rapport à une analyse complète des risques métiers et techniques
- Définir le modèle de charge, par rapport au modèle d'usage de l'application
- Caractériser les données pour chaque scénario
- Enregistrer les scénarios



# Tests préliminaires

---

- Mettre en œuvre des moyens et définir la plate-forme de test
- Valider et Exécuter les tests de charge (préliminaires)
- Analyser les résultats
- Optimiser le système



# Tests de charge à grande échelle

---

- Effectuer plusieurs tirs
- Prouver la cohérence des résultats (écarts types, percentiles)
- Obtenir des métriques simples : débit, indice de satisfaction utilisateur,
- Standardiser ses rapports



# Plate-forme de test

---

La plate-forme de test comprend :

Un **injecteur de charge** : logiciel capable de simuler les actions des utilisateurs et de générer la charge. Ces actions sont définies dans des scripts automatisant des scénarios et valorisés sur un lot de données particulier.

=> 2 fonctionnalités :

- **Simulation d'utilisateurs**
- **Générations de données** qui assure que chaque utilisateur virtuel n'effectue pas exactement la même opération.

Des **sondes** : placées au niveau du système cible, elles permettent de remonter des métriques. (utilisation de la mémoire, processeur, disque et réseau).



# Données de test

---

Deux principaux types d'outils :

- ceux qui s'appuient sur les **données de production** et proposent des outils d'extraction et de transformation des données de production
- ceux qui s'appuient sur des **mécanismes de génération** pour produire à partir de rien (ou presque) les jeux de données de test.



# Panorama des outils

---

HP LoadRunner/Performance Center, ex produit Mercury Interactive, le leader du marché.

Oracle Load Testing, ex produit Empirix, racheté par Oracle Corporation en 2008

IBM Rational Performance Tester.

Apache JMeter (produit Open Source). Support *BlazeMeter*

Gatling : Open Source avec version payante





# Introduction

---

Tests de performance/charge

**Installation JMeter**

Interface JMeter



# Historique

---

Développé par Stefano Mazzocchi de l'Apache Software Foundation

Initialement, pour tester les performance de *Apache Jserv* qui est devenu *Tomcat*

Puis étendu aux tests de charge de serveurs FTP, base de données, Servlets et objets Java.

Désormais reconnu comme un bon outil de test de performance pour les applications web

Également utilisé pour les tests fonctionnels



# Avantages

---

JMeter par rapport à la concurrence présente les avantages suivants

1. Gratuit
2. Simple, une courbe d'apprentissage rapide (HTML et expressions régulières)
3. Extensible : Une API est disponible pour la personnalisation. Le code source est libre
4. Support : Documentation en ligne, forums, ...  
*BlazeMeter*



# Fonctionnalités

---

JMeter peut être utilisé pour tester la performance de ressources statiques ou dynamiques

- Principalement utilisé pour le web, il peut également stressé via FTP, JDBC, Mongo, ...
- Enregistrement des scénarios de test via le navigateur
- Simulateur pour différents types de serveurs, différents types de client
- Peut exécuter des tests distribués
- Peut effectuer des tests fonctionnels avec les assertions
- Automatisation aisé via scripts, Ant, Maven, Jenkins, ...



# Installation : Pré-requis

---

JMeter est une application 100% Java

Les dernières versions nécessitent une JVM 1.8

La simulation de nombreuses threads peuvent provoquer de forte charge sur la machine cliente

=> Bonne machine ou architecture maître/esclave



# Installation

---

L'installation consiste à télécharger la dernière release stable de JMeter et de la décompresser.

- Il est recommandé de positionner la variable d'environnement ***JAVA\_HOME***

La distribution contient tous les fichiers nécessaires pour construire et exécuter la plupart des tests, cependant

- Dans le cas de stress JDBC, vous devez récupérer le driver JDBC.
- Dans le cas de stress JMS, vous devez récupérer une implémentation d'un client JMS



# Structure répertoire

---

Une fois dézippé, on obtient les sous-répertoires suivants

- ***bin*** : Binaires, démarrage, arrêt, etc.
- ***docs*** : JavaDoc des classes
- ***extras*** : Pour étendre *JMeter* avec du code spécifique
- ***lib*** : Librairies tierces ou d'intégration JMeter
- ***printable\_docs*** : Documentation (copie de ce que l'on trouve sur le site)



# Exécution

---

Le démarrage se fait par :

- *jmeter.bat* (Windows)
- ou *jmeter* (Unix)

Le script a de nombreuses options.

Principalement, il permet de :

- Démarrer l'interface utilisateur
- Démarre un tir en mode batch
- Générer un rapport à partir d'un fichier résultat





# Classpath JMeter

---

JMeter recherche les classes dans les répertoires suivants :

- ***JMETER\_HOME/lib*** : Utilitaires, drivers JDBC, JMS
- ***JMETER\_HOME/lib/ext*** : Composants JMeter et add-ons

Cette configuration peut être modifiée via la propriété *search\_paths* dans *jmeter.properties*

L'utilisation de la variable d'environnement CLASSPATH n'a pas d'effet car JMeter est démarré avec la commande "*java -jar*" qui ignore cette variable.



# Utilisation d'un proxy

---

· Lors de l'exécution de JMeter derrière un proxy, il est possible de fournir les paramètres du proxy en ligne de commande :

- **-H** [IP ou adresse du serveur proxy ]
- **-P** [Port du proxy]
- **-N** [Les hôtes ne nécessitant pas de proxy] (ex : \*.apache.org|localhost)
- **-u** [username pour l'authentification]
- **-a** [password pour l'authentification]

Exemple :

```
jmeter -H my.proxy.server -P 8000 -u username -a password -N localhost
```



# Principales options de la ligne de commande

---

- n** : Exécution en mode non-gui
- t** : fichier contenant le plan de test.
- l** : Fichier servant à enregistrer les résultats.
- j** : Fichier de log de JMeter.
- r** : Exécuter le test sur les serveurs spécifiés par la propriété JMeter "*remote\_hosts*"
- R** : liste de serveurs distants, Exécution du test sur les serveurs spécifiés

Exemple : *jmeter -n -t my\_test.jmx -l log.jtl -H my.proxy.server -P 8000*



# Options pour génération d'un rapport de test

---

Depuis la version 3, JMeter propose de générer un rapport de test complet et formaté contenant les principaux indicateurs d'un test de charge.

Les options sont alors :

- g *[path to CSV file]*** : Génération du rapport uniquement (sans exécution du test de charge)
- e** : Génération du rapport après l'exécution d'un test de charge
- o** : Répertoire où est généré le rapport. (Le répertoire n'existe pas ou est vide)



# Propriétés de configuration

---

3 types de propriétés de configuration peuvent avoir une influence lors de l'exécution d'un plan de test :

- **Propriété Système** (JVM) : Propriété de configuration cœur de JMeter  
Ex : *language* : Langage de l'interface
- **Propriété JMeter** : Propriété de configuration utilisée dans les scripts. Ex : Nbre d'utilisateurs simulés, Adresse du serveur à tester, ...
- **Propriété de trace**

Ces propriétés sont définies dans ***jmeter.properties***, ***user.properties*** ou surchargées via la ligne de commande



# Propriétés et ligne de commande

---

Les propriétés peuvent être spécifiées sur la ligne de commande :

- D[*prop\_name*]=[*value*]** : Propriété JVM
- S[*propertyFile*]** : Fichier contenant les propriétés système
- J[*prop name*]=[*value*]** : Propriété JMeter
- q[*propertyFile*]** : Fichier contenant les propriétés JMeter. Par défaut *user.properties*
- G[*prop name*]=[*value*]** : Propriété JMeter à envoyer aux serveurs distants
- G[*propertyfile*]** : Fichier de propriétés JMeter à envoyer à tous les serveurs distants
- L[*category*]=[*priority*]** : Niveau de trace pour une catégorie

Exemples :

```
jmeter -Duser.dir=/home/mstover/jmeter_stuff \  
-Jremote_hosts=127.0.0.1 -Lorg.apache=DEBUG
```

```
jmeter -LDEBUG
```



# Configuration JMeter

---

Quelques propriétés de *jmeter.properties* :

- *ssl.provider* : Implémentation SSL
- *xml.parser* : Implémentation du parseur XML (par défaut : `org.apache.xerces.parsers.SAXParser` )
- *remote\_hosts* : La liste des serveurs *JMeter* distants contrôlés par l'interface graphique
- *not\_in\_menu* : Liste des composants n'apparaissant pas dans le menu
- *search\_paths* : Liste de chemins pour rechercher les add-on en plus de `lib/ext`
- *user.classpath* : Liste de chemins pour les classes utilitaires en plus de `lib`
- *user.properties* : Fichier contenant des propriétés JMeter supplémentaires. (avant les options `-q` et `-J`)
- *system.properties* : Fichier contenant des propriétés système supplémentaires. (avant les options `-S` et `-D`)



# Introduction

---

Tests de performance/charge

Installation *JMeter*

**Interface utilisateur *JMeter***





# Introduction

---

L'interface utilisateur de JMeter permet :

- D'élaborer des plans de test
- De les sauvegarder
- D'ouvrir des plans de test précédemment sauvegardés
- D'exécuter des plans de test
- De sauvegarder/inclure des fragments de plans de tests

Elle sert principalement à l'élaboration des plans de test. Le mode batch est utilisé pour les tirs



# Plan de test

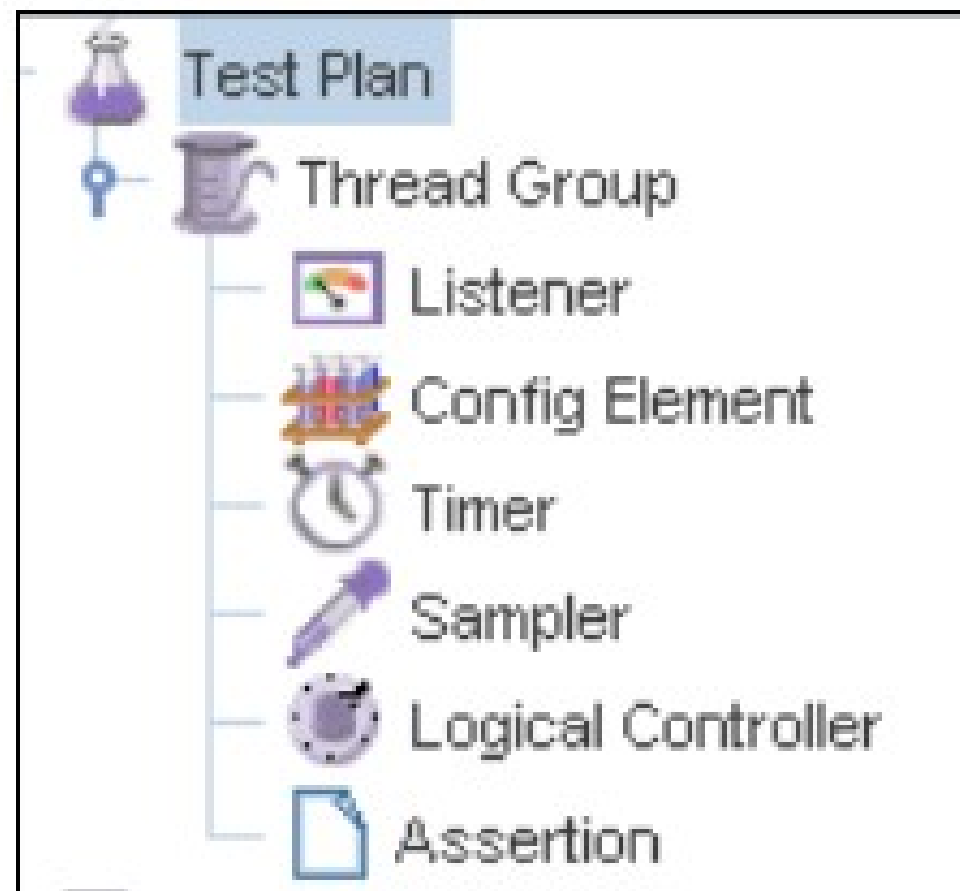
---

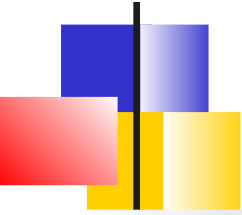
Un plan de test JMeter est un arbre hiérarchique (fichier XML) dans lequel des éléments de test ont été placés.

Chaque élément peut avoir un élément parent et  $n$  éléments enfants.

L'élaboration d'un plan de test consiste à construire l'arbre des éléments.

# Éléments d'un plan de test





# Ajout d'éléments

---

L'ajout et la suppression d'éléments se fait via le clic-droit sur un nœud

Il est également possible :

- de charger des éléments (fragment XML) à partir d'un fichier.

*Sélection du nœud → Ouvrir dans le menu contextuel*

- De sauvegarder un fragment de l'arbre des éléments
- L'interface supporte également le Drag&Drop

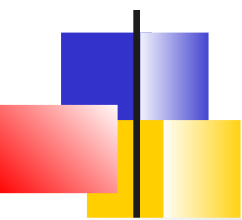


# Opérations sur les nœuds

---

L'UI propose différentes fonctionnalités sur les nœuds :

- Cut/Copy/Paste/Duplicate/Remove
- Copie sous un format image
- Activer/Désactiver
- Aide

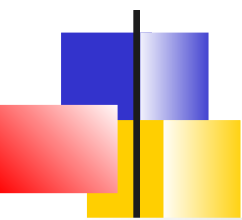


# Sauvegarde du plan de test

---

Il est recommandé de sauvegarder le plan de test avant de l'exécuter

Le plan de test est sauvegardé dans un fichier XML (*.jmx*)



# Exécution d'un plan de test

"Start" (Control + r) à partir du menu  
"Run"

10 / 10 

Le nombre à gauche de la boîte verte  
indique le nombre de threads actives /  
le total de threads

- Cela n'inclut pas les threads démarrées  
de façon distante



# Arrêt

2 types de commandes sont disponibles à partir du menu :

- **Stop (Control + '.')** : Stoppe les threads immédiatement si possible.  
Si cela ne marche pas, réessayer ou au pire redémarrer JMeter.
- **Shutdown (Control + ',')** : Demande aux threads de s'arrêter à la fin ce qu'elles sont en train de faire.  
La dialogue reste active jusqu'à ce que toutes les threads soient arrêtées.  
Si cela prend trop de temps, il est possible de fermer la dialogue et essayer un Stop.





# Arrêt en mode non GUI

Lorsque JMeter s'exécute en mode non-GUI, il écoute sur un port TCP spécifique

- par défaut 4445 : *jmeterengine.nongui.port*
- choix automatique si le port est occupé jusqu'à *jmeterengine.nongui.maxport*
- Le port choisi est affiché dans la fenêtre console

Les commandes supportées sont alors :

- ***StopTestNow*** : Arrêt immédiat
- ***Shutdown*** : Arrêt en douceur

Ces commandes peuvent être envoyées en utilisant les scripts *shutdown[.cmd|.sh]* ou *stoptest[.cmd|.sh]*



# Nettoyer

---

Les éléments récepteurs reçoivent les résultats des échantillons

Entre chaque test, il est utile de les réinitialiser :

*Run → Clear*

*Run → Clear All*



# Trace des erreurs

---

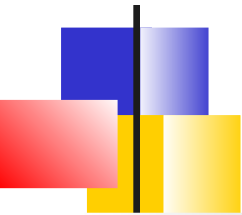
JMeter reporte les avertissements et les erreurs dans le fichier ***jmeter.log***.

*Si il ne peut pas reporter les erreurs, celles-ci s'affichent sur la console.*

Le fichier de log est visible dans l'interface utilisateur

*Options → Log viewer*

*Les erreurs des échantillons (comme un 404) n'apparaissent pas dans le fichier de log mais sont stockés dans les fichiers résultats du test.*



# Recherche d'éléments

---

JMeter propose le menu « **Rechercher** » qui propose les options suivantes :

- Sensible à la casse
- Regexp : Utilisation des expressions régulières

La recherche s'effectue alors sur l'ensemble des éléments de test du plan de test



# Concepts de base

---

## **Éléments d'un plan de test**

Enregistrement de scénarios

Variables et fonctions

Éléments de test créant des variables

Plan de test

Nom : Plan de test

Commentaires :

Variables pré-définies

Nom :	Valeur :

Détail

Ajouter

Ajouter depuis Presse-papier

Supprimer

Monter

Descendre

- ☐ Lancer les groupes d'unités en série (c'est-à-dire : lance un groupe à la fois)
- ☐ Exécuter le Groupe d'unités de fin même après un arrêt manuel des Groupes d'unités principaux
- ☐ Mode de test fonctionnel

Sélectionner le mode de test fonctionnel uniquement si vous avez besoin d'enregistrer les données reçues du serveur dans un fichier à chaque requête.

Sélectionner cette option affecte considérablement les performances.

Ajouter un répertoire ou un fichier 'jar' au 'classpath'

Parcourir...

Supprimer

Nettoyer

Librairie



# Plan de test

---

Le plan de test est l'élément racine. Il définit des configurations appliquées globalement au plan de test.

- Des variables-prédéfinies : Utile pour des valeurs répétées dans de nombreux éléments  
ex : `${SERVER}`
- Le mode test fonctionnel, à ne pas utiliser en test de charge, enregistre les réponses dans des fichiers résultats (affecte énormément les performances)
- Options sur l'exécution des groupes d'unité.



# Groupe d'unités

Les groupes d'unités (ou Thread group) représentent un ensemble d'utilisateurs simulés

Un plan de test contient au moins un groupe d'unités. Il peut en contenir plusieurs

La plupart des autres éléments de test doivent se trouver à l'intérieur d'un groupe d'unités.

Les attributs d'un groupe d'unités sont :

- Le **nombre de threads** : Le nombre total d'utilisateurs simulés ou nombre de threads que va démarrer JMeter
- La **période de lancement** : La période consacrée au démarrage des threads en secondes => *JMeter* lancera un thread tous les nombre de threads/ramp-up.  
La période de lancement doit être assez longue afin d'éviter des charges trop importante au démarrage du test mais assez courte pour que la dernière thread démarre avant que la première ait terminé  
Une méthode recommandée est de démarrer avec Ramp-up = nombre de threads puis ajuster.
- Le **nombre d'itérations** : Nombre d'exécution du scénario de test pour chaque thread





# Groupe d'unités

## Groupe d'unités

Nom : JMeter Users

Commentaires :

Action à suivre après une erreur d'échantillon

☒ Continuer ☐ Démarrer itération suivante du Thread ☐ Arrêter l'unité ☐ Arrêter le test ☐ Arrêter le test immédiatement

## Propriétés du groupe d'unités

Nombre d'unités (utilisateurs) : 1

Durée de montée en charge (en secondes) : 5

Nombre d'itérations : ☐ Infini 2

☒ Même utilisateur à chaque itération

☐ Créer les unités seulement quand nécessaire

☐ Programmeur de démarrage

Durée (secondes) :

Délai avant démarrage (secondes) :



# Création des threads

---

L'option « *Créer les unités seulement lorsque c'est nécessaire* » :

- Les threads sont créées seulement si la proportion adéquate de la période de lancement s'est écoulée.
- Ceci est pratique pour les tests qui ont une période de lancement plus longue que l'exécution d'une simple thread.
- Si la case n'est pas cochée, toutes les threads sont créées lorsque le test démarre puis démarrées en fonction de la période de lancement.  
C'est la valeur par défaut.



# Scheduler

---

La case à cocher « *Programmateur de démarrage* » permet de faire apparaître des champs supplémentaires.

- L'heure de démarrage et d'arrêt : Lorsque le test est lancé, JMeter attend l'heure de démarrage pour lancer les threads.
- A la fin de chaque cycle, JMeter vérifie l'heure d'arrêt et stoppe éventuellement le test.



# Groupe d'unités de début ou de fin

---

Il y a 2 types de groupe d'unités particulier :

- Les groupes d'unités de ***setUp*** permettent d'exécuter des threads avant le test
- Les groupes d'unités de ***tearDown*** permettant d'exécuter des threads à la fin du test



# Typologie des éléments

Plusieurs types d'éléments peuvent être ajoutés sous un groupe d'unité :

- **Éléments de configuration** : Paramètres de configuration mutualisés sur plusieurs échantillons
- **Échantillon** : L'unité à tester. (requêtes, appel systèmes, ...)
- **Contrôleurs** : Modifie le séquençement des échantillons
- **Récepteurs** : Sauvegarde ou affiche des résultats
- **Pré-processeur / post-processeur** : Traitement sur les échantillons avant ou après leur exécution. Permet de la dynamicité
- **Assertions** : Vérification de la réponse
- **Compteurs** : Temporisation de l'exécution du test



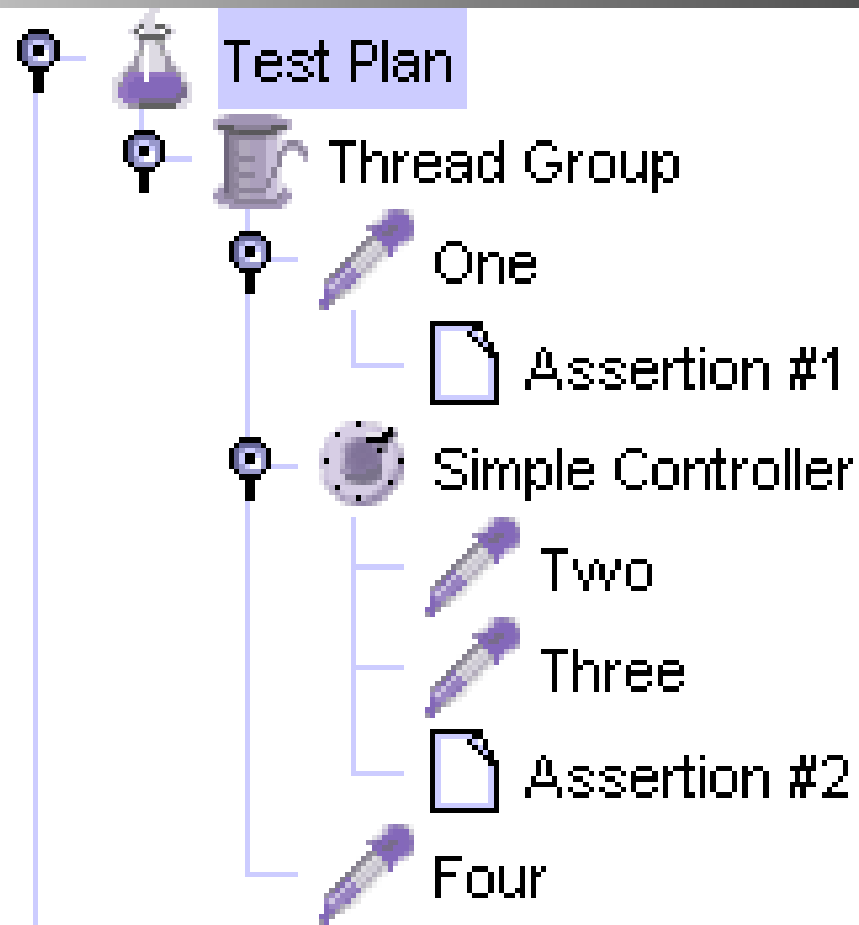
# Contexte et séquence

---

- Les éléments contrôleurs et échantillons sont des éléments ordonnés s'exécutant en séquence, ils définissent également un *contexte*
- Les autres éléments : récepteurs, configuration, post-processeurs, pré-processeurs, assertions, compteurs de temps ne sont pas ordonnés et affectent leur contexte parent

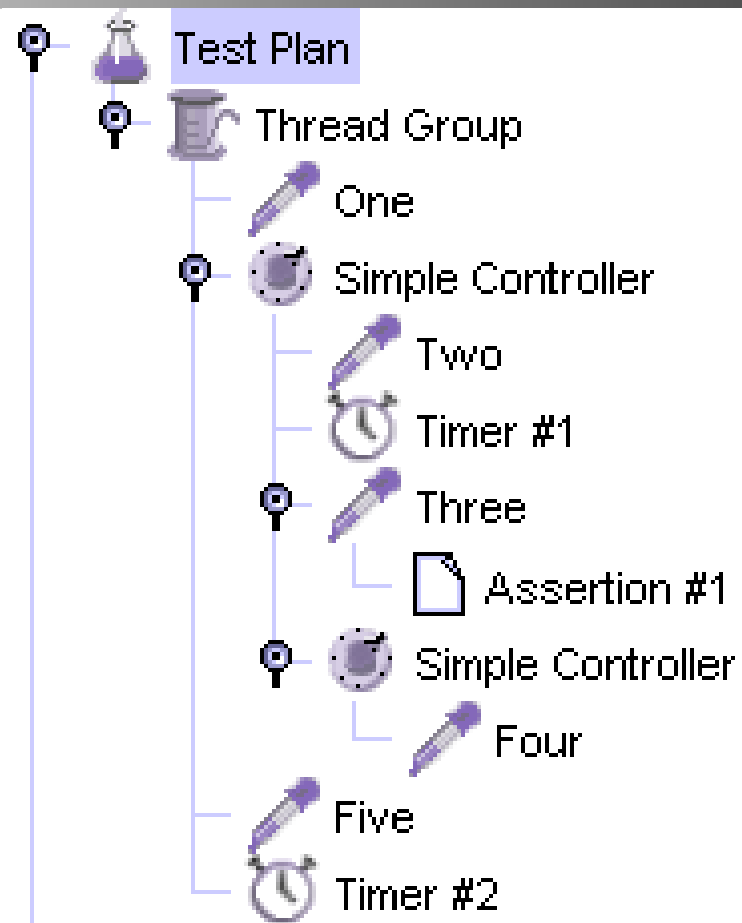
# Exemple

*Assertion #2 affecte l'échantillon Two, Three*

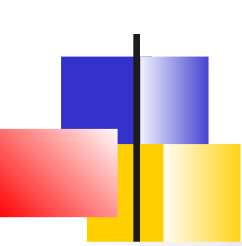


# Exemple

*Timer #2 affecte One, Simple Controller et Five*







# Éléments de configuration

---

Les éléments de configuration peuvent être utilisés pour

- Positionner des valeurs par défaut
- Positionner des variables
- Ajouter une fonctionnalité (Ex : gestion des cookies par exemple)
- Positionner des *entêtes-http*

Ils s'appliquent à tous les échantillons de leur contexte



# Éléments de configuration

---

**Requêtes par défaut** FTP / HTTP / Java / LDAP / TCP

**Gestionnaire d'authentification HTTP** : Permet d'indiquer des login/mot de passe sur certaines URLs et de s'authentifier via HTTP

**Gestionnaire de cache HTTP** (basé sur les entêtes Last-Modified et Etag)

**Gestionnaire de cookies** (nécessaire pour maintenir les sessions http)

**Gestionnaire d'entêtes HTTP**



# Éléments de configuration (2)

---

**Configuration de connexion JDBC**

**Configuration de keystore**

**Variables pré-définies/ aléatoires** : Permet de définir et initialiser des variables de script avec des valeurs prédéfinies ou aléatoires

**Compteur** : Une variable compteur utilisable dans un moteur d'utilisateur

**Source de données CSV** : Utilisé pour positionner des variables à partir d'un fichier CSV



# Échantillons

Les éléments **échantillons** spécifient les requêtes que JMeter effectuera lors de l'exécution du test.

La requête peut être personnalisée :

- via ses attributs propres
- et des éléments de configuration communs à tous les échantillons.



# Types d'échantillons

---

**Requête FTP** (GET/PUT)

**Requête HTTP** (GET/POST, ...), REST, SOAP

**Requêtes TCP, SMTP, AJP**

**Requête JDBC** : Exécution d'une requête SQL

**Requête Java, *JUnit*** : Solliciter une classe Java (implémentant une interface JMeter) par plusieurs threads

**Requête LDAP et LDAP étendue** : Stress d'un annuaire LDAP via des requêtes LDAP (Ajouter, Modifier, Supprimer et rechercher)



# Types d'échantillons (2)

---

**Exécution de scripts** (BeanShell ou JSR223),

**Exécution de commandes systèmes**

**JMS** : Publisher, Subscriber, Point-to-Point : Files d'attente et topic JMS

**Mail Reader, SMTP** : Lecture de boîte mails  
(Test d'une procédure d'enregistrement par ex.)

**Test Action** : Permet de stopper une thread, de la mettre en pause. Utilisé avec un contrôleur conditionnel



# Contrôleurs logiques

---

Les **contrôleurs logiques** permettent de personnaliser la logique utilisée par JMeter pour envoyer les requêtes.

Ils peuvent changer l'ordre des requêtes, les modifier, les répéter.

Un élément contrôleur agit sur ses éléments échantillons enfants



# Types de contrôleurs

---

**Simple** : Aucune fonctionnalité, si ce n'est de grouper logiquement plusieurs échantillons

**If, Switch** : Exécution conditionnelle

**Loop, While** : Boucles

**ForEach** : Itération avec modification des données d'entrées

**Once Only** : Les échantillons ne sont exécutés qu'une seule fois (quelque soit le nombre de boucles défini dans le groupe d'unités)

**Interleave** : JMeter choisit un échantillon différent à chaque itération de boucle (dans l'ordre)

**Random** : Idem que *intervleave* mais l'échantillon est choisi au hasard

**Random Order** : Exécute tous les échantillons mais dans un ordre aléatoire





# Types de contrôleurs (2)

---

**Throughput** : Permet de contrôler le nombre d'exécutions ou le pourcentage

**Runtime** : Garantit que le temps d'exécution des échantillons enfants ne dépasse pas un certain temps

**Module** : Permet de modifier des fragments de plan de test lors de l'exécution. Les fragments sont alors variabilisés

**Include** : Inclusion d'un autre fichier jmx

**Transaction** : Délimite une transaction dont le temps d'exécution sera enregistré

**Recording** : Détermine l'endroit où est enregistré le plan de test lorsque l'on utilise la fonctionnalité d'enregistrement de JMeter



# Récepteurs

---

Les récepteurs remplissent plusieurs usages

- Visualiser en temps-réel,
- Sauvegarder les résultats dans un fichier
- Lire les résultats précédemment sauvegardés.

Pour chaque récepteur, Il est possible de spécifier le fichier dans lequel sont écrits (ou lus) les résultats.

Par défaut, les résultats sont stockés dans des fichiers CSV avec l'extension *.jtl*. Mais il est également possible de les stocker en XML



# Version 3.x

---

Depuis la version 3.x, les récepteurs dédiés à la visualisation des résultats ont été dépréciés et remplacés par les fonctionnalités de génération de rapport standard

=> Les récepteurs sont surtout utilisés pour mettre au point les plans de test.

En particulier :

- L'**arbre de résultat** permet de vérifier précisément chaque échantillon
- Le **tableau de résultat** pour vérifier le séquençement des échantillons
- Le **rapport agrégé** qui donne des indicateurs de performance et de taux d'erreur



# Autres éléments

---

Les **assertions** sont utilisées pour exécuter des vérifications additionnelles sur les échantillons et sont traitées après chaque échantillon du même contexte

Les **compteurs** de temps permettent d'introduire des pauses dans les séquences de requêtes

Les **pré-processeurs** modifient les échantillons avant leur exécution.

Les **post-processeurs** sont appliqués après l'exécution de l'échantillon. Ils servent à extraire des données de la réponse et de les stocker dans des variables



# Ordre d'exécution

---

1. Éléments de configuration
2. Pré-processeurs
3. Compteurs de temps
4. Échantillonneur
5. Post-processeur (à moins que le résultat de l'échantillon soit vide)
6. Assertions (à moins que le résultat de l'échantillon soit vide)
7. Récepteurs (à moins que le résultat de l'échantillon soit vide)



# Pré-processeurs

---

Les pré-processeurs modifient les échantillons de leur contexte.

**Analyseur de lien HTML** : parse la réponse HTML précédente et y extrait les liens et formulaires selon des gabarits. (Spider)

**Transcripteur d'URL HTTP** : Peut être utilisé pour ajouter systématiquement un ID de session à un ensemble d'échantillons

**Paramètres utilisateur** : Permet de spécifier des valeurs de variables différentes pour chaque threads.

**BeanShell, JSR223** : Exécution d'un script avant le lancement de la requête

**JDBC** : Exécuter du SQL avant un échantillon

**Paramètre Utilisateurs RegEx** : Permet de spécifier des valeurs dynamiques pour les paramètres HTTP en utilisant des expressions régulières.



# Autres éléments

---

**Serveur Proxy HTTP** : Enregistrement de scénario

**Serveur Miroir HTTP** : Ce serveur renvoie simplement les données qui lui sont envoyées .

**L'afficheur de propriétés** présente les propriétés système ou JMeter et permet de les modifier

**Échantillon Debug** : Génère un échantillon contenant les valeurs de toutes les variables JMeter et des propriétés.  
Visibles dans l'arbre de résultats



# Concepts de base

---

Éléments d'un plan de test

**Enregistrement de scénarios**

Variables et fonctions

Éléments de test créant des variables





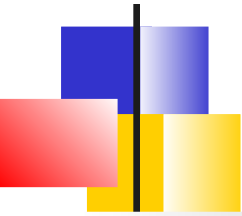
# Introduction

---

JMeter permet d'enregistrer la navigation de son navigateur directement dans un plan de test

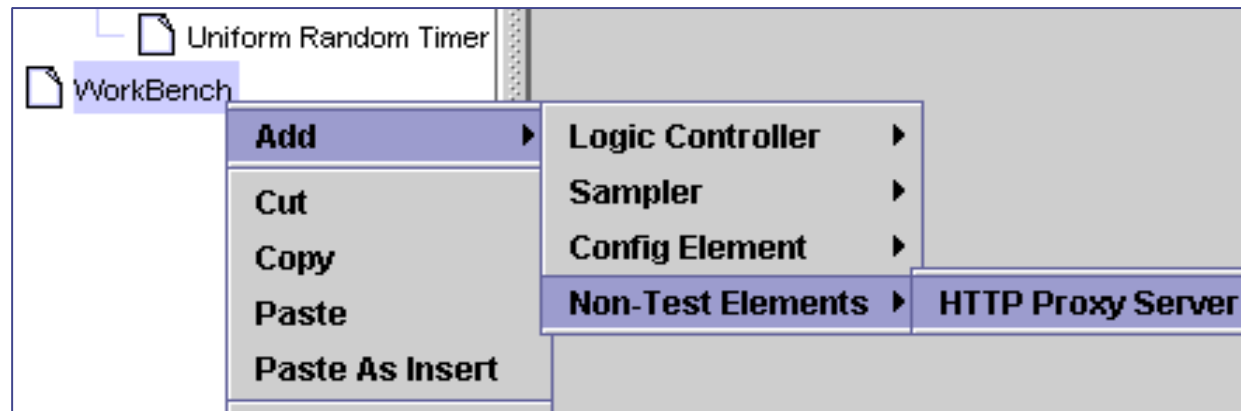
- L'élément « Enregistreur script de test » doit être ajouté au Plan de Test, puis démarré
- Le navigateur doit être configuré pour utiliser le proxy de JMeter

# Utilisation de JMeter



Définition du proxy de Jmeter et d'un port

Une fois configuré : « Start »

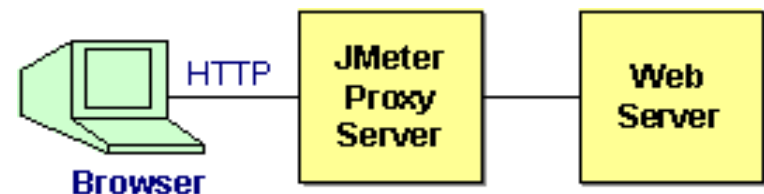
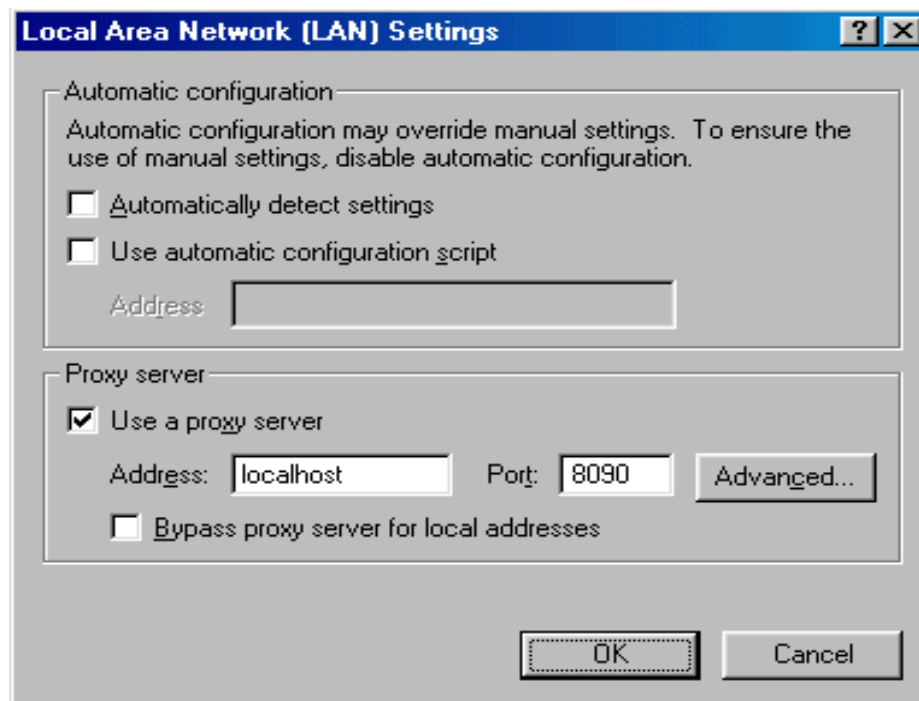


HTTP Proxy Server	
Name:	HTTP Proxy Server
Port:	8090

# Utilisation de JMeter

Enregistrement des use cases

Définition d'un proxy dans le navigateur





# Enregistreur

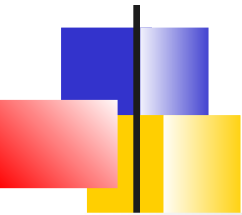
## *Paramètres du plan de test*

**Le contrôleur Cible** : Le contrôleur JMeter où les échantillons seront stockés. Par défaut, il recherche un contrôleur d'enregistrement (Recording Controller)

**Grouper** : Indique si il faut grouper les échantillons provenant d'un seul clic et comment représenter ce regroupement:

- « *Ne pas grouper* » : Tous les échantillons sont sauvegardés en séquence
- « *Ajouter des séparateurs* » : Ajoute un contrôleur nommé "-----" pour créer une séparation
- « *Mettre chaque groupe dans un nouveau contrôleur* » : Créé un contrôleur simple pour chaque groupe
- « *Stocker le 1er échantillon pour chaque groupe* » : Seul la première requête de chaque groupe sera stocké, les options "*Follow Redirects*" et "*Retrieve All Embedded Resources...*" seront actives.
- « *Mettre chaque groupe dans un contrôleur de transaction* » : Crée une nouveau contrôleur transaction pour chaque groupe/clic

La propriété **proxy.pause** détermine le delta minimum entre 2 requêtes pour que JMeter les traite comme 2 groupes distincts (Par défaut : 1s)



# Enregistreur

## *Paramètres du plan de test (2)*

**Capter les entêtes HTTP** : Les entêtes doivent elles être ajoutées dans le plan de test ?

- Si oui, un gestionnaire d'entête est ajoutée à chaque échantillon.
- Le serveur supprime toujours les entêtes *Cookie* et *Authorization*.
- Par défaut, il supprime également les entêtes *If-Modified-Since* et *If-None-Match*.
- Il est possible de définir d'autres entêtes à supprimer via la propriété *proxy.headers.remove*

**Ajouter des assertions** : Ajouter une assertion vierge à chaque échantillon

**Correspondance des variables par regexp** : Les remplacements de variables se font sur le mot entier (et non pas sur une partie). Voir +loin



# Enregistreur

## *Paramètres échantillon HTTP*

---

Paramètres de nommage de l'échantillon :

- **Préfixe** : Ajoutez un préfixe
- **Nom de la transaction** : Remplacez le nom par celui de l'utilisateur

**Temps d'inactivité** pour délimiter un groupe  
(possibilité de surcharger *proxy.pause*)

**Type** : Quel type d'implémentation du protocole HTTP  
(Fournie par la JVM ou librairie Apache)

**Positionnement d'options** dans les échantillons  
générés : *Redirect Automatically, Follow Redirects, Keep-Alive, Retrieve all Embedded Resources*



# Gestion des paramètres HTTP par défaut

---

Si le serveur proxy trouve un élément  
« *Paramètres HTTP par défaut* » dans  
le contrôleur où il enregistre ou un de  
ses parents.

=> les échantillons enregistrés auront  
des champs vides pour les valeurs par  
défaut précisées.



# Remplacement des variables prédéfinies

---

Si le serveur Proxy trouve un élément « *Variables prédéfinies* » sous le contrôleur d'enregistrement,

=> les données des échantillons enregistrés qui contiendront la valeur d'une de ces variables seront remplacées par l'expression de la variable.

*Par exemple, si serveur cible est "xxx.example.com" et qu'une variable prédéfinie nommée serveur a la valeur "xxx.example.com".*

*=> Tous les champs des échantillons enregistrés qui contiennent "xxx.example.com" seront remplacés par "{\$serveur}".*





# Filtres des échantillons

---

Il est possible d'appliquer différents filtres pour limiter les échantillons sauvegardés :

- Filtre sur le type de contenu : par exemple : "*text/html [;charset=utf-8 ]*". Le filtre d'inclusion est appliqué en premier
- Motifs d'URL : Des expressions régulières permettent d'inclure et exclure des URLs



# Cas du HTTPS

---

Quand le serveur cible est en HTTPS, JMeter est capable de générer un certificat qui usurpe l'identité du serveur et qui est self-signed.

*JMeter utilise keytool et le certificat généré est **ApacheJMeterTemporaryRootCA.crt***

Ensuite, le certificat doit être installé dans le navigateur afin qu'il fasse confiance à l'autorité JMeter.



# Concepts de base

---

Éléments d'un plan de test  
Enregistrement de scénarios

## **Variables et fonctions**

Éléments de test créant des variables



# Introduction

---

JMeter définit 2 types d'éléments dynamiques :

Les variables, elles sont référencées comme suit :

***`${VARIABLE}`***

Les fonctions prédéfinies qui effectuent un traitement. L'appel d'une fonction se fait comme suit :

***`${_functionName(var1,var2,var3)}`***

Les fonctions et variables peuvent être affectées à n'importe quel champ des éléments de test (à l'exception du plan de test).

Certains champs attendant des entiers n'acceptent pas les fonctions



# Règles de syntaxe

---

Les variables, fonctions (et propriétés) sont sensibles à la casse.

Les espaces de début ou fin dans les noms de variables sont supprimées avant utilisation.

A la différence des propriétés, les **variables sont locales à une thread**.

- Pour accéder à une propriété, on peut utiliser les fonctions `__P` ou `__property`

Si une variable ou fonction référencée n'est pas définie. JMeter ne trace pas une erreur mais ne remplace pas la valeur. (La valeur de `${UNDEF}` est `${UNDEF}`)

Les variables ne peuvent pas être imbriquées `${Var${N}}` n'est pas correct. Il faut alors utiliser la fonction `__V ${__V(Var${N})}`

Le caractère d'échappement est \



# Appel de fonction

---

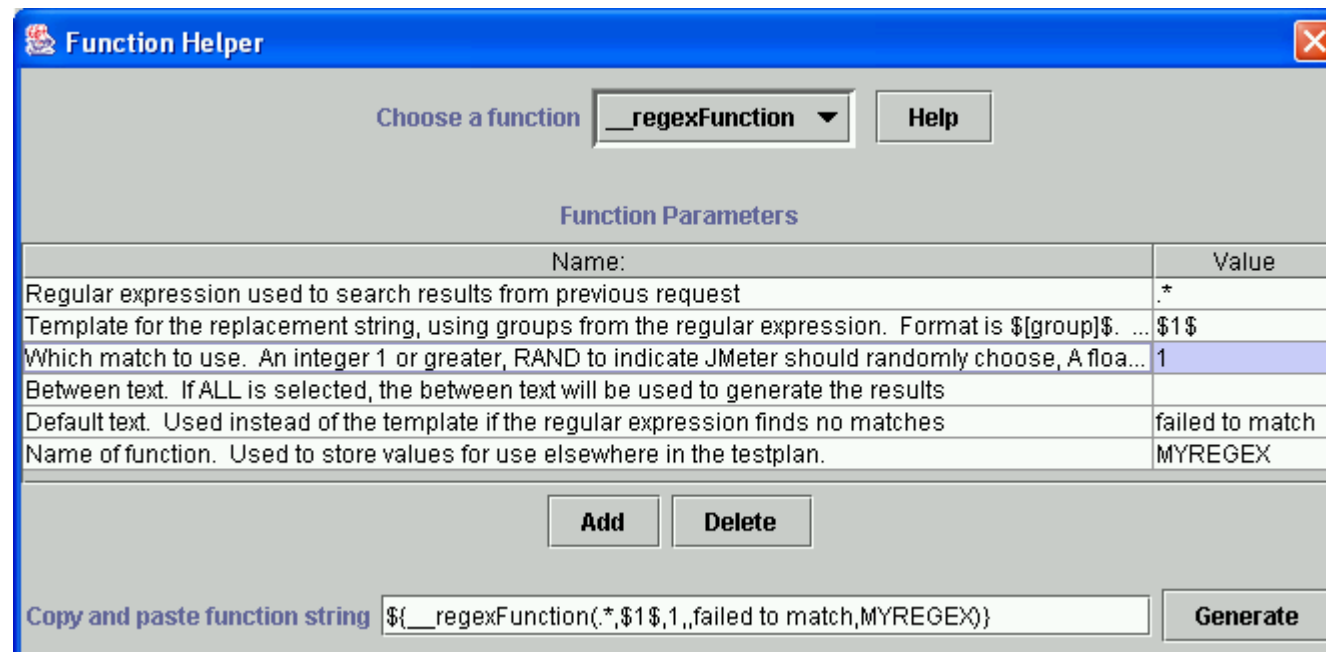
Avec les fonctions prédéfinies, il est possible de calculer de nouvelles valeurs lors de l'exécution à partir d'une réponse précédente, du temps ou d'autres sources.

Ces valeurs sont générées à chaque appel pendant toute la durée du test.

Les fonctions sont partagées par les threads mais chaque appel de fonction est traité par une instance séparé.

# Assistant pour les fonctions

Un assistant pour la mise au point de fonction est disponible via le menu



The screenshot shows the 'Function Helper' dialog box. At the top, there is a 'Choose a function' dropdown menu set to '\_regexFunction' and a 'Help' button. Below this is a section titled 'Function Parameters' containing a table with two columns: 'Name:' and 'Value'. The table lists several parameters for the regex function, with the third row highlighted. At the bottom of the table are 'Add' and 'Delete' buttons. Below the table is a text field labeled 'Copy and paste function string' containing the generated function call, and a 'Generate' button to its right.

Name:	Value
Regular expression used to search results from previous request	.*
Template for the replacement string, using groups from the regular expression. Format is \${group}\$. ...	_\${1}\$
Which match to use. An integer 1 or greater, RAND to indicate JMeter should randomly choose, A floa...	1
Between text. If ALL is selected, the between text will be used to generate the results	
Default text. Used instead of the template if the regular expression finds no matches	failed to match
Name of function. Used to store values for use elsewhere in the testplan.	MYREGEX

Copy and paste function string: `${__regexFunction(.*_${1}$,1,,failed to match,MYREGEX)}` Generate



# Fonctions informatives

---

**\_\_TestPlanName** : Le nom du plan de test courant

**\_\_threadNum** : Retourne le n° de thread

**\_\_samplerName** : Retourne le nom de l'échantillon

**\_\_machineIP** : Retourne l'IP de la machine locale

**\_\_machineName** : Retourne le nom de la machine

**\_\_time** : Retourne le timestamp courant en différents formats

**\_\_log, \_\_logn** : Trace un message





# Fichiers

---

***\_\_StringFromFile, \_\_StringToFile*** :  
Lecture/écriture de fichier

***\_\_FileToString*** : Lecture intégrale d'un  
fichier

***\_\_CSVRead*** : Lecture d'un fichier CSV

***\_\_XPath*** : Utilisation de Xpath pour lire  
des données dans un fichier XML



# Calculs et script

---

***counter*** : génère un compteur incrémental

***IntSum, longSum*** : ajout de nombres

***Random*** : Génère un nombre aléatoire

***RandomString*** : Génère une chaîne aléatoire

***UUID*** : Génère un identifiant unique

***BeanShell, Groovy, JavaScript*** : Exécution  
script

***jexl, jexl2*** : Evaluate une expression jexl



# Propriétés et variables

---

**\_\_property** ou **\_\_P** : Retourne la valeur d'une propriété

**\_\_setProperty** : Met à jour une propriété

**\_\_V** : Evaluate une variable 2.3RC3

**\_\_eval** : Evaluate une expression de variable

**\_\_evalVar** : Evaluate une expression stockée dans une variable



# Manipulation de chaînes

---

**\_\_split** : Sépare une chaîne en variables

**\_\_regexFunction** : Parse une réponse précédente avec une expression régulière

**\_\_escapeOroRegexChars** : Echappe les caractères spéciaux utilisés par les expressions régulières

**\_\_char** : Génère des caractères Unicode à partir d'une liste de nombres

**\_\_unescape** : Traite les String contenant les caractères d'échappement Java

**\_\_escapeHtml, \_\_unescapeHtml** : Encode/Décode les chaînes HTML

**\_\_uudecode, \_\_uuencode** : Encode/décode les URLs



# Restrictions d'usage

---

Sur le plan de test, seulement les fonctions suivantes ont un sens :  
*intSum, longSum, machineName, BeanShell, javaScript, jexl, random, time, property , log*

Sur un élément de configuration (traité par une thread séparé), *\_\_threadNum* ne fonctionne pas



# Passer des variables entre threads

---

Les variables JMeter sont locales à une thread. Pour s'échanger des données entre différentes threads quelque soit leur groupe d'unités, plusieurs techniques sont possibles :

- Utiliser des propriétés
- Utiliser des fichiers.  
Par exemple, un récepteur peut sauvegarder dans un fichier et un échantillon HTTP peut utiliser le protocole file pour lire le fichier plus un post-processeur pour extraire l'information
- Si les données à passer peuvent être déterminées au démarrage du test, un simple éléments source de données CSV peut suffire



# Concepts de base

---

Éléments d'un plan de test  
Enregistrement de scénarios  
Variables et fonctions

**Éléments de test créant des  
variables**



# Variables pré-définies

---

L'élément « **Variables pré-définies** » ou la propriété équivalente du plan de test permettent de définir et initialiser un ensemble de variables.

Quelque soit sa position dans le plan de test, Il est traité au démarrage du test. L'ensemble de variables résultant est copié dans chaque thread.

Il est donc recommandé

- de placer ces éléments juste en dessous d'un groupe d'unités ou du test plan
- De ne pas utilisé des fonctions qui génèrent des résultats différents à chaque appel. (Seul, le résultat du premier appel sera stocké dans la variable)





# Paramètres utilisateur

---

L'élément pré-processeur « **paramètres utilisateur** » permet de spécifier des valeurs individuellement aux threads

L'élément permet de spécifier une série de valeurs.

- Chaque thread reçoit une valeur de la série en séquence.
- Si il y a plus de threads que de valeurs, les valeurs sont réutilisées.

Les paramètres utilisateur sont référencés via  $\${paramètre}$

La configuration comporte un booléen « *Mettre à jour une fois par itération* » utile lorsque l'on utilise des fonctions.

- Soit la fonction est évaluée à chaque itération du contrôleur parent, soit elle est évaluée à chaque requête



# Configuration

## User Parameters

Name: User Parameters

☒ Update Once Per Iteration

### Parameters

Name:	User_1	User_2	User_3
username	user1	user2	user3
password	pass1	pass2	pass3
category	cat1	cat2	cat3
color	red	green	

Add Variable

Delete Variable

Add User

Delete User



# Sources de données CSV

L'élément de configuration « **Source de données CSV** » permet de lire les lignes d'un fichier CSV et de les séparer en variables.

- Les valeurs de l'ensemble des variables correspondant à une ligne sont passées individuellement à une thread
- Par défaut, le fichier est ouvert une seule fois et chaque thread utilise une ligne différente.
- L'ordre dans lequel les lignes sont passées aux threads dépend de leur ordre d'exécution (ce qui peut varier entre les itérations)
- Les lignes sont lues à chaque itération. Le nom du fichier est résolu à la première itération.

Cet élément est adapté pour gérer un nombre important de variables.

et est plus facile à utiliser que les fonctions `__CSVRead()` et `_StringFromFile()`.



# Attributs de configuration

---

**Noms des variables** : Définit les noms des variables mises à jour par le fichier CSV. Le champ peut être mis à vide et les noms de variable sont alors définies dans la première ligne du fichier CSV

**Délimiteur** : Le caractère délimiteur

**Autoriser les données avec les quotes** : Permet d'avoir le caractère délimiteur dans les données

**Recycler en fin de fichier** : A la fin de fichier, la lecture reprend au début

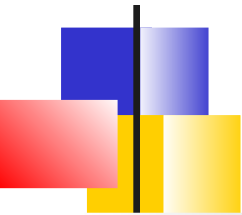
**Arrêter la thread à la fin de fichier** : Sans recyclage, provoque l'arrêt de la thread sur le EOF



# Mode de partage

L'attribut mode de partage peut prendre différentes valeurs :

- **Toutes les threads** (défaut) : Le fichier est partagé par toutes les threads.
- **Groupe d'unité courant** : Chaque fichier est ouvert une fois par groupe d'unités où il apparaît
- **Unité courante** : Chaque thread ouvre séparément le fichier. L'expression du fichier peut être *test\${\_\_threadNum}.csv*
- **Identifier** : Toutes les threads qui ont le même identifiant partage le même fichier.



# Données aléatoires unique

---

L'élément *CSV DataSet* est également utile lorsque l'on veut utiliser des valeurs aléatoires unique pour un test.

- Générer des valeurs aléatoires uniques à l'exécution peut être consommateur de ressources, c'est donc une bonne pratique de les préparer à l'avance
- Si nécessaire, les données aléatoires peuvent être combinées avec un paramètre d'exécution pour créer différents jeux de valeurs pour chaque exécution.



# Post-processeurs

---

Certains post-processeurs sont utiles pour créer des variables dont la valeur est lue dans la réponse d'un échantillon.

Plusieurs syntaxes sont possibles pour extraire la valeur de la réponse :

- Expression régulière
- JQuery
- Xpath
- JsonPath
- ...

Les post-processeurs sont exécutés après chaque échantillon de son contexte



# Configuration avec expressions régulières

---

## Appliqué sur

**Portée** / Champs de la réponse

L'**expression régulière** utilisée pour parser les données de la réponse. Elle doit contenir au moins un jeu de parenthèses (). Si plusieurs jeux de parenthèses, plusieurs groupes sont créés

**Nom de référence** : Le nom de la variable. Si plusieurs groupes, les variables sont nommées `<reference>_g#`

**Canevas** : Le canevas utilisé pour créer une String à partir de la correspondance. C'est une chaîne arbitraire avec des éléments faisant référence aux groupes de l'expression régulière : '\$1\$' référence le groupe 1, etc. . '\$0\$' référence tous les groupes.

**Match No.** Indique quel correspondance utiliser. 0 pour laisser JMeter choisir. Une valeur négative est possible

**Valeur par défaut** : Valeur utilisée si rien ne correspond à l'expression régulière





# Variables

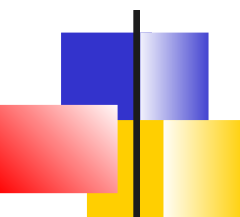
---

Si le n° de correspondance est un nombre positif et qu'il y a une correspondance, les variables sont positionnées comme suit :

- **refName** : La valeur du canevas
- **refName\_gn**, : Les groupes pour la correspondance
- **refName\_g** : Le nombre de groupes dans l'expression régulière
- Si il n'y a pas de correspondance, alors *refName* est positionné à la valeur par défaut et les variables *refName\_gn* et *refName\_g* sont supprimées

Si le n° de correspondance est un nombre négatif, les variables sont positionnées comme suit :

- **refName\_matchNr** : Le nombre de correspondance trouvées
- **refName\_n** : La valeur du canevas pour la nième correspondance
- **refName\_n\_gm** : Les groupes pour la nième correspondance
- **refName** – toujours à la valeur par défaut
- **refName\_gn** – pas positionnée



# Configuration avec *jQuery*

---

## **Appliqué sur**

L'**implémentation** de l'extracteur CSS/jQuery Jsoup (défaut) ou Jodd-Lagerto (CSSelly)

**Nom de référence** : Le nom de base pour construire les noms de variables

**L'expression CSS/jQuery** : L'expression jQuery retournant un ensemble de nœuds du DOM HTML

**Attribut** : le nom de l'attribut HTML à extraire des nœuds résultats. Si vide, c'est la valeur des nœuds et de ses enfants qui est retournée

**Nombres de correspondance** : **0** au hasard. ***n*** pour la *n*<sup>ième</sup> occurrence, **-1** pour toutes les occurrences

**Valeur par défaut** : La valeur par défaut si aucun nœud n'est trouvée



# Variables

---

Si le n° de correspondance est un nombre positif et qu'il y a une correspondance, les variables sont positionnées comme suit :

- **refName** : La valeur du canevas
- Si aucune correspondance : la valeur par défaut

Si le n° de correspondance est un nombre négatif, les variables sont positionnées comme suit :

- **refName\_matchNr** : Le nombre de correspondances
- **refName\_n** : La valeur du canevas pour la nième correspondance
- **refName** : Toujours la valeur par défaut



# Extracteur JSON

L'extracteur JSON permet d'extraire des données des réponses JSON à l'aide de la syntaxe ***JSON-PATH***.

Il fonctionne de façon similaire à l'extracteur JQuery ou autres

**Appliquer sur**

☐ L'échantillon et ses ressources liées ☒ L'échantillon ☐ Les ressources liées ☐ Nom de la variable à utiliser :

Noms des variables créées:

Expressions JSON Path:

Récupérer la Nième corresp. (0 : Aléatoire):

Calculer la variable de concaténation (suffix `_ALL`): ☐

Valeur par défaut:



# Propriétés prédéfinies

---

JMeter fournit automatiquement certaines propriétés en plus de celles définies dans *jmeter.properties*, *user.properties* ou la ligne de commande

- ***START.MS*** : Le timestamp de démarrage en ms
- ***START.YMD*** : Le timestamp de démarrage au format *yyyyMMdd*
- ***START.HMS*** : Le timestamp de démarrage au format *HHmmss*
- ***TESTSTART.MS*** : Le timestamp de démarrage en millisecondes



# Variables prédéfinies

---

JMeter définit également automatiquement certaines variables :

- ***COOKIE\_cookieName*** : Contient la valeur du cookie
- ***JMeterThread.last\_sample\_ok*** : Booléen indiquant si le dernier échantillon est OK. Mis à jour après les post-processeurs et les assertions



# Debug

L' **échantillon Debug** peut être utilisé pour afficher les valeurs des variables dans un récepteur « arbre de résultats »

La plupart des éléments de test inclut des traces de debug. Le debug d'un élément peut être activé via le **menu Aide**

Les traces peuvent être visualisées directement dans l'interface Jmeter (**Options > Afficher la console**)

- Par défaut, la console est désactivée, on peut l'activer via la propriété `jmeter.loggerpanel.display=true`



# *ForEach* Contrôleur

---

Le contrôleur ***ForEach*** permet d'effectuer une boucle sur un ensemble de variables d'entrée ayant le même préfixe et se terminant par un nombre.

=> A chaque itération, une variable de sortie est mise à jour avec la  $n^{\text{ième}}$  variable d'entrée

=> La variable de sortie peut être utilisée par les échantillons du contexte

Ce type de contrôleur est spécialement adapté pour s'exécuter avec les extracteurs configuré avec -1 dans le nombre d'occurrences



# Example

## ForEach Controller

Name: ForEach Controller

Comments:

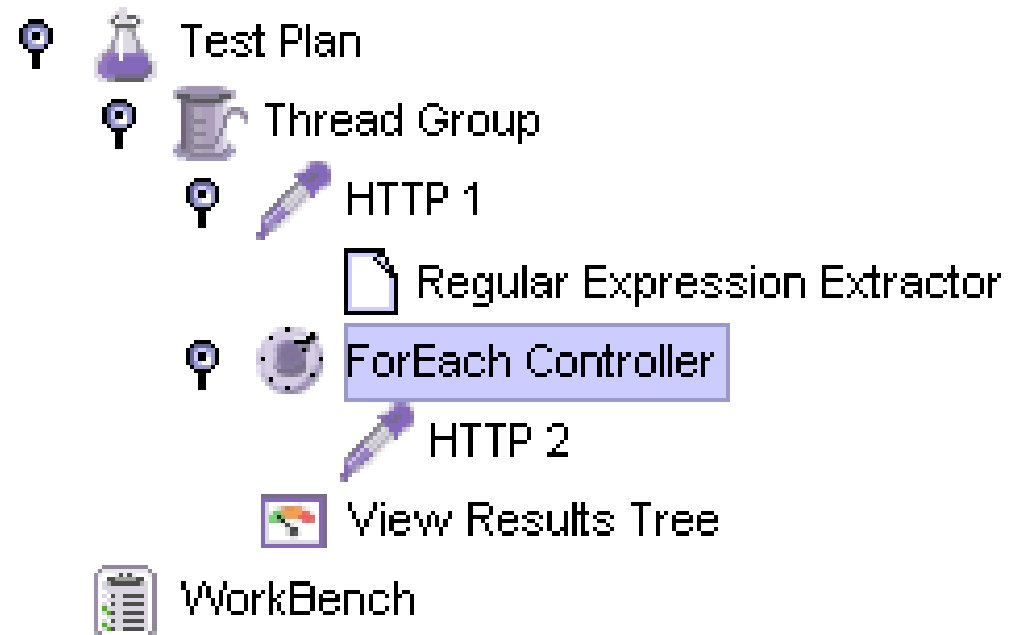
Input variable prefix inputVar

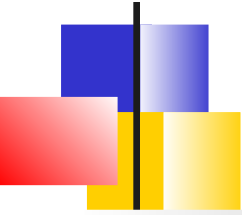
Start index for loop

End index for loop

Output variable name returnVar

☒ Add "\_" before number ?





# Boucle sur fichiers

## *Contrôleurs While et Loop*

---

Les contrôleurs While et Loop permettent de tester de boucler en fonction d'une condition

A l'intérieur de la boucle, on peut modifier la variable testée via un appel de fonction

Par exemple, faire appel à `__StringFromFile` pour lire une valeur dans un fichier



# Tests de performance

---

Compteurs  
Avant les tests  
Récepteur lors de l'exécution  
Rapport standard



# Les compteurs de temps

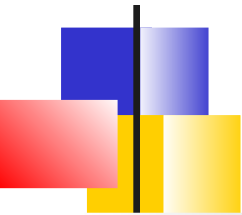


# Introduction

---

Comme l'objectif est de simuler une activité utilisateur réelle, il est nécessaire d'introduire des délais entre les requêtes correspondant au temps de réflexion (think time) d'un utilisateur.

Les délais sont introduits par les compteurs de temps



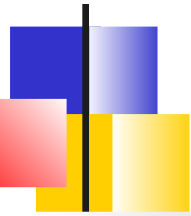
# Rappels

---

Les compteurs de temps sont traités avant chaque échantillon du contexte dans lequel il se trouve.

Si il y a plusieurs compteurs de temps dans le même contexte, les compteurs de temps s'additionnent

Les compteurs de temps ne sont traités seulement en conjonction avec un échantillon.



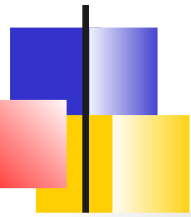
# Types de compteur de temps

**Compteur de temps constant** : Chaque thread est mis en pause la même durée

**Compteur de temps aléatoire uniforme** : chaque thread est mis en pause une durée aléatoire. Le nombre aléatoire est choisi entre 0 et une valeur max. Le délai est alors la somme de la valeur aléatoire et la valeur d 'offset. (Plutôt adapté au think time)

**Compteur de temps aléatoire selon la loi de Poisson** : introduit des pauses de durée variable. Le délai total est la somme de la valeur distribué selon la loi de Poisson plus un offset

**Compteur de temps aléatoire gaussien** : Chaque thread est mise en pause une durée aléatoire. La durée est égale à la valeur gaussienne (Moyenne de 0 et écart type de 1.0) multipliée par la déviation spécifiée plus un offset



# Types de compteur de temps

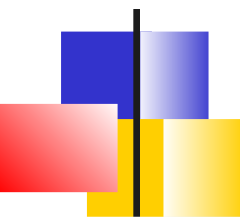
---

**Le compteur de temps débit constant** introduit des pauses variables. Les pauses sont calculées afin de garder le débit total (en termes d'échantillon par minutes) aussi près que possible qu'un nombre fourni.

**Le compteur de temps de synchronisation** bloque les threads jusqu'à ce que X threads soit bloquées et les redémarre en même temps. Provoque de fortes charges à certains moments du plan de test

**Les compteurs de temps BeanShell, BSF, JSR223.** Les délais sont générés avec des scripts





# Avant les tests



# Introduction

---

Avant de démarrer les tests de charge, il est nécessaire de s'assurer que le test fonctionne correctement

Le plan de test est alors adapté à cette phase afin qu'il contienne des éléments permettant de vérifier le bon déroulement du test.



# Recommandations

---

- ✓ Exécuter le premier test avec un utilisateur unique pour une première validation
- ✓ Observer le test pendant son exécution et analyser tout comportement suspect
- ✓ Utiliser le système manuellement durant l'exécution du test pour comparer ses observations aux résultats obtenus
- ✓ S'assurer de l'exactitude des résultats de tests et des métriques rassemblés
- ✓ Vérifier le contenu des pages retournées
- ✓ Exécuter un test qui boucle sur toutes les données d'entrées pour vérifier si il n'y a pas d'erreurs inattendues.
- ✓ Vérifier si vous pouvez réinitialiser le test (reproduire les mêmes conditions)
- ✓ A la fin du test, vérifier que la base de données a été mises à jour
- ✓ Après des erreur, nettoyer la base pour s'assurer de sa cohérence
- ✓ Répéter les tests en ajustant les variables comme les logins, les temps de pause
- ✓ Exécuter les tests dans des séquences différentes
- ✓ Ne pas utiliser les résultats des tests de validation dans le rapport final



# Données dynamiques

---

Pour qu'un test soit valide, il est important de faire varier les données d'entrées

- Utiliser les mêmes données d'entrées peut provoquer des usages artificiels de cache ou quelquefois des timeouts sur des verrous

L'idéal est de mettre en place des moyens afin que ses données d'entrées soit dynamiques. En général cela permet :

- de réduire la taille du script (paramétrer les URLs par exemple)
- de reproduire des bugs plus complexes
- de tester la réaction du système lorsqu'on lui fournit des valeurs d'entrées erronées



# Test paramétrés

---

Il est souvent utile de ré-exécuter le même test avec des configurations différentes.

- Une possibilité est de définir un ensemble de variables au niveau du plan de test et d'utiliser ces variables dans les éléments de test.

Par exemple, définir une variable BOUCLES et y faire référence dans le groupe d'unités via la syntaxe `${BOUCLES}`

- Ou même mieux faire en sorte que la variable puisse être mise à jour via la ligne de commande ou un fichier *.properties* .

La variable *BOUCLES* dans le plan de test est alors définie comme suit :

`BOUCLES=${__P(boucles,10)}` : Utiliser la propriété boucles ou 10 si elle n'est pas définie



# JMeter et la validation

---

JMeter propose plusieurs mécanismes pour valider des test :

- Le fichier de trace permet de détecter des anomalies
- Des récepteurs sont très appropriés pour surveiller la validité des réponses
- Un récepteur est dédié au debug et à la vérification des variables
- Les éléments assertions peuvent automatiser la validation d'une réponse



# Table de résultats

---

La table de résultats crée une ligne pour chaque échantillon indiquant :

- Le temps de réponse
- Le statut de la réponse
- La taille de la réponse
- La latence

Cela peut avoir une utilité pour repérer les échantillons suspects



# Arbre de résultats

---

L'arbre de résultats affiche un arbre de toutes les réponses des échantillons permettant de voir la réponse pour n'importe quel échantillon.

En plus de la réponse, il affiche le temps de réponse, le code retour, les entêtes http

On peut rechercher facilement un motif dans l'ensemble des vues qu'il propose (recherche Java)

=> Outil idéal pour la validation de test





# Mode de visualisation de la réponse

---

**Document** : Permet d'extraire le texte de documents au format divers (office, mp4, ...) Nécessite une librairie additionnelle (Apache Tika : *tika-app-x.x.jar*)

**HTML** : Une approximation de ce que pourrait rendre un navigateur. Les images, feuilles de style ne sont pas rapatriés

**HTML et ressources** : Rapatriement des ressources.

**JSON** : Vue en arbre et est adapté au Javascript.

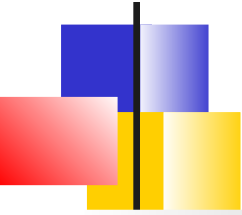
**Text** : Le texte contenu dans la réponse (défaut)

**XML** : Vue en arbre

**Testeur Regexp** : Montre le texte mais permet de tester ses expressions régulières



# Récepteurs lors de l'exécution



# Récepteurs et consommation mémoire

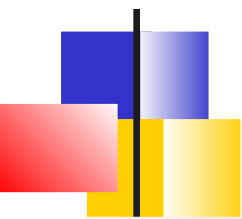
---

Les récepteurs peuvent utiliser beaucoup de mémoire si il y a beaucoup d'échantillons. La plupart des récepteurs gardent une copie de chaque échantillon en mémoire sauf :

- L'enregistreur de données
- Les récepteurs scripts (BeanShell/BSF/JSR223)
- La notification email
- Le moniteur de connecteur
- Le rapport résumé

Les récepteurs suivants ne gardent pas des copies de chaque échantillon mais agrègent des échantillons selon différents propriétés. Ils nécessitent moins de mémoire et peuvent éventuellement être utilisés :

- Rapport agrégé
- Graphe agrégé
- Graphe de distribution



# Enregistreur de données

---

Le récepteur enregistre les résultats dans un fichier mais pas dans l'interface utilisateur.

C'est le moyen le plus efficace pour enregistrer les données sans ajouter de la surcharge due à l'interface graphique

En mode non-GUI, l'option `-/` peut être utilisée. Les champs sauvegardés sont alors ceux définis dans les propriétés JMeter



# Configuration des données d'enregistrement

---

Les récepteurs peuvent être configurés pour sauvegarder différentes données dans les fichiers de résultat.

- Les valeurs par défaut sont définies dans la documentation.
- Certaines rubriques ne sont disponibles que dans un des 2 formats.



# Configuration de la sauvegarde des résultats

**Sauvegarder la configuration de la sauvegarde des échantillons**

<input type="checkbox"/> Enregistrer au format XML	<input checked="" type="checkbox"/> Libellé des colonnes (CSV)	<input checked="" type="checkbox"/> Horodatage
<input checked="" type="checkbox"/> Temps écoulé	<input checked="" type="checkbox"/> Libellé	<input checked="" type="checkbox"/> Code de réponse HTTP
<input checked="" type="checkbox"/> Message de réponse	<input checked="" type="checkbox"/> Nom d'unité	<input checked="" type="checkbox"/> Type de données
<input checked="" type="checkbox"/> Succès	<input checked="" type="checkbox"/> Messages d'erreur des assertions	<input checked="" type="checkbox"/> Nombre d'octets reçus
<input checked="" type="checkbox"/> Nombre d'octets envoyés	<input checked="" type="checkbox"/> Nombre d'unités actives	<input type="checkbox"/> URL
<input type="checkbox"/> Nom de fichier de réponse	<input checked="" type="checkbox"/> Latence	<input checked="" type="checkbox"/> Temps établissement connexion
<input type="checkbox"/> Encodage	<input type="checkbox"/> Nombre d'échantillon et d'erreur	<input type="checkbox"/> Nom d'hôte
<input checked="" type="checkbox"/> Temps d'inactivité	<input type="checkbox"/> Entêtes de requête (XML)	<input type="checkbox"/> Données d'échantillon (XML)
<input type="checkbox"/> Entêtes de réponse (XML)	<input type="checkbox"/> Données de réponse (XML)	<input checked="" type="checkbox"/> Sous résultats (XML)
<input checked="" type="checkbox"/> Résultats des assertions (XML)		

Fait



# Configuration

---

Les informations à sauvegarder lorsque l'on utilise l'option *-l* sont définies dans *jmeter.properties* ou (*user.properties*)

Pour changer le format par défaut :

**`jmeter.save.saveservice.output_format`**

Pour les informations maximales, choisir « *xml* » et « *Functional Test Mode* »



# Informations par défaut

---

Les informations par défaut sont :

- Un horodatage (le nombre de ms depuis le 1<sup>er</sup> janvier 1970)
- Le type de données
- Le nom de la thread
- Le libellé
- Le temps de réponse
- Le message,
- Le code
- Un indicateur de succès





# Example

---

```
jmeter.save.saveservice.output_format=csv
jmeter.save.saveservice.assertion_results_failure_message=true
jmeter.save.saveservice.default_delimiter=|
-----
timestamp|time|label|responseCode|threadName|dataType|success|
failureMessage
02/06/03 08:21:42|1187|Home|200|Thread Group-1|text|true|
02/06/03 08:21:42|47|Login|200|Thread Group-1|text|false|Test Failed:
expected to contain: password etc.
```



# Attributs XML

---

**by, sby** : Octets reçus / envoyés

**de, dt** : Data encoding / type

**ec** : Nombre d'erreurs (0 ou 1 en général)

**hn** : Host qui a généré l'échantillon

**lt** : Idle Time = Temps non passé dans l'échantillonnage en millisecondes) (en général 0)

**lb** : Libellé de l'échantillon

**lt** : Latency = Temps pour la réponse initiale (Pas supporté par tous les échantillonneurs)

**ct** : Connect Time = Temps pour établir la connexion (Pas supporté par tous les échantillonneurs)



# Attributs XML (2)

---

***na, ng*** : Nombre de threads actives pour tous les groupes de threads, pour le groupe de l'échantillon

***rc, rm*** : Code réponse (e.g. 200), message de réponse (e.g. OK)

***s*** : Success flag (true/false)

***sc*** : Nombre d'échantillons (1 à moins que ce la soit un échantillon d'agrégation)

***t*** : Temps écoulé depuis le démarrage du test (millisecondes)

***tn*** : Nom de la thread

***ts*** : timeStamp (millisecons depuis le 1er Janvier 1970)

***varname*** : Valeur d'une variable



# Notification email

---

Le récepteur notification email peut être configuré pour envoyer un email si le test reçoit trop de réponses en échec de la part du serveur



# Rapport consolidé

---

Le rapport consolidé présente les résultats en tableau. Il crée une ligne pour chaque requête nommée différemment

Le tableau contient les colonnes suivantes :

- **Label** : Le nom de l'échantillon
- **Moyenne** : La moyenne du temps de réponse pour cet échantillon
- **Min** : La valeur minimale
- **Max** : La valeur maximale
- **Ecart-type** : L'écart type
- **Error %** : Pourcentage de requête en erreur
- **Débit** : Nombre de requêtes par seconde/minute/heure. (prend en compte les compteurs de temps)
- **Kb/sec** : Le débit en Kilobytes par seconde
- **Moyenne de la taille** : La taille moyenne de la réponse en octets



# Générer les rapports consolidés

---

L'élément « **Générer les rapports consolidés** » peut être placé n'importe où dans le plan de test.

Il génère ses données dans le fichier de log ou la sortie standard.

La sortie est générée régulièrement (par défaut toutes les 3 minutes) et affiche les totaux courants et les différences avec les précédents totaux

Cet élément est automatiquement ajouté lors d'une exécution en batch



# Exemple

---

label +	263	in	31.0s	=	8.5/s	Avg:	1138	Min:	1000	Max:	1250	Err:	0
	(0.00%)												
label =	697	in	80.3s	=	8.7/s	Avg:	1136	Min:	1000	Max:	1250	Err:	0
	(0.00%)												
label +	109	in	12.4s	=	8.8/s	Avg:	1092	Min:	47	Max:	1250	Err:	0
	(0.00%)												
label =	806	in	91.6s	=	8.8/s	Avg:	1130	Min:	47	Max:	1250	Err:	0
	(0.00%)												

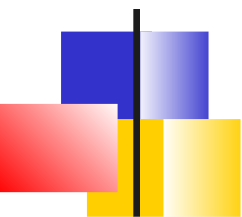
Le "label" est le nom de l'élément.

Le signe "+" indique une ligne différence et affiche les changements par rapport au dernier résultat

- L'exemple "806 in 91.6s = 8.8/s" signifie qu'il y a eu 806 échantillons enregistrés en 91.6 secondes, ce qui donne 8.8 échantillons/seconde.

Les temps moyen, min. et max sont exprimés en millisecondes.

"Err" indique le nombre d'erreur



# Rapport standard





# Définitions statistiques

---

**Moyenne** : Utiliser les moyennes des temps de réponses avec prudence et accompagner ses résultats avec la valeur minimale, maximale et l'écart type.

**Percentile** : Le percentile peut être utilisée seul seulement lorsqu'il représente des données qui sont uniformément ou normalement distribuée avec un nombre acceptable de valeurs extrêmes

Une valeur **médiane** est simplement la valeur milieu d'un jeu de données ordonnées de plus bas au plus haut

Valeur **normale** : Celle qui arrive le plus souvent



# Définitions statistiques (2)

---

**Ecart-type** (Standard deviation) : L'écart type indique la densité de points autour d'une valeur moyenne. Plus l'écart type est petit, plus les données sont consistantes

**Distribution uniforme** (ou linéaire) représente une collection de données uniformément espacées entre les limites supérieure et inférieure. Les distributions uniformes sont fréquemment utilisées pour modéliser les délais utilisateur mais ne sont pas communes dans les résultats des temps de réponse

**Distribution normale ou Gaussienne** : les données sont pondérées vers le centre (ou valeur médiane). Les temps de réponse pour les applications Web sont fréquemment distribués normalement.



# Recommandations

---

A la place d'utiliser des calculs mathématiques rigoureux, le sens commun est généralement suffisant :

1. Si plus de 20 % des résultats des exécutions ne sont pas similaires, il y a sûrement un problème : l'application ou le test
2. Si un 90ème percentile est plus grand que le maximum ou moins que le minimum des autres exécutions. Le jeu de données n'est probablement pas similaire
3. Si les mesures d'un test sont notablement plus grande ou plus basse que des résultats d'autres tests. Le jeu de données n'est probablement pas similaire
4. Si le temps de réponse d'une seule URL est notablement différente et que les autres sont similaires. Le jeu de données est probablement similaire



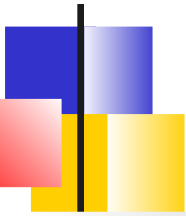
# Récepteurs

---

Les récepteurs donnant des résultats intéressants à inclure dans le rapport de test sont des récepteurs consommant trop de ressources pour les activer pendant le test.

Ces récepteurs doivent plutôt être utilisés « offline » à partir de fichiers de résultats sauvegardés.

Leur utilisation est moins intéressante depuis la version 3.x



# Génération de rapport HTML

---

Depuis la version 3+, possibilité de générer un rapport HTML

**=> Tous les récepteurs des versions antérieures sont dépréciés**

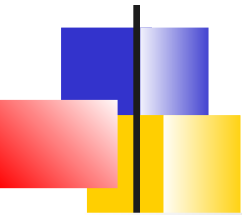


# Contenu du rapport

---

Le générateur de rapport fournit les informations suivantes :

- **APDEX** : Standard de présentation basé sur des seuils configurables
- Un graphique de **résumé** des échantillons ayant réussi ou échoués
- Une tableau statistique pour **chaque** échantillon
- Un tableau **d'erreurs**
- Les **top** 5 erreurs
- Des graphiques **zoomables**, ...
-



# APDEX

L'objectif est de fournir un seul métrique clair prenant en compte la satisfaction utilisateur.

La valeur varie entre 0 (Aucun utilisateur satisfait) et 1 (Tous les utilisateurs sont satisfaits)

Il faut définir un seuil de satisfaction pour le temps de réponse duquel découle un seuil de tolérance (4 fois le seuil de satisfaction)

$$Apdex_t = \frac{SatisfiedCount + \frac{ToleratingCount}{2}}{TotalSamples}$$



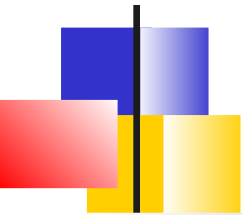
# Tableau Statistiques

---

Pour chaque échantillon, les informations statistiques disponibles sont :

- Nombre d'exécutions :
  - Nombre d'échantillons
  - Nombre ayant échoué
  - Nombre ayant réussi
- Temps de réponse :
  - Moyenne
  - Min/Max
  - 3 percentiles configurable
  - Débit
- Utilisation du réseau (kb/s)
  - Reçus
  - Envoyés





# Erreurs

---

Un tableau résumé rassemble les informations sur les erreurs :

- Type de l'erreur (Code HTTP, SocketTimeout, Pas de code réponse, réponse null, erreur d'assertion, ...)
- Nombre
- Pourcentage parmi les erreurs
- Pourcentage parmi tous les samples

Un autre tableau affiche les 5 échantillons ayant provoqué le plus d'erreur



# Graphiques en fonction du temps

---

Le rapport propose plusieurs graphiques ayant pour abscisse le temps. Ils sont zoomables :

- Evolution des temps de réponse moyen pour chaque échantillon
- Evolution des percentiles (tout échantillon confondu)
- Evolution des threads actives
- Evolution du débit (kb/s)
- Evolution de la latence moyenne pour chaque échantillon
- Evolution du temps de connexion moyen
- Evolution du nombre de hits par secondes
- Evolution du nombre de hits par secondes pour chaque code http
- Evolution du nombre de transactions (inclut les contrôleurs de transactions)

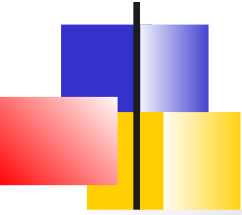


# Autres graphiques

---

D'autres graphiques sont zoomables :

- Corrélation du temps de réponse avec le nombre de requête par secondes
- Corrélation de la latence avec le nombre de requête par secondes
- Histogramme des requêtes groupé par des intervalles de temps de réponse
- Les percentiles des temps de réponse (de 0 à 100) pour chaque échantillon
- Corrélation des temps de réponse avec le nombre de threads actives
- Graphique de distribution des temps de réponse



# Configuration de la génération

---

Toutes les propriétés de génération  
peuvent être trouvées dans  
***reportgenerator.properties***

On peut les customiser en copiant les  
propriétés dans *user.properties*



# Configuration pour le csv d'entrée nécessaire (défaut)

---

```
jmeter.save.saveservice.bytes = true
# Only available with HttpClient4
#jmeter.save.saveservice.sent_bytes=true
jmeter.save.saveservice.label = true
jmeter.save.saveservice.latency = true
jmeter.save.saveservice.response_code = true
jmeter.save.saveservice.response_message = true
jmeter.save.saveservice.successful = true
jmeter.save.saveservice.thread_counts = true
jmeter.save.saveservice.thread_name = true
jmeter.save.saveservice.time = true
jmeter.save.saveservice.connect_time = true
# the timestamp format must include the time and should include the date.
# For example the default, which is milliseconds since the epoch:
jmeter.save.saveservice.timestamp_format = ms
# Or the following would also be suitable
jmeter.save.saveservice.timestamp_format = yyyy/MM/dd HH:mm:ss
```



# Quelques configurations générales

---

***report\_title*** : Le titre du rapport (par défaut : "Apache JMeter Dashboard")

***date\_format*** : Par défaut : yyyyMMddHHmmss. Possibilité d'utiliser un fuseau horaire (zzz)

***apdex\_satisfied\_threshold*** : Seuil de satisfaction APDEX

***apdex\_tolerated\_threshold*** : Seuil de tolérance APDEX

***jmeter.reportgenerator.apdex\_per\_transaction*** : Seuils apdex par transaction.

Le format est : *sample\_name:satisfaction|tolerance[;]*

Possibilité d'utiliser des expressions régulières

***sample\_filter*** : Expressions régulières pour filtrer les échantillons

***aggregate\_rpt\_pct1, aggregate\_rpt\_pct2 , aggregate\_rpt\_pct3*** : Percentiles utilisés dans le graphe résumé



# Génération du rapport

---

On peut demander la génération du rapport lors du démarrage du tir de charge. Le rapport est alors généré à la fin de l'exécution du test

```
jmeter -n -t <test JMX file> -l <test log file> -e -o  
<Path to output folder>
```

Il peut également être généré à posteriori à partir d'un fichier résultat au format CSV

```
jmeter -g <log file> -o <Path to output folder>
```



# Tests fonctionnels

---

Introduction  
Expressions régulières  
Types d'assertions  
Récepteurs pour les tests fonctionnels





# Introduction

---

JMeter peut être utilisé pour du test fonctionnel

Les éléments de tests fonctionnels (assertions) peuvent être intégrés dans le plan de test. Cependant, ils peuvent avoir un impact sur les performances.

Plutôt que de mélanger les genres, il est recommandé de séparer les plans de test dédiés à la performance et les plans de tests dédiés aux tests fonctionnels

En effet, pour les tests fonctionnels, il n'est pas nécessaire de charger le serveur, il n'est même pas nécessaire de répliquer l'environnement de production et ses tests peuvent être exécutés sur une machine développeur

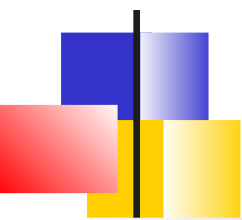


# Check-box

L'élément plan de test a une case à cocher  
« **Mode de test fonctionnel** »

- Si elle est cochée, Jmeter enregistre toutes les données retournées du serveur pour chaque échantillon.
- Si un récepteur utilise un fichier. Celui-ci va grossir très rapidement et provoquer des problèmes de performance.

Cette option est donc adaptée à des tests courts qui valide un test ou qui permette de s'assurer de la validité des réponses du serveur.

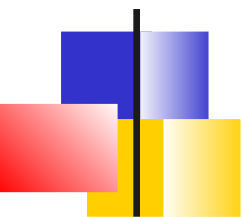


# Cas d'usage des assertions

---

Validation ou debugging d'un test de performance : « Durée », « Taille », « Compare »

Tests fonctionnels : « Réponse », « Taille », « XML » « Script », « MD5Hex », « HTML » , « XPath », « XML Schema »



# Types d'assertions



# Assertions

---

Les assertions sont utilisées pour exécuter des vérifications additionnelles sur les échantillons et sont traitées après chaque échantillon du même contexte.

- Pour être sûr qu'une assertion ne s'applique qu'à un seul échantillon, il suffit de l'ajouter comme enfant .



# Types d'assertion

---

L'assertion **réponse** permet de définir des motifs de chaîne de caractères à rechercher dans différents champs de la réponse

L'assertion **durée** teste que la réponse est reçue dans un temps donné

L'assertion **taille** teste que la réponse a le bon nombre d'octets

L'assertion **XML** teste que la réponse est un document XML correct

Les assertions **BeanShell**, **JSR223** permette d'effectuer des assertions via script

L'assertion **MD5Hex** permet de vérifier le code de hachage MD5 de la réponse



# Types d'assertion (2)

---

L'assertion **HTML** permet de vérifier la syntaxe HTML avec JTidy

L'assertion **XPath** teste que la réponse est un document correct et validé vis à vis d'une DTD ou validé par jTidy et tester pour une expression XPath

L'assertion **XML Schema** permet de valider la réponse vis à vis d'un schéma XML

L'assertion **Compare** peut être utilisée pour comparer les résultats des échantillons. Soit le contenu, soit le temps est comparé. Le contenu peut être filtré avant la comparaison

L'assertion **SMIME** peut évaluer des résultats de l'échantillonneur Mail Reader



# Assertion Réponse

---

## Les motifs de chaîne de caractères

- Contient, Correspond à: expression régulière
  - "contient" signifie que l'expression régulière correspond au moins à une partie du texte cible  
'alphabet' « contient » 'ph.b.'
  - "correspond à" signifie que l'expression régulière correspond à l'intégralité du texte cible  
'alphabet' « correspond à » 'al.\*t'.
  - Par défaut, le motif est en mode multi-lignes et sensible à la casse
- Est égale à, contient: Java, sensible à la casse





# Autres champs de Assertion Réponse

---

**Appliquer sur** : Utilisé pour les échantillonneurs qui utilise des sous-échantillons. Plusieurs choix sont possibles :

- L'échantillon principal seulement
- Les sous-échantillons seulement
- Tous les échantillons
- Une variable

**Section de réponse à tester** : Le texte de la réponse, le document extrait par Apache Tika, l'URL, Le code réponse, le message HTTP, les entêtes HTTP (incluant les cookies)

**Ignore status** : Si le code réponse doit être ignoré pour la validité du test

**Les motifs** : Liste de motifs à tester. Si un motif échoue, les autres ne sont pas testés



# Assertion taille

---

L'assertion taille teste si chaque réponse a le bon nombre d'octets

Il peut de spécifier une valeur et un opérateur de comparaison ( $=$ ,  $>$ ,  $<$ ,  $\neq$ )

Il comporte également le champ « Appliqué sur »



# Assertion HTML

---

Cet élément vérifie la syntaxe HTML de la réponse en utilisant Jtidy

Options de configuration :

- **doctype** : omit/auto/strict/loose
- **Format** HTML, XHTML or XML
- **Seulement les erreurs** : Note juste les erreurs
- **Seuil d'erreur** : Le nombre d'erreurs déclarant la réponse invalide
- **Seuil d'avertissement** : Le nombre d'avertissement déclarant la réponse invalide
- **Nom de fichier** : Nom du fichier où le rapport est écrit



# Assertion XML

---

2 éléments Assertion sont dédiés au XML :

- Assertion XML : Valide que le document XML est bien formé
- Assertion Schéma XML : Valide que le document est bien formé et qu'il respecte un schéma particulier



# Assertion XPath

---

Validation : Bien formé, DTD ou JTidy + Xpath

Supporte les expressions booléennes :

*count(//\*error)=2*

Configuration

- Appliqué sur
- Analyse XML : JTidy ou Validation XML
- XPath : Possibilité d'inverser le résultat du test



# Assertions JSON

---

Permet d'indiquer une expression *JSONPath* permettant de valider la réponse.

- Vérifie que le chemin existe
- Et qu'il a telle valeur



# Script Assertion

---

Les assertions sont alors vérifiées par script.

Dans les dernières versions :

- BeanShell (<http://www.beanshell.org/>) Depuis 3.x :
- JSR223

JMeter fournit des données d'entrée :  
paramètres, variables, propriétés

Le script peut manipuler les données et en particulier positionner le statut de l'assertion



# Configuration de l'assertion script JSR223

---

**Langage** : Langage à utiliser. Par défaut Groovy, syntaxe identique à Java

**Paramètres** à passer au script. Les paramètres sont stockés dans les variables suivantes :

- ***parameters*** : Chaîne contenant les paramètres dans une variable unique
- ***args*** : tableau de chaîne contenant les paramètres, la division étant faite sur le caractère espace

**Nom du fichier** : Un fichier contenant le script à exécuter





# Données du script

---

**log** - (Logger) : Peut être utilisé pour écrire dans le fichier de log

**Label** : Le nom de l'assertion

**Filename** : Le nom du fichier de script

**ctx** : ( JMeterContext ) Accès au context JMeter

**vars** : ( JMeterVariables ) Accès aux variables JMeter: *vars.get(key);*  
*vars.put(key,val); vars.putObject("OBJ1",new Object());*  
*vars.getObject("OBJ2");*

**props** : (JMeterProperties - class java.util.Properties) :  
*props.get("START.HMS"); props.put("PROP1","1234");*

**SampleResult, prev** - ( SampleResult ) : Accès au précédent résultat

**sampler** - (Sampler) : Accès à l'échantillonneur courant

**OUT** - System.out :. *OUT.println("message")*

**AssertionResult** : Le résultat - *AssertionResult.setFailure()*

...



# Exemple

---

```
if (ResponseCode != null && ResponseCode.equals ("200") == false ) {
    Failure=true ;
    FailureMessage ="Response code was not a 200 response code it was " + ResponseCode + "." ;
    print ( "the return code is " + ResponseCode);    // this goes to stdout
    log.warn( "the return code is " + ResponseCode); // this goes to the JMeter log file
} else {
    try {
        int nbMembres = Integer.parseInt(vars.get("NB_MEMBRES"));
        int expected = Integer.parseInt(props.get("TC1.EXPECTED"));

        if (nbMembre != expected)
        {
            Failure= true ;
            FailureMessage = "The number of members is not as exepected" ;
        }
    } catch ( Throwable t ) {
        print ( t ) ;
        log.warn("Error: ",t);
    }
}
```



# Assertion de comparaison

---

L'assertion ***Comparaison*** peut être utilisée pour comparer les résultats des échantillons du même contexte.

Soit le contenu, soit les temps de réponse sont comparés.

Le contenu à comparer peut être modifier avant la comparaison en utilisant des expressions régulières



# Configuration

---

**Contenu** : Si il faut comparer le contenu de la réponse

**Temps de réponse** : Si la valeur  $\geq 0$ , alors vérifie que la différence des temps de réponse n'est pas plus grande

**Filtres de comparaison** : Peuvent être utilisés pour supprimer des chaînes de la comparaison.

Par exemple, si la page contient un timestamp. Il peut être reconnu via l'expression : "Time: \d\d:\d\d:\d\d" et remplacé avec une valeur statique : "Time: HH:MM:SS".



# Récepteurs pour les tests fonctionnels



# Récepteurs pour les assertions

---

Trois récepteurs sont pratiques pour les tests fonctionnels :

- Le **récepteur d'assertion** qui s'applique à toutes les assertions du contexte sauf les comparaisons
- Le **récepteur d'assertion** pour les comparaisons
- L'**arbre de résultat** qui indique en rouge les URLs n'ayant pas vérifié les assertions



# Récepteurs d'assertion

Il permet de visualiser toutes les assertions qui ont échouées.

**Assertion Results**

Name:

**Write All Data to a File**

Filename   ☐ Log Errors Only

**Assertions:**

Sample 1 - 1	The result was the wrong size: It was 5 bytes, but should have been greater than 100 bytes.
Sample 1 - 2	The result was the wrong size: It was 5 bytes, but should have been greater than 100 bytes.
Sample 1 - 3	The result was the wrong size: It was 5 bytes, but should have been greater than 100 bytes.

# Récepteurs d'assertion pour les comparaisons

**Comparison Assertion Visualizer**

**Name:**

**Comments:**

**Write results to file / Read from file**

**Filename**  **Browse...** **Log/Display Only:** ☐ Errors ☐ Successes **Configure**

 HTTP 1	GET http://www.apache.org/	GET http://www.apache.org/
 HTTP 2	[no cookies] Connection: keep-alive	[no cookies] Connection: keep-alive
	Response Time: 1184	Response Time: 636





# Sauvegarder les réponses dans un fichier

---

Cet élément peut être placé n'importe où dans le plan de test.

Pour chaque échantillon de son contexte, il crée un fichier avec les données de la réponse

Le nom de fichier est créé avec un préfixe spécifié plus un nombre.

L'extension est déduite du mime-type



# Pour aller plus loin

---

Scripting  
Automatisation et Intégration continue  
Architecture maître esclave



# Scripting



# JSR 223

Depuis la version 3, JMeter recommande l'utilisation d'un langage de script respectant la JSR 223

Ces langages sont compilables et peuvent donc améliorer la performance. La compilation s'effectue si :

- On utilise des fichiers séparés (à la différence de coder inline dans le jmx). Le résultat de la compilation est alors caché par JMeter
- On utilise le dévpt inline mais on coche l'option *Cache compiled script if available*

Lors de l'utilisation du cache, ne pas utiliser les variables JMeter directement mais plutôt les passer en paramètres du script.



# Utilisation des scripts

---

Les scripts peuvent remplacer presque n'importe quel éléments :

- Un échantillon
- Une assertion
- Un compteur de temps
- Un pré-processeur
- Un post-processeur
- Un récepteur



# Principes communs

---

Le principe commun pour tous les éléments scripts est :

- De définir ses paramètres d'entrée
- Dé définir le fichier ou le code inline
- Utiliser des variables prédéfinies de JMeter (logger, table des propriétés, etc..)

Ensuite en fonction du type de l'élément, des variables spécifiques supplémentaires sont utilisables



# Variables communes

---

**log** - (Logger) : Peut être utilisé pour écrire dans le fichier de log

**Label** : Le nom de l'assertion

**Filename** : Le nom du fichier de script

**ctx** : ( JMeterContext ) Accès au context JMeter

**vars** : ( JMeterVariables ) Accès aux variables JMeter: *vars.get(key);*  
*vars.put(key,val); vars.putObject("OBJ1",new Object());*  
*vars.getObject("OBJ2");*

**props** : (JMeterProperties - class java.util.Properties) :  
*props.get("START.HMS"); props.put("PROP1","1234");*

**SampleResult, prev** - ( SampleResult ) : Accès au précédent résultat

**sampler** - (Sampler) : Accès à l'échantillonneur courant

**OUT** - System.out :. *OUT.println("message")*

...



# Cas d'un échantillon

---

Un échantillon script peut manipuler la variable ***SampleResult***

<http://jmeter.apache.org/api/org/apache/jmeter/samplers/SampleResult.html>

Les méthodes disponibles :

*SampleResult.setData(data) // Les données de la réponse*

*SampleResult.setSuccessful(true/false)*

*SampleResult.setResponseCode("code")*

*SampleResult.setResponseMessage("message")*





# Cas d'une assertion

---

Le script peut vérifier différents aspects de ***SampleResult***.

Si une erreur est détectée, le script doit utiliser :

```
AssertionResult.setFailureMessage("message")
```

```
AssertionResult.setFailure(true)
```



# Cas d'un compteur

---

Le script doit retourner le nombre de millisecondes à attendre.



# Mise au point de fonctions via script

Via les échantillons scripts, JMeter permet d'utiliser des scripts comme **fonctions**

- Créer un simple plan de test avec l'échantillonneur JSR223 et le récepteur Arbre de résultats
- Coder le script et le tester en exécutant le test. Si il y a des erreurs, elles apparaîtront dans le récepteur. Le résultat du script sera également visible dans la réponse
- Lorsque le script s'exécute correctement, il peut être stocké comme variable du plan de test. La variable peut être utilisée pour l'appel du script. Exemple :  
**`${_groovy(${SCRIPT_VAR})}`** .



# Automatisation



# Automatisation

Les problèmes de performance sont comme tous les autres type de bug : plus ils sont détectés tard plus le coût de leur résolution est important.

Il est donc assez légitime d'automatiser les tests de performance et de charge afin que les goulots d'étranglement qui dégradent les performances ne deviennent difficiles à corriger.

Concernant, l'automatisation JMeter offre une intégration Ant (mais pas Maven)



# Maven JMeter

---

Un plugin Maven pour *JMeter* (pas édité par Apache) existe

```
<groupId>com.lazerycode.jmeter</groupId>  
<artifactId>jmeter-maven-plugin</artifactId>
```



# Exemple pom.xml

```
<plugin>
  <groupId>com.lazerycode.jmeter</groupId>
  <artifactId>jmeter-maven-plugin</artifactId>
  <version>1.9.0</version>
  <configuration>
    <testResultsTimestamp>false</testResultsTimestamp>
    <overrideRootLogLevel>DEBUG</overrideRootLogLevel>
    <suppressJMeterOutput>false</suppressJMeterOutput>
    <ignoreResultFailures>true</ignoreResultFailures>
    <propertiesUser>
      <webservice.host>${performancetest.webservice.host}</webservice.host>
      <webservice.port>${performancetest.webservice.port}</webservice.port>
      <webservice.path>${performancetest.webservice.path}</webservice.path>
      <webservice.connectTimeout>${performancetest.connectTimeout}</webservice.connectTimeout>
      <webservice.responseTimeout>${performancetest.responseTimeout}</webservice.responseTimeout>
      <threadCount>${performancetest.threadCount}</threadCount>
      <loopCount>${performancetest.threadCount}</loopCount>
    </propertiesUser>
    <propertiesJMeter>
      <jmeter.save.saveservice.thread_counts>true</jmeter.save.saveservice.thread_counts>
    </propertiesJMeter>
    <jmeterPlugins>
      <plugin>
        <groupId>kg.apc</groupId>
        <artifactId>jmeter-plugins</artifactId>
      </plugin>
    </jmeterPlugins>
  </configuration>
  <executions>
    <execution>
      <id>execute-jmeter-tests</id>
      <goals>
        <goal>jmeter</goal>
      </goals>
      <phase>verify</phase>
    </execution>
  </executions>
```



# Intégration Jenkins

---

L'intégration de test de performance dans Jenkins pourrait être :

1. Installer JMeter sur Jenkins et mettre au point des fichiers JMX dans le dépôt GIT
2. Installer le plugin « Performance » pour lire les logs jMeter
3. Après une phase de déploiement, lancer les tests via un script sh
4. Ajouter une action post-build de publication des résultats, indication à Jenkins les emplacements des fichiers logs de JMeter
5. Indiquer les valeurs seuils qui font échouer le build





# Tests distribuídos



# Introduction

---

Alléger la charge d'une machine cliente

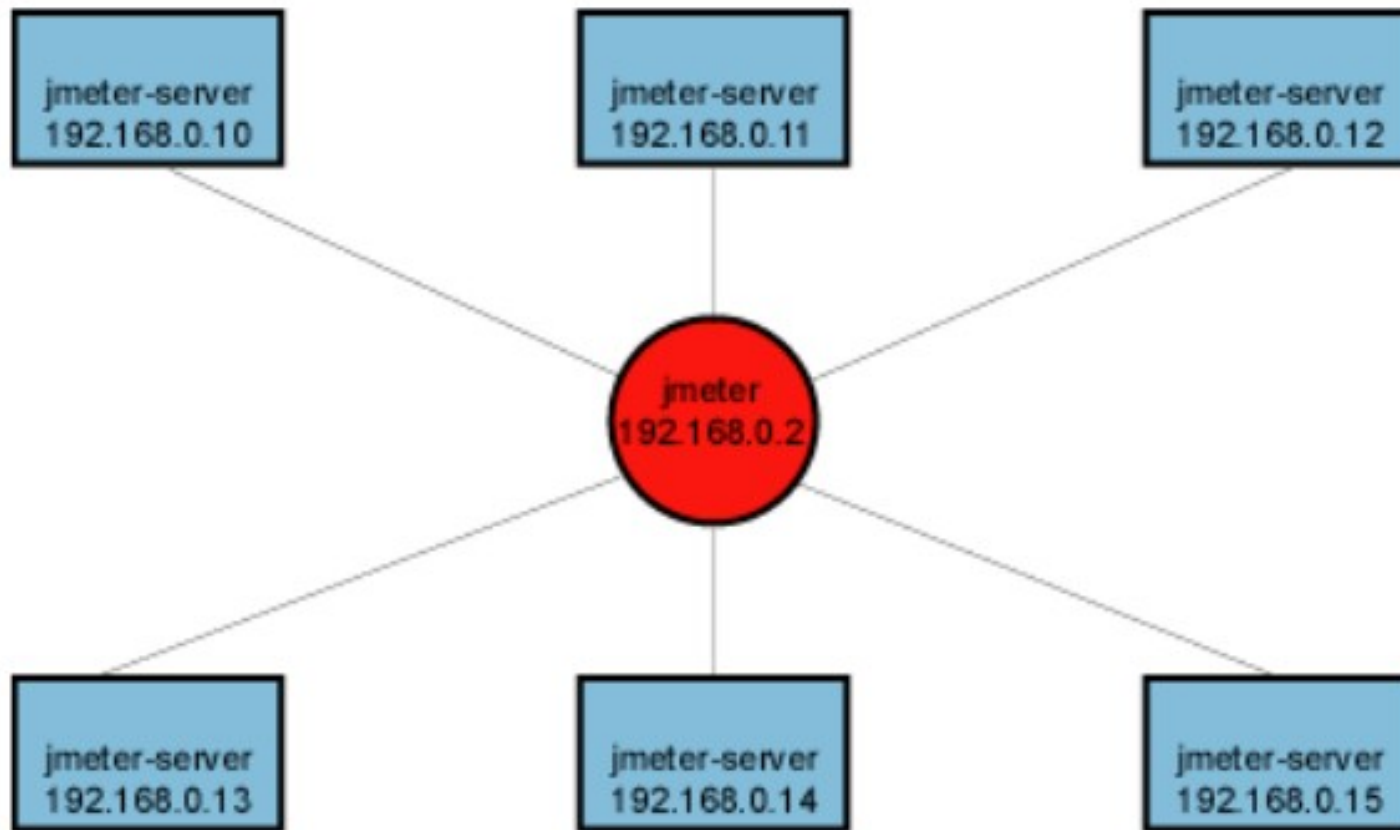
=> Pouvoir simuler de plus grosses charges sur le serveur

L'architecture consiste en une machine maître qui démarre les tests sur plusieurs machines esclaves

La machine maître exécute l'interface graphique qui contrôle le test

Les machines esclaves exécute *jmeter-server* qui reçoit les commandes du maître et envoie les requêtes vers le serveur cible

# Architecture





# Vérifications

---

Avant de démarrer des test utilisant cette architecture, plusieurs vérifications sont nécessaires

1. Les firewalls sur les systèmes sont désactivés
2. Tous les machines sont dans le même sous-réseau ou les machines clientes utilisent les adresses 192.x.x.x ou 10.x.x.x
3. Toutes les machines utilisent la même version de JMeter



# Etapes

1. Sur les machines esclaves, démarrer ***jmeter-server[.bat]***
2. Sur la machine maître, éditer jmeter.properties et renseigner les adresses des esclaves dans la propriété remote\_hosts.  
Exemple :

***remote\_hosts=192.168.0.10,192.168.0.11,192.168.0.12***

3. Démarrer jmeter sur la machine maître et ouvrir le plan de test à exécuter.
4. Eventuellement, vérifier sur les machines esclaves le fichier *jmeter.log* qui doit contenir  
***Jmeter.engine.RemoteJMeterEngineImpl: Starting backing engine***
5. Démarrer via le menu le plan de test sur une machine esclave ou toutes. (Voir entrées du menu)



# Limitations

---

Il y a quelques limitations connues concernant les tests distribués

1. RMI ne peut pas communiquer à travers des sous-réseaux différents
2. Comme JMeter envoie les résultats de test à la console maître. Il est facile de saturer le réseau. Utiliser plutôt un simple enregistreur de réponses pour visualiser les données plus tard
3. En général, 2 clients suffisent à surcharger le serveur cible



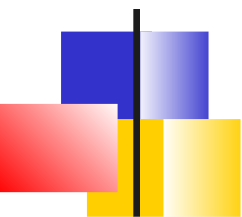
# Références

---

*Performance Testing Guidance for Web Applications*, Microsoft

*User Manual*, Jakarta Jmeter

*Apache Jmeter, A practical guide to automated testing and performance measurments for your website*, Packt Publishing



# Annexes





# Scripts additionnels Windows

---

*jmeter.bat, jmeterw.cmd* : JMeter dans le mode GUI (Windows)

*jmeter-n.cmd* : Exécute un test (fichier *.jmx*) dans le mode non-GUI

*jmeter-n-r.cmd* : Exécute un test (fichier *.jmx*) en distant

*jmeter-t.cmd* : Charge un test (fichier *.jmx*) dans le mode GUI

*jmeter-server.bat* : Démarre JMeter en mode serveur pour les tests distribués

*mirror-server.cmd* : Démarre le serveur Miroir JMeter en mode non-GUI

*shutdown.cmd* : Arrêt d'un client démarré en mode non GUI

*stoptest.cmd* : Arrêt brutal d'un client en mode non GUI



# Scripts additionnels Linux

---

*jmeter, jmeter.sh* :

- lance le mode GUI par défaut
- Non GUI avec certaines options

*jmeter-server* : Démarre JMeter en mode serveur

*mirror-server, mirror-server.sh.cmd* : Démarre le serveur Miroir JMeter en mode non-GUI

*shutdown.cmd* : Arrêt d'un client démarré en mode non GUI

*stoptest.cmd* : Arrêt brutal d'un client en mode non GUI

*heapdump.sh* : Demande de génération d'un heap dump sur un test en cours

```
JVM_ARGS="-Xms1024m -Xmx1024m"
```

```
jmeter -t test.jmx [etc.]
```



# Expressions régulières



# Introduction

---

JMeter inclut Apache Jakarta ORO pour la correspondance de motif (pattern matching)

Une expression régulière (regex ou regexp) est une séquence de caractères dont certains ont une signification symbolique et d'autres sont pris tel quel

L'expression permet d'identifier des motifs dans une donnée texte, ou de les traiter. Les comparaisons de motifs peuvent être de pure égalité ou des similarités très générales

L'expression est en fait une instruction dans un langage très concis. Pour être traitée, l'expression est fournie à un interpréteur.

Tutoriel : <http://www.regular-expressions.info/tutorialcnt.html>



# Caractères spéciaux

Ces caractères doivent être échappés par \ afin qu'il soit traité comme caractère ordinaire.

( ) : Groupement

[ ] : Classes de caractère

{ } : Répétition

\* + ? : Répétition

. : Caractère joker

\ : Caractère d'échappement

| : Alternative

^ \$ : Début ou fin d'une ligne ou d'une String

\b : Limite d'un mot



# Exemples

---

Balise HTML :

`<TAG\b[^>]*>(.*?)</TAG>`

Adresse email :

`\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b`

Date au format yyyy-mm-dd :

`^(19|20)\d\d[- /.](0[1-9]|1[012])[- /.]  
(0[1-9]|[12][0-9]|3[01])$`

Enlever les espaces :

`^[ \t]+|[ \t]+$`



# Syntaxe étendue

---

Jmeter utilise une syntaxe étendue en permettant plusieurs modificateur

- Les modificateurs simple-ligne (?s) et multi-lignes (?m) sont normalement placés au début de la regex.
- Le modificateur ignorer-casse (?i) peut s'appliquer sur une partie de la regex.



# Mode ligne

---

Le mode ligne unique a des effets sur le caractère spécial '.' dans ce mode ce caractère peut représenter le caractère « nouvelle ligne »

Le mode ligne multiples a des effets sur les caractères '^' et '\$' qui s'applique alors sur l'ensemble des lignes du texte





# Extracteur XPath



# Configuration XPath

---

**Utiliser Tidy** (tolerant parser) : Utilise Tidy pour parser la réponse HTML et la transformer en XHTML.

**Utiliser les espaces de nom** : Le parser utilise la résolution d'espace de noms

**Validation XML** : Valide le XML par rapport à un schéma (si Tidy n'est pas sélectionné)

**Ignorer les espaces** : (si Tidy n'est pas sélectionné)

**Récupérer les DTDs externes** : (si Tidy n'est pas sélectionné)

**Retourner le fragment** Xpath au lieu du contenu

**Nom de référence** : Le nom utilisé dans les noms de variables

**Requête XPath** : Peut retourner plusieurs correspondances

**Valeur par défaut** : A utiliser quand il n'y a pas de correspondance ou quand la valeur du nœud est vide

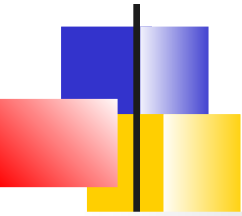


# Variables

---

Les variables suivantes sont positionnées :

- ***refName*** : La valeur de la première correspondance ou la valeur par défaut
- ***refName\_matchNr*** : Le nombre de correspondance
- ***refName\_n*** : La valeur de la nième correspondance



# HTTPS



# Implémentation

---

Il y a 2 implémentations disponibles pour les échantillons HTTP : Java et HttpClient 4.x (Apache HttpComponents)

L'implémentation Java a 2 limitations concernant HTTPS

- Pas possible d'utiliser un Proxy
- Ne supporte pas les certificats clients basés sur une configuration d'un keystore



# Contexte SSL

---

Par défaut, un contexte séparé est utilisé pour chaque thread.

Si l'on veut utiliser un contexte SSL unique :

```
https.sessioncontext.shared=true
```

Par défaut, le contexte SSL est préservé durant tout le test.

Si l'on veut réinitialiser le contexte SSL à chaque thread :

```
https.use.cached.ssl.context=false
```



# Niveau du protocole SSL

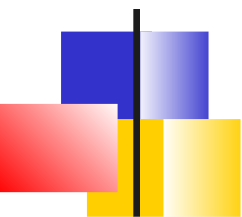
---

Par défaut, JMeter utilise le niveau TLS.

Si on veut utiliser un niveau différent :

`https.default.protocol=SSLv3`

La propriété *https.socket.protocols*  
permet d'autoriser des protocoles  
supplémentaires



# REST et SOAP





# Éléments REST

---

Pour tester une API REST, on peut par exemple utiliser les éléments suivants :

- Gestionnaires d'entêtes http avec  
Content-type=application/json
- Pour une requête POST/PUT/PATCH, utiliser le corps de la requête pour fournir les données d'entrée JSON
- Utiliser l'extracteur JSON
- Installer le plugin JSON Assertion

Attention, l'implémentation Java ne supporte pas les méthodes PATCH



# Éléments SOAP

---

L'échantillon SOAP a disparu de la version 3

Il existe un gabarit qui permet de créer un plan de test pour SOAP

Les éléments utilisés sont :

- Un gestionnaire d'entête  
Content-type=text/xml ; charset=utf-8  
Entêtes SOAP ...
- Une requête POST dont le corps est le message SOAP



# XPath

# Expression des chemins

La syntaxe basique de XPath est similaire à celle de système de fichiers et répertoires et utilise le caractère /

Si le chemin commence par / ; c'est un chemin absolu

**/AAA/CCC** sélectionne tous les éléments CCC enfant du nœud racine AAA

```
<AAA>
  <BBB/>
  <CCC/>
  <BBB/>
  <BBB/>
  <DDD>
    <BBB/>
  </DDD>
  <CCC/>
</AAA>
```

# Caractère //

Si le chemin commence par //, alors tous les éléments du document qui respectent l'expression sont sélectionnés.

**//DDD/BBB** sélectionne tous les éléments BBB enfants de DDD

```
<AAA>
  <BBB/>
  <CCC/>
  <BBB/>
  <DDD>
    <BBB/>
  </DDD>
  <CCC>
    <DDD>
      <BBB/>
      <BBB/>
    </DDD>
  </CCC>
</AAA>
```

# Caractère \*

Le caractère \* peut être utilisé comme joker

**/AAA/CCC/DDD/\*** sélectionne tous les éléments derrière la suite /AAA/CCC/DDD

```
<AAA>
  <CCC>
    <DDD>
      <BBB/>
      <BBB/>
      <EEE/>
      <FFF/>
    </DDD>
  </CCC>
  <CCC>
    <BBB>
      <BBB>
        <BBB/>
      </BBB>
    </BBB>
  </CCC>
</AAA>
```

# Indice

Le *i*ème élément peut être indiqué avec les crochets []

**/AAA/BBB[1]** sélectionne le premier élément BBB enfant de l'élément racine AAA

<AAA>

<BBB/>

<BBB/>

<BBB/>

<BBB/>

</AAA>

# Fonction last()

La fonction ***last()*** sélectionne le dernier élément.

***/AAA/BBB[last()]*** sélectionne le dernier élément  
BBB enfant de l'élément racine AAA

**<AAA>**

**<BBB/>**

**<BBB/>**

**<BBB/>**

***<BBB/>***

**</AAA>**



# Attributs

Les attributs sont spécifiés par le préfixe @.

**//@id** sélectionne tous les attributs @id

```
<AAA>
  <BBB id = "b1"/>
  <BBB id = "b2"/>
  <BBB name = "bbb"/>
  <BBB/>
</AAA>
```

**//BBB[@id]** sélectionne tous les éléments BBB qui ont l'attribut id

```
<AAA>
  <BBB id = "b1"/>
  <BBB id = "b2"/>
  <BBB name = "bbb"/>
  <BBB/>
</AAA>
```

# Critère de sélection



---

Les valeurs des attributs peuvent être utilisé comme critère de sélection.

**//BBB[@name='bbb']** Sélectionne les éléments BBB dont l'attribut à la valeur 'bbb'

<AAA>

<BBB id = "b1"/>

**<BBB name = " bbb "/>**

**<BBB name = "bbb"/>**

</AAA>

# Fonction *count()*

La fonction *count()* compte le nombre d'éléments sélectionnés.

***/\*[count(BBB)=2]*** Sélectionne les éléments qui ont deux enfants BBB

```
<AAA>
  <CCC>
    <BBB/>
    <BBB/>
    <BBB/>
  </CCC>
  <DDD>
    <BBB/>
    <BBB/>
  </DDD>
  <EEE>
    <CCC/>
    <DDD/>
  </EEE>
</AAA>
```



# *JSONPath*

Une expression *JSONPath* fait référence à une structure JSON à la façon de XPath pour XML

- Comme JSON n'a pas spécifiquement d'élément racine, le signe **\$** fait référence au niveau le plus haut de la structure de donnée
- Les expressions peuvent utiliser la notation **.** Ou **[]**  
`$.store.book[0].title`  
`$['store']['book'][0]['title']`
- JSONPath supporte **\*** comme caractère joker
- Le caractère **@** représente l'élément courant
- Expression scriptée  
`$.store.book[(@.length-1)].title`
- Le caractère **?** permet de filtrer  
`$.store.book[?(@.price < 10)].title`



# Récepteurs 2.x



# Graphique de résultats

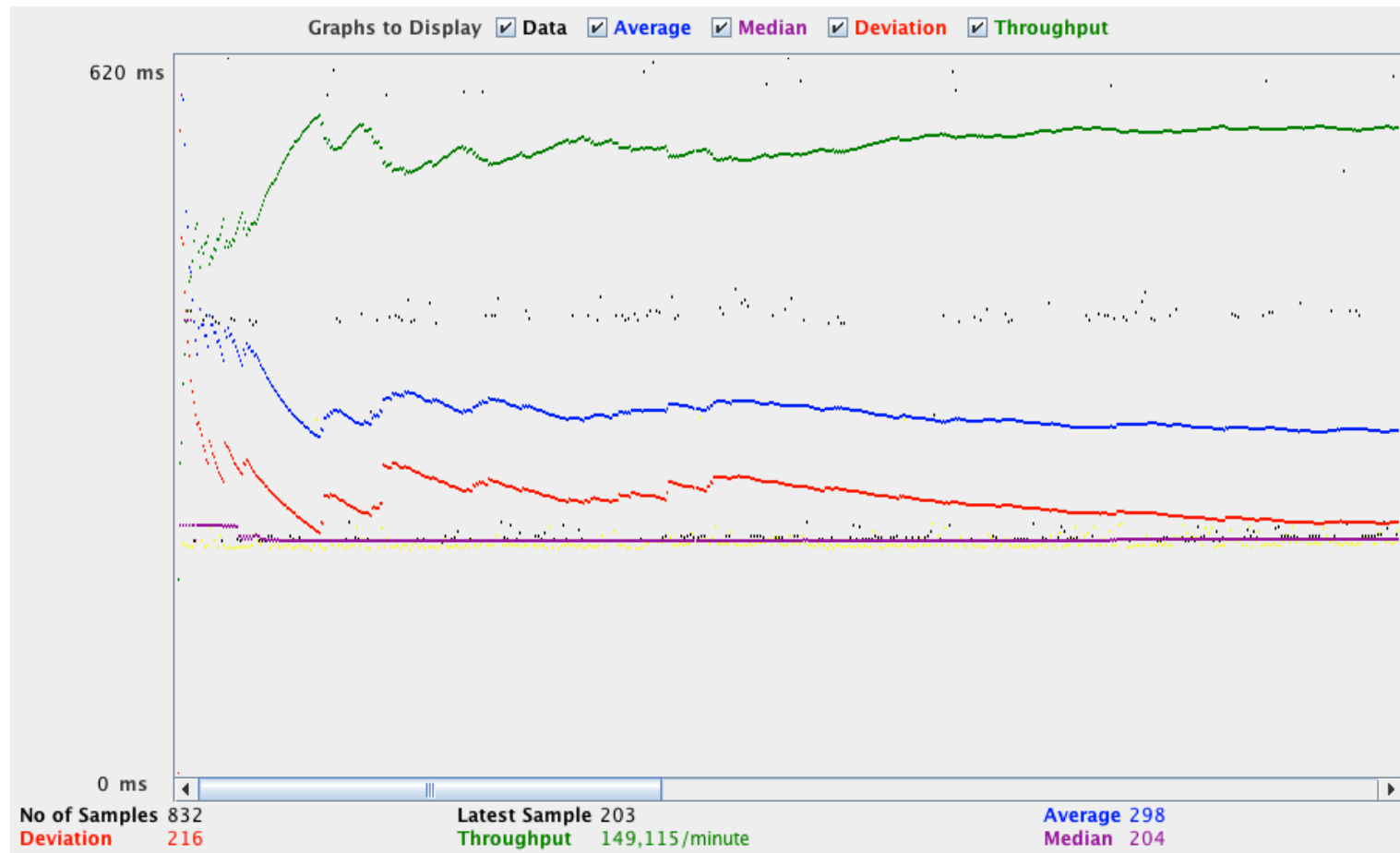
---

Le graphique de résultat génère un graphe qui affiche tous les temps des échantillons.

En bas du graphique, les valeurs courantes sont indiquées

L'intérêt de ce graphique est de voir les variations de ces valeurs durant l'exécution du test

# Example





# Moniteur de courbes (spline)

## *Déprécié*

---

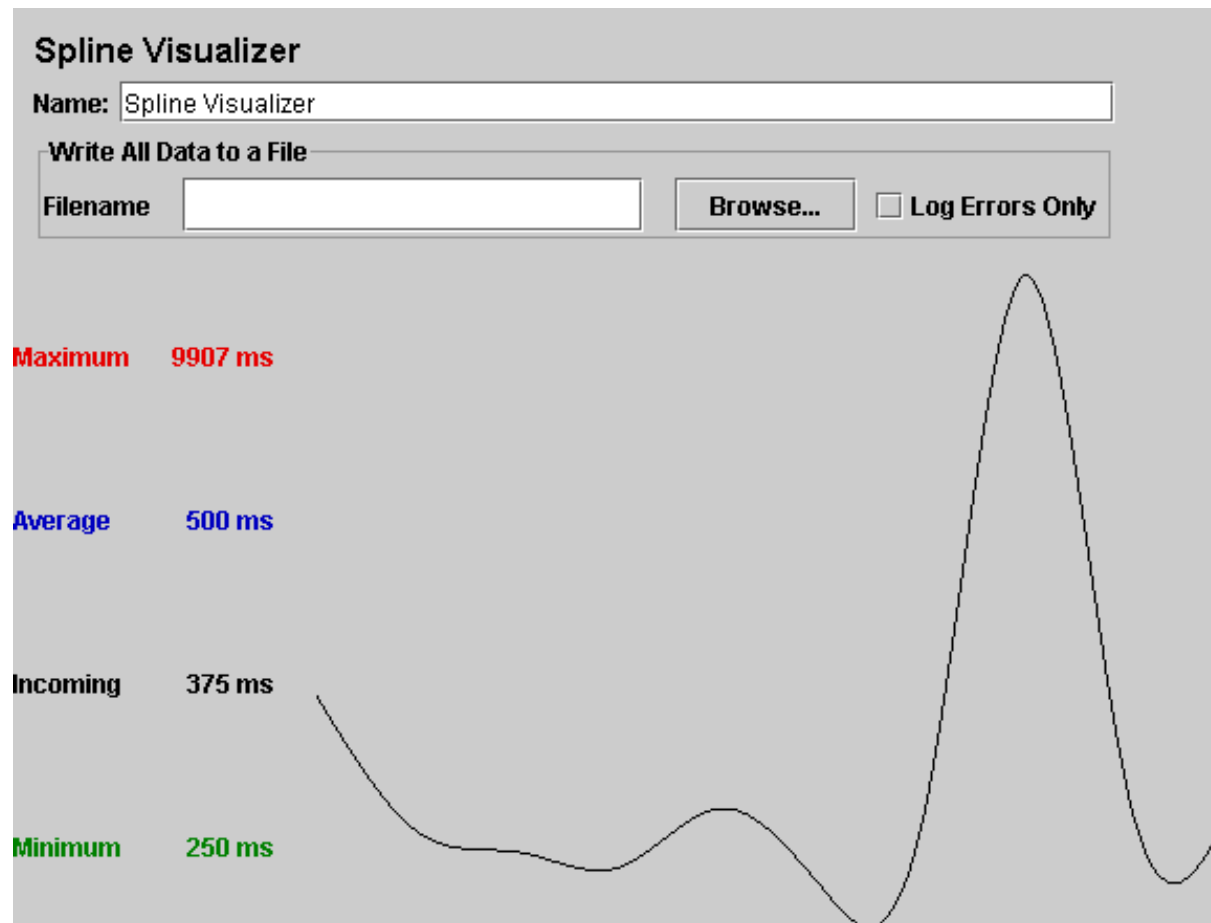
Le moniteur de courbes fournit une vue de l'ensemble des temps d'échantillonnage à partir du début de l'essai jusqu'à la fin quelque soit le nombre d'échantillons.

La courbe contient 10 points chacun représentant 10 % des échantillons et les 10 points sont connectés selon une logique de spline

Permet de visualiser le modèle de distribution des données



# Example





# Rapport agrégé

---

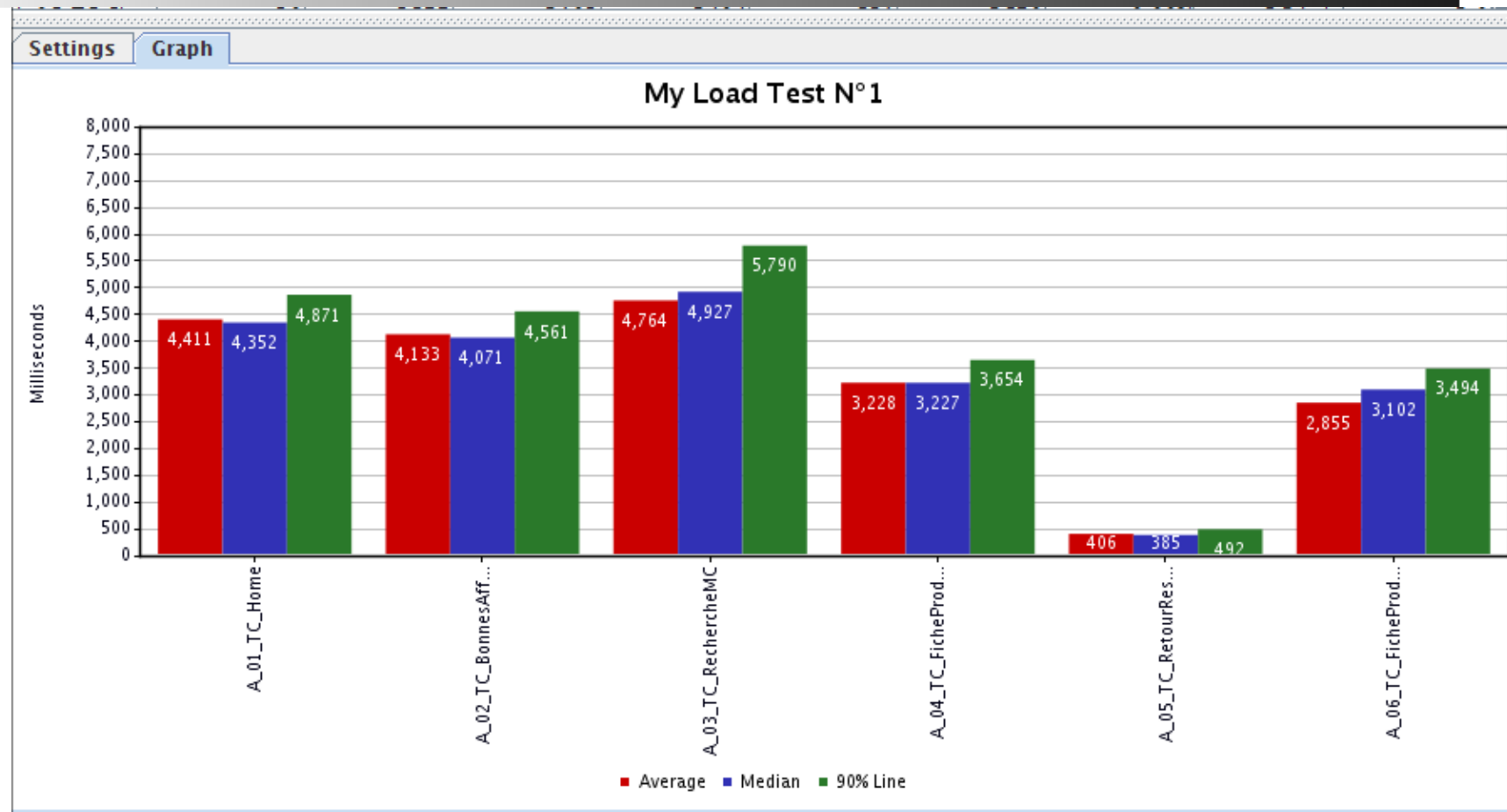
Le rapport agrégé présente un tableau de données. Il crée une ligne par échantillonneur ayant un nom différent.

Pour chaque requête, il affiche le nombre de requêtes, le min, max, moyenne, taux d'erreur, débit et koctets/seconde.

Il n'est pas trop consommateur de mémoire et peut être présent lors d'un test de charge.

Une version en mode graphique est disponible (graphique agrégé)

# Graphique agrégé



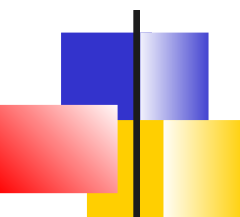


# Graphique évolution temps de réponse

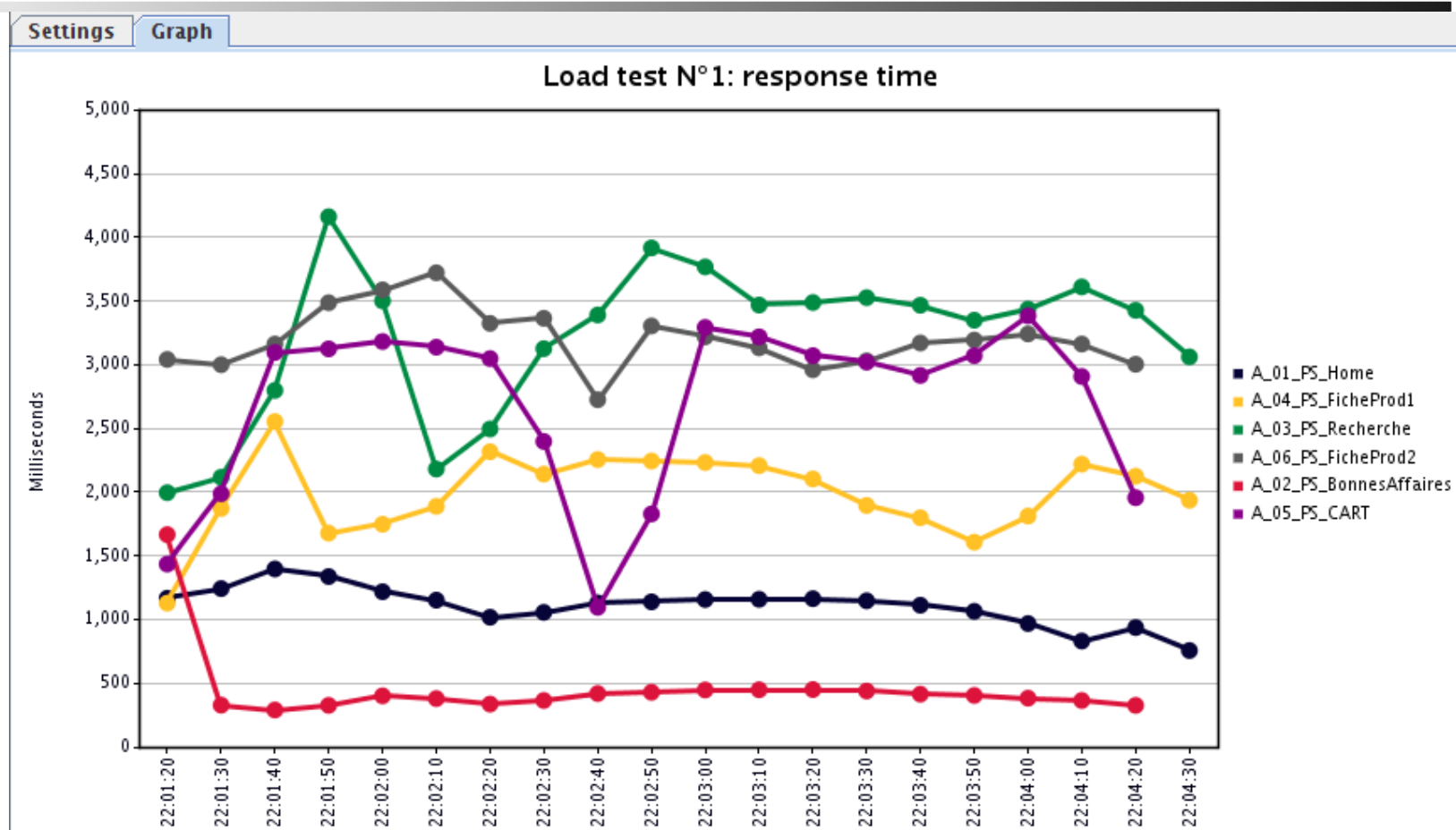
---

Le graphique « évolution temps de réponse » dessine l'évolution des temps de réponse durant le test pour chaque requête.

Si plusieurs échantillons existent pour le même timestamp, la valeur moyenne est utilisée.



# Evolution temps de réponse





# Graphique de distribution

## *Déprécié*

---

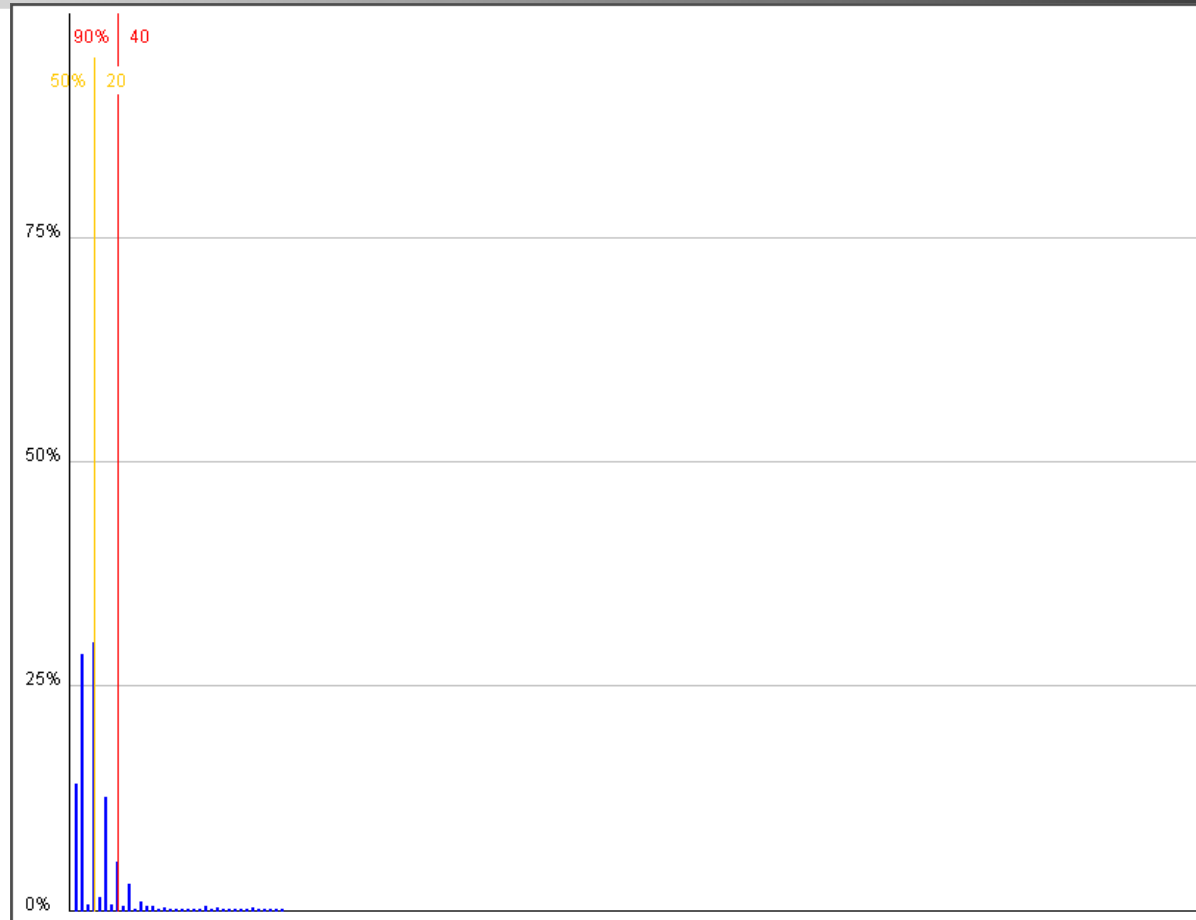
Le graphique de distribution affiche une barre pour chaque temps unique de temps de réponse (la granularité étant de 10 millisecondes)

Le graphique dessine 2 lignes seuil : 50% et 90% qui signifient que 50%/90 % des temps de réponses se situent entre 0 et ces lignes de seuils

Une application performante produit des graphiques où les barres sont très proches.

Une application ayant des problèmes de performances peut générer de grosses dispersions

# Exemple





# Moniteur de connecteurs

## *Déprécié*

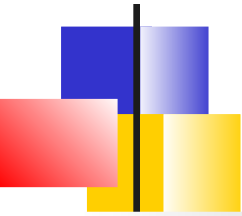
---

Le moniteur de connecteur est un récepteur affichant le statut du serveur. Il est compatible avec Tomcat 5 mais le servlet status présent dans Tomcat peut être porté facilement sur n'importe quel conteneur de servlet

Le récepteur présente 2 onglets :

- L'onglet « **Santé** » : affiche le statut de 1 ou plusieurs serveurs en fonction du nombre de threads actives sur le serveur et ses capacités max.
- L'onglet « **Performance** » affiche le performance d'un serveur (CPU, Mémoire, nombre de threads) lors des derniers mille échantillons





# *BeanShell*



# Introduction

---

Chaque élément BeanShell a sa propre copie de l'interpréteur

Si l'élément est répété ou appelé à l'intérieur d'une boucle alors l'interpréteur est conservé entre les invocations à moins que l'option « **Réinitialiser l'interpréteur avant chaque appel** » soit sélectionné. Si l'interpréteur utilise trop de mémoire, on peut cocher cette option

Les scripts peuvent être testés en dehors de JMeter via la ligne de commande

```
$ java -cp bsh-xxx.jar[;other jars as needed] bsh.Interpreter file.bsh  
[parameters]
```

ou

```
$ java -cp bsh-xxx.jar bsh.Interpreter  
bsh% source("file.bsh");  
bsh% exit(); // or use EOF key (e.g. ^Z or ^D)
```



# Partage de variables

Des variables peuvent être définies dans des scripts de démarrage. Elles seront conservées pendant toutes invocations sauf si l'option de réinitialisation est cochée

Les scripts peuvent accéder aux variables JMeter en utilisant les méthodes *get()* et *put()* de la variable "vars" ou *getObject()* et *putObject()* pour des objets autre que des String

Une autre méthode pour partager des variables entre scripts consiste à utiliser l'espace de nom partagé "*bsh.shared*". Par exemple :

```
if (bsh.shared.myObj == void){  
    // not yet defined, so create it:  
    myObj=new AnyObject();  
}  
bsh.shared.myObj.process();
```