

Développement de fonctionnalités de l'outil GREENSPECTOR

Rapport de stage - Solène MOREAU



4 octobre 2016

Tuteur école : Jean-Yves MARTIN - Tuteur entreprise : Thierry LEBoucQ



Centrale
Nantes



GREENSPECTOR

1. Remerciements

Je souhaiterais tout d'abord remercier l'ensemble de l'équipe pédagogique de l'option informatique de Centrale Nantes, menée par Jean-Yves MARTIN, pour avoir réussi à transmettre leur passion et leurs compétences pendant leurs cours.

Je souhaiterais également remercier tout particulièrement Thierry LEBOUQC pour avoir pu rendre possible ce stage à la suite d'une discussion au détour d'un forum «Informatique Durable».

Enfin, un grand merci à l'ensemble de l'équipe GREENSPECTOR pour m'avoir accueillie : Olivier, Thomas et Thomas, Abdou, Maxime, Guillaume, Michaël, Gwenaëlle, Yann, Bertrand, Kim, Héloïse, Pierre-Alexandre, Bilong... Même si la liste est loin d'être exhaustive, de nombreux moments ont fait de ce stage une expérience agréable et enrichissante :

- l'entraide, la disponibilité et l'expertise de chacun ;
- les sorties course à pied du midi ;
- la bonne humeur toujours au rendez-vous ;
- les blagues du lundi matin, du vendredi (et puis, aussi des autres jours).

Table des matières

1. Remerciements	2
2. Introduction	6
3. Présentation de l'entreprise	8
4. Cadre général	9
4.1. Écoconception logicielle	9
4.2. Description de l'outil GREENSPECTOR	10
4.2.1. Architecture	10
4.2.2. Intelligence	11
4.3. Description des projets	13
4.3.1. Objectifs, évolutions de la roadmap	13
4.3.2. Travail demandé pendant ce stage	14
5. Mise en place d'un environnement de test pour des règles PHP	16
6. Évolution de l'outil Meter	26
6.1. Intégration de nouvelles sondes dans l' <i>API Meter Java</i>	27
6.2. Mise à jour du plugin pour JMeter	31
6.3. Création d'un module indépendant	35
7. Évolution du tableau de bord utilisateur : implémentation de règles dynamiques sur les ressources	39
8. Participation à la mise en place d'une procédure de suivi qualité	45
9. Tâches transverses liées au développement et à l'édition de logiciels	47
9.1. Méthodes de travail	47
9.2. Liens avec les clients	48
9.3. Documentation utilisateur et documentation interne	48
9.4. Déploiement et intégration continue	48
10. Conclusion	50
10.1. Bilan du travail réalisé	50
10.2. Difficultés rencontrées	50
10.3. Perspectives	50
10.4. Bilan personnel	50
A. Description de la règle «Préférer les regex PCRE aux regex POSIX»	52
A.1. Problème	52
A.2. Description	52
A.3. Solution	52
A.4. Référence	52
B. Description d'une règle dynamique du domaine ressources : «Minimiser l'impact en CPU plateforme sur mobile pendant l'étape idle»	53
B.1. Problème	53

B.2. Description	53
B.3. Solution	53
C. Utilisation de l'<i>API Meter Java</i> dans un test Selenium	54
D. Tableau de bord : exemple d'analyse du projet PatriMath	56

Table des figures

1.	Architecture et flux	10
2.	Référentiel de règles sur l'interface web	13
3.	JMeter : groupe d'unités	18
4.	JMeter : requête HTTP	18
5.	Mesure d'un processus (<i>firefox</i>) avec 3 modules de PowerAPI : en abscisse, le temps en ms, en ordonnée, la puissance en mW	20
6.	Mesure d'une machine virtuelle avec le module <i>sigar-cpu-simple</i> de PowerAPI et la sonde DCScope pour Unix : en abscisse, le temps en ms, en ordonnée, le taux d'utilisation CPU en %	21
7.	Exemple de code <i>vert</i> pour la règle : Utiliser un chemin absolu pour un import de fichier	22
8.	Exemple de code <i>gris</i> pour la règle : Utiliser un chemin absolu pour un import de fichier	22
9.	Exemple de code <i>gris</i> pour la règle : Utiliser un chemin absolu pour un import de fichier	22
10.	Fonctionnement général des modules de l'outil <i>Meter</i>	26
11.	Diagramme de séquence de l'interface <i>Probe</i>	27
12.	Exemple de mesure du processus <i>firefox</i> sur PC avec la sonde PowerAPI	29
13.	Diagramme de classes du plugin pour JMeter	32
14.	Diagramme de séquences du plugin pour JMeter	34
15.	Utilisation du plugin GREENSPECTOR pour JMeter	34
16.	Résultats sur l'interface web de mesures utilisant le plugin pour JMeter	35
17.	Message d'aide au lancement du module indépendant <i>Meter</i>	38
18.	Ancien tableau de bord	39
19.	Nouveau tableau de bord	40
20.	Exemple de test du plugin pour JMeter	46
21.	Vision systémique du processus de développement d'un logiciel (source : [5])	47
22.	Goulot d'étranglement (source : [5])	47
23.	Tableau de bord - vue 1	56
24.	Tableau de bord - vue 2	56
25.	Tableau de bord - vue 3	57

2. Introduction

Les technologies de l'information et de la communication émettent autant de gaz à effet de serre que le trafic aérien international. En 2030, les fermes de serveurs et les infrastructures de télécommunication consommeront autant d'électricité que l'humanité en 2008.

C'est ce que rapporte une étude menée en 2008 à l'université de Dresde [9], et confirme le constat suivant : le numérique ne cesse de grossir.

Les services et contenus sont à la fois de plus en plus nombreux, mais aussi de plus en plus volumineux. Ces nouvelles technologies ont permis, et permettent encore, de nombreuses avancées dans la gestion des activités et des échanges au sein de la sphère professionnelle comme personnelle : amélioration des échanges, partage d'information facilité... Cependant, les bénéfices environnementaux espérés (diminution des déplacements, de la consommation de papier) sont à nuancer, comme le souligne notamment l'Agence de l'Environnement et de la Maîtrise de l'Énergie [6]. En effet, l'analyse du cycle de vie de ces nouvelles pratiques montre que la consommation de ressources liée aux équipements impliqués reste toujours croissante.

Face à ce constat, il faut mettre en opposition :

- la raréfaction des énergies fossiles et des métaux rares, actuellement indispensables pour fabriquer et faire fonctionner les technologies de l'information et de la communication ;
- le réchauffement climatique, lié notamment à l'augmentation des émissions de gaz à effet de serre, émissions provoquées entre autres par la fabrication et l'utilisation des outils du numérique.

Dans le domaine informatique, un des axes d'action consiste à optimiser les logiciels dès leur conception : c'est l'écoconception logicielle. Selon le Benchmark Green IT 2016 [7], cette pratique possède un fort potentiel en ce qui concerne le développement d'un numérique plus vertueux et responsable : «de l'ordre de 2 à 100 fois moins de ressources informatiques nécessaires, à tous les niveaux du système d'information». La tendance actuelle n'allant pas vers une modération et une sobriété de l'utilisation du numérique, ce levier d'action apparaît comme primordial.

GREENSPECTOR est la première solution outillée pour l'écoconception des logiciels. Intégrée dans les outils du développeur, GREENSPECTOR permet aux équipes de direction des systèmes d'information de réduire la consommation de ressources des logiciels qu'elles produisent ou qu'elles réceptionnent. Les gains qui en découlent permettent d'augmenter l'autonomie des appareils mobiles et des objets connectés, de dégager des économies dans les datacenters ou d'améliorer la performance générale des applications.

Les sujets abordés pendant ce stage sont les suivants :

- participer à la conception et à l'architecture de l'outil GREENSPECTOR ;
- développer les fonctionnalités back-end de l'outil ;
- mesurer et développer des règles de bonnes pratiques de développement ;
- mettre en place des tests unitaires des développements ;
- participer à la mise en production continue des développements.

La démarche générale de ce stage a été une intégration progressive dans le quotidien de l'équipe :

1. mesure des règles de bonnes pratiques de développement ;

2. prise en main de l'outil GREENSPECTOR et ajout d'une petite fonctionnalité ;
3. participation au développement et à la spécification d'une évolution majeure de l'outil.

3. Présentation de l'entreprise

GREENSPECTOR est une jeune entreprise créée en novembre 2010 à Nantes, alors sous le nom de KaliTerre, par Thierry Leboucq et Martin Dargent. Leur objectif était de développer des solutions pour appliquer les principes du développement durable au monde du numérique. L'entreprise se spécialise dans les missions de conseil et formation en RSE et Green IT ¹ auprès de l'ADEME, Nantes Métropole, CGI France, le Ministère de la Culture et de la Communication...

En 2011, Martin Dargent quitte KaliTerre pour prendre la direction d'une autre start-up, EasyVirt, spécialisée dans l'optimisation de la virtualisation dans les datacenters. Il reste par la suite actionnaire et partenaire de l'entreprise.

En 2013, deux nouveaux associés rejoignent l'entreprise comme co-gérants : Thomas Corvaisier et Olivier Philippot.

Entre 2010 et 2014, KaliTerre finance sur fonds propres ses projets de R&D sur l'éco-conception en participant à différents projets de recherche nationaux et européens sur l'efficacité logicielle. À la suite de ces projets de R&D, KaliTerre crée une suite logicielle, GREENSPECTOR, permettant de :

- mesurer la consommation énergie-ressource des applications ;
- détecter les objets surconsommateurs dans le code source des logiciels ;
- proposer des corrections aux développeurs.

Soutenue par le Fonds d'investissement breton Nestadio Capital à hauteur de 300 000 €, KaliTerre, devenue GREENSPECTOR, amorce un virage stratégique en 2016. Elle lance la commercialisation de sa solution d'écoconception des logiciels, s'adressant en priorité aux entreprises de services du numérique, aux acteurs de l'industrie du mobile et de l'objet connecté, ainsi qu'aux grands comptes qui achètent et produisent du logiciel.

L'entreprise souhaite à présent renforcer ses effectifs et dépasser le million d'euros de chiffre d'affaire d'ici 2 ans, afin de se développer rapidement sur le marché international à l'horizon 2018.

GREENSPECTOR a été la première solution logicielle labellisée par le European Code of Conduct for Datacenters, et est partenaire du projet WebEnergyArchive, qui est une première «étiquette énergétique» des sites web. L'entreprise a également fondé la communauté française des logiciels écoconçus (le Green Code Lab), auteure du livre Green Patterns [8] qui est un manuel d'écoconception logicielle à destination des développeurs.

1. Selon Wikipédia [3], le Green IT (informatique durable) comme un concept qui vise à réduire l'empreinte écologique, économique et sociale des technologies de l'information et de la communication.

4. Cadre général

4.1. Écoconception logicielle

L'enjeu de l'écoconception est d'intégrer les enjeux environnementaux dès la phase de conception des produits en se basant sur la prise en compte de l'ensemble des étapes du cycle de vie des produits.

Même s'ils sont considérés comme immatériels, les logiciels ont besoin d'équipements informatiques pour fonctionner (serveurs, ordinateurs, écrans, mobiles, imprimantes, etc). Or l'impact environnemental de ces équipements ne cesse de grossir, et leur empreinte écologique se concentre aux étapes de fabrication et de fin de vie. Pour rendre le logiciel plus respectueux de l'environnement, il est donc primordial d'allonger la durée d'utilisation des équipements informatiques. Or la couche logicielle est un facteur d'obsolescence non négligeable, car les ressources matérielles nécessaires (quantité de mémoire vive, puissance du processeur, espace disque) ne cessent d'augmenter. Par exemple, chaque nouvelle version du binôme Windows - Office requiert le double de ressources par rapport à la précédente version : il faut 70 fois plus de mémoire vive pour écrire un simple texte avec Windows 7 - Office 2010 qu'avec Windows 98 - Office 97 [8] !

L'écoconception appliquée aux logiciels consiste donc à optimiser les logiciels dès leur conception pour :

- diminuer la consommation énergétique du logiciel pendant sa phase d'utilisation (et donc les émissions de gaz à effet de serre liées) ;
- améliorer la performance des applications et sites web, l'autonomie des appareils mobiles pour une meilleure expérience utilisateur ;
- allonger la durée de vie des équipements informatiques (et donc diminuer leur impact environnemental).

L'écoconception logicielle consiste à appliquer des bonnes pratiques pour diminuer la consommation en ressources matérielles tout en maintenant la performance et en répondant au besoin de l'utilisateur, par exemple :

- améliorer la méthodologie pour optimiser la coordination, la productivité ;
- optimiser le code source pour réduire les ressources nécessaires ;
- optimiser les fonctionnalités ;
- travailler sur l'architecture ;
- mesurer précisément la consommation des applications et sites web ;
- évaluer et comparer les librairies ;
- augmenter la scalabilité des applications et des matériels pour augmenter leur cycle de vie.

Quelques exemples...

Un audit d'une application Android (réalisé par GREENSPECTOR) a conduit à des corrections dans le code source de l'application résultant à un gain de 69% sur la consommation d'énergie de l'application, augmentant ainsi l'autonomie du mobile de 8h.

L'utilisation de GREENSPECTOR pendant le développement d'une application de gestion de congés pour 80 000 salariés a permis d'améliorer la performance et l'expérience utilisateur en divisant le temps de réponse par 3.

4.2. Description de l'outil GREENSPECTOR

L'objectif de GREENSPECTOR est de permettre aux développeurs en informatique d'utiliser un outil facile d'utilisation, pertinent et s'intégrant dans leurs environnements de développement pour les aider à coder des applications et logiciels plus optimisés.

Sauf mention contraire, la description de l'outil faite dans cette partie concerne la version 1.7.0 sortie en septembre 2016.

4.2.1. Architecture

L'outil GREENSPECTOR permet d'analyser une application de 2 manières différentes :

- par une analyse statique qui permet de détecter des violations de règles de bonnes pratiques de développement au sein du code source, c'est la fonctionnalité *Scan* ;
- par une analyse dynamique qui permet de mesurer l'application pendant son fonctionnement (utilisation de sondes pour mesurer l'énergie, la mémoire, le trafic réseau, etc), c'est la fonctionnalité *Meter*.

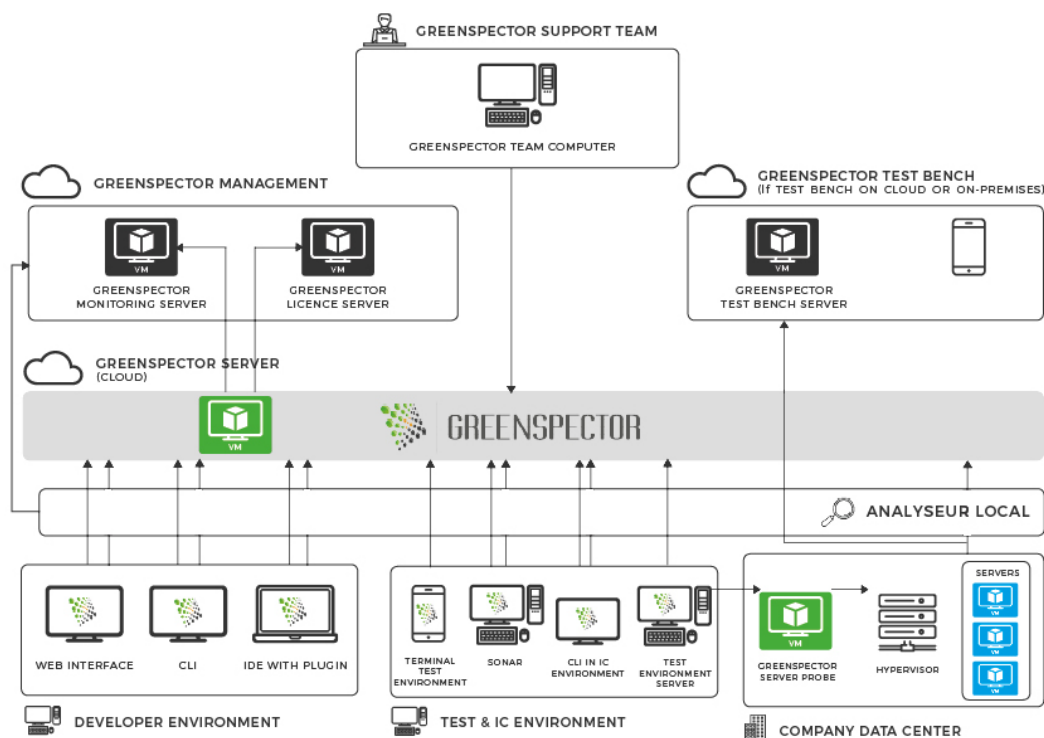


Figure 1 – Architecture et flux

La figure 1 présente un schéma de l'architecture et des flux de données.

Le point central de cette architecture est le serveur GREENSPECTOR, qui possède l'intelligence de l'outil. Ce serveur central est relié à un serveur de supervision et un serveur de licence, qui servent respectivement à vérifier le fonctionnement de l'outil et à vérifier que la licence en cours est valide.

Afin d'aider les utilisateurs à analyser de façon statique et dynamique leurs applications, l'outil GREENSPECTOR s'intègre à la fois dans l'environnement du développeur, et à la fois dans les environnements de test et d'intégration continue.

Sur son poste, le développeur a plusieurs outils à sa disposition :

- une interface web qui lui permet de visualiser les résultats de ses analyses de code et mesures dynamiques et de lancer des analyses de code ;
- des modules (ou plugins) greffés dans des environnements de développement intégrés (Eclipse, IntelliJ...) lui permettant d'analyser du code source pendant la phase de développement.

Pour les environnements de test et d'intégration continue, d'autres outils peuvent être utilisés :

- des modules ou des API accompagnées de sondes (sonde Android et sonde serveur) pour effectuer des mesures dynamiques en s'intégrant dans des outils de tests tels que Selenium, JMeter, UIAutomator ;
- des modules greffés dans des outils d'intégration continue (Jenkins, SonarQube...) pour analyser du code source et effectuer des mesures dynamiques de façon automatisée.

Que ce soit dans son propre environnement de développement, ou dans des environnements mutualisés de test et d'intégration continue, l'utilisateur a à sa disposition des modules indépendants utilisables en ligne de commande qui lui permettent de lancer des mesures dynamiques, des analyses de code, d'exécuter des tests standard automatisés, et d'envoyer les résultats au serveur GREENSPECTOR. L'utilisation en ligne de commande permet de s'intégrer facilement dans des tâches automatisées et continues.

Tous les outils présentés précédemment communiquent avec le serveur central GREENSPECTOR (envoi des résultats d'analyse statique, envoi des mesures dynamiques...).

4.2.2. Intelligence

Référentiel de règles et de mesures

GREENSPECTOR utilise un référentiel de règles organisé en 3 domaines principaux :

- le domaine code pour l'analyse statique du code source des logiciels et applications ;
- le domaine réseau pour l'analyse dynamique des requêtes et de leur contenu ;
- le domaine ressources pour l'analyse de la mesure dynamique de la consommation de ressources (énergie, CPU, ...) sur une plateforme côté client (PC, mobile) ou côté serveur.

Ce référentiel de règles est construit en s'appuyant sur une démarche continue de recherche, de développement et de veille sur les bonnes pratiques d'écoconception logicielle.

Des indicateurs sont associés aux règles d'analyse statique du code source, issus de mesures comparant du code *vert* (respectant la règle) et du code *gris* (enfreignant la règle) :

- le gain en énergie ;
- le gain en mémoire ;
- le gain en performance ;
- la temps de correction.

Ces indicateurs sont notés entre 1 et 4 : 1 correspond à la criticité la plus faible, c'est-à-dire à un faible gain ou à un temps de correction élevé.

Moteur d'analyse

L'analyse statique et l'analyse dynamique d'une application permettent de distinguer les règles respectées des règles violées.

Moteur d'analyse statique Il sert à analyser le code source d'une application en fonction d'un référentiel de règles statiques : il détecte l'ensemble des éléments dans le code qui ne respectent pas ces règles et permet donc de lister le nombre de violations pour chaque règle ainsi que l'endroit où elles apparaissent dans le code.

En se basant sur la liste des violations et les indicateurs associés à chaque règle statique, ces mêmes indicateurs sont extrapolés à l'application analysée. Ce calcul prend en compte les gains unitaires mesurés pour chacune des règles, ainsi que le nombre de violations par ligne de code. En effet, l'impact de 10 violations sur un projet de 100 lignes de code n'est pas le même que sur un projet de 1000 lignes de code.

Moteur d'analyse dynamique Il sert à analyser des mesures dynamiques effectuées pour une application en fonction d'un référentiel de règles dynamiques.

Les éléments qui composent une règle dynamique sont :

- une métrique (par exemple, le nombre de requêtes HTTP envoyées entre le client et le serveur) ;
- un ensemble de seuils permettant d'attribuer un écoscore à la règle (par exemple, 75% du score maximal s'il y a entre 5 et 10 requêtes, 100% du score maximal s'il y a moins de 5 requêtes, etc).

Le moteur d'analyse dynamique consiste donc en l'implémentation de l'analyse correspondante à chaque règle dynamique.

Interface utilisateur

Le tableau de bord présent sur l'interface web de l'outil GREENSPECTOR intègre tous les constats de l'ensemble des analyses possibles d'une application. Cette section décrit l'intelligence de l'outil en se basant sur une explication de la vue utilisateur de l'outil.

Les figures référencées sont présentes dans l'annexe D.

La figure 23 illustre une première vue du tableau de bord. L'écoscore global indique le niveau d'écoconception d'une application : il est noté entre 0 et 100 (le meilleur score étant 100) et est une moyenne des écoscores de chaque domaine (code, réseau, ressources). Pour évaluer correctement le niveau d'écoconception d'une application, il est ainsi nécessaire d'analyser l'ensemble de ces domaines.

L'écoscore de chaque domaine est calculé en faisant la somme des points obtenus pour chacune des règles liées à ce domaine. Il est donc fonction des gains possibles.

La section «IMPACT DES RESSOURCES» permet de visualiser l'impact de l'application sur mobile, c'est-à-dire de combien l'autonomie de la batterie diminue si l'application tourne en boucle sur le mobile. L'autonomie de référence est déduite à partir de la capacité de la batterie.

La section «RÈGLES», illustrée par la figure 24, liste l'ensemble des règles analysées en indiquant :

- le score obtenu pour cette règle (égal au score maximal si la règle est respectée) ;

- la priorité de correction basée sur le gain potentiel de la règle (c'est-à-dire le nombre de points pour arriver au score maximal) ;
- un petit texte détaillant le résultat de l'analyse (par exemple ici, 6 fichiers sans entête de cache) ;
- éventuellement, une liste des éléments qui violent la règle.

La section «TESTS DETAILS» dont un exemple est fourni sur la figure 25 indique les ressources consommées par l'application pendant les différentes étapes des tests effectués. Ces tests sont standardisés et fournis par l'outil GREENSPECTOR. Par exemple, pour le test d'une page web, les étapes des tests standardisés sont : chargement de la page, inactivité, défilement vers le bas de la page. Ces tests sont utilisés par les règles du domaine ressources.

Cette interface utilisateur permet également de suivre l'évolution d'une application au fur et à mesure des versions analysées. Pour chaque valeur affichée sur le tableau de bord (ressources consommées, écoscores), l'évolution entre la valeur actuelle et la valeur de la précédente version est indiquée.

La figure 2 présente le référentiel de règles tel qu'il est affiché sur l'interface web. Les domaines d'analyse sont indiqués par des onglets en haut. Pour le domaine code, une barre de sélection permet de trier les règles par langage.



Figure 2 – Référentiel de règles sur l'interface web

4.3. Description des projets

4.3.1. Objectifs, évolutions de la roadmap

Pendant mon stage, j'ai eu l'occasion de participer à une séance de réflexion (*brainstorming*) sur la feuille de route de GREENSPECTOR, ce qui m'a permis de mieux appréhender l'ensemble des fonctionnalités de l'outil. Cette feuille de route est basée sur 3 mots-clef : pertinence, intégration et facilité d'usage.

L'outil doit être pertinent pour apporter une réelle plus-value à l'utilisateur et lui proposer des corrections qui auront un réel impact sur la consommation d'énergie et la performance de son application. L'outil doit s'intégrer facilement dans les environnements de développement, de tests et d'intégration continue des clients. Enfin, l'utilisation de GREENSPECTOR doit être simple et intuitive pour inciter l'utilisateur à analyser et mesurer ses applications.

Le résultat d'une séance de réflexion sur la feuille de route consiste en la liste des évolutions et tâches à effectuer dans les 6 prochains mois pour améliorer l'outil.

4.3.2. Travail demandé pendant ce stage

Mise en place d'un environnement de test pour des règles PHP

L'objectif est ici de répondre au critère de pertinence. Afin que les corrections proposées aux développeurs pour améliorer leur code soient pertinentes, il est nécessaire de mesurer le gain apporté par les règles de bonnes pratiques de développement. Ces mesures permettent d'identifier les règles les plus intéressantes et d'ajuster les calculs de gains fournis à l'utilisateur.

Dans ce cadre, il m'a été demandé de travailler plus particulièrement sur les règles pour le langage PHP.

Évolution de l'outil *Meter*

L'objectif principal est ici le critère d'intégration. En effet, il m'a été demandé d'améliorer l'intégration de l'outil GREENSPECTOR avec des sondes de mesure et des outils externes.

Évolution du tableau de bord utilisateur : implémentation de règles dynamiques sur les ressources

Pour cette partie, le but est la facilité d'usage de GREENSPECTOR en fournissant à l'utilisateur un tableau de bord qui synthétise l'ensemble des mesures et analyses effectuées sur une application. J'ai participé au développement de cette fonctionnalité sur la partie concernant l'implémentation de règles permettant d'analyser la consommation de ressources d'une application (mesurée dynamiquement).

Participation à la mise en place d'une procédure de suivi qualité

Dans un contexte d'amélioration continue de l'outil GREENSPECTOR, la procédure de suivi qualité et de tests a évolué tout au long de mon stage. Avant la livraison de la version majeure 1.7.0 en septembre 2016, j'ai participé à la rédaction et à l'exécution de tests fonctionnels permettant de tester à la fois les nouvelles fonctionnalités et la non-régression de l'outil.

Tâches transverses liées au développement et à l'édition de logiciels

Enfin, il m'a été demandé pendant ce stage de m'intégrer à des tâches et des méthodes de travail plus transverses (méthodes agiles de développement, déploiement et intégration continue, contacts avec les clients, etc).

5. Mise en place d'un environnement de test pour des règles PHP

Afin de répondre au critère de pertinence, l'outil GREENSPECTOR doit proposer des règles de bonnes pratiques de développement qui apportent des gains réels. Il est donc nécessaire d'effectuer des mesures pour comparer du code *vert* (la « bonne » pratique) et du code *gris* (la « mauvaise » pratique) et ainsi estimer les gains apportés par une règle. Il s'agit donc de mettre en place un environnement de test le plus industrialisé possible pour pouvoir lancer et relancer des mesures facilement.

Le référentiel de règles PHP utilisé par l'analyse statique nécessitait d'être revu et mis à jour, car certains clients de l'entreprise souhaitaient utiliser GREENSPECTOR sur des projets PHP. Dans ce cadre, il m'a été demandé de travailler plus particulièrement sur la mise en place d'un environnement de test pour des règles PHP. Nous avons été accompagnés par un consultant externe expert en PHP pour identifier des règles intéressantes et les documenter.

Spécifications

Le référentiel de règles PHP doit contenir les éléments suivants :

- des règles de bonnes pratiques de développement identifiées et documentées;
- des mesures (énergie, temps d'exécution...) associées à chacune de ces règles, pour le code *vert* et *gris*.

Ces mesures sont ensuite stockées en base, afin d'alimenter les calculs de criticités et de priorités mentionnés dans la section .

Une particularité du langage PHP est le fait que le code s'exécute côté serveur (et non côté client). Pour mettre en place un environnement de test, il faut donc simuler des requêtes vers un serveur web qui va exécuter du code PHP et répondre à l'émetteur des requêtes. Les données recherchées pour les mesures sont :

- l'énergie consommée par le serveur pendant l'exécution du code (c'est-à-dire l'intégrale de la puissance consommées dans le temps);
- le temps d'exécution du code;
- la quantité de mémoire vive utilisée (en moyenne) pendant l'exécution du code.

Il s'agit donc de répertorier pour chacune des règles identifiées des exemples de bonnes et mauvaises pratiques à opposer.

Méthodes utilisées

Une première étape a été de mettre en place un serveur web. Nous avons pour cela créé une machine virtuelle sur un serveur interne de GREENSPECTOR. J'ai utilisé l'outil Vagrant² qui permet de simplifier la gestion des machines virtuelles. Un simple fichier (Vagrantfile) permet de décrire le type, la configuration et le provisionnement de la machine virtuelle.

L'outil choisi pour simuler des requêtes vers un serveur web a été Apache JMeter : une application Java permettant de faire des tests fonctionnels, des tests de charge et

2. <https://www.vagrantup.com>

des mesures de performance, et ce pour différents types de protocoles (HTTP, JDBC, TCP, ...). Le choix de cet outil résulte du fait qu'un module GREENSPECTOR pour Apache JMeter existait déjà (et sur lequel j'ai également travaillé pendant mon stage, voir la section 6.2 pour plus de détails). Ce module permet d'intégrer la mesure de consommation d'un serveur dans un plan de test JMeter.

Afin de mesurer la consommation en ressources de la machine virtuelle utilisée comme serveur web, il a également été nécessaire d'installer une sonde. J'ai pour cela comparé deux sondes différentes :

- une sonde DCScope fournie par l'entreprise EasyVirt³ ;
- la sonde PowerAPI⁴.

DCScope est un logiciel permettant d'analyser et de visualiser les données d'une infrastructure virtualisée : état de l'infrastructure, taux d'utilisation des machines virtuelles, consommations en cycles CPU et en RAM des machines virtuelles, consommation électrique des serveurs, coût global, impact CO₂... Elle peut s'utiliser avec 2 types de pilotes :

- un pilote permettant de faire des mesures sur un serveur avec ESX⁵ ;
- un pilote permettant de faire des mesures sur un serveur de type Unix.

L'installation de la sonde (sur une machine virtuelle dédiée) est relativement conséquente. Les mesures de consommation des machines virtuelles monitorées y seront enregistrées dans une base de données. Une API⁶ est proposée pour récupérer ces mesures.

La sonde PowerAPI est une boîte à outils permettant de construire des wattmètres logiciels en récupérant des données brutes depuis différents types de capteurs (capteur physique, interface du processeur). Elle peut s'utiliser avec différents systèmes d'exploitation (Windows, Linux, Mac OS X...), et consiste simplement en une archive à déposer et configurer (selon les caractéristiques du processeur) sur la machine à monitorer. Elle propose plusieurs modules permettant de récupérer des informations sur la puissance consommée par le processeur, notamment :

- *procfs-cpu-simple*, qui utilise le pseudo-système de fichiers présent sur les systèmes de type Unix contenant des informations du noyau sur les processus ;
- *sigar-cpu-simple*, qui utilise l'interface Sigar permettant d'obtenir des informations sur la mémoire, le CPU, et ce pour la majorité des systèmes d'exploitation ;
- *rapl*, qui utilise le pilote RAPL (Running Average Power Limit) d'Intel.

Travail effectué

Plan de test avec Apache JMeter Mon travail a tout d'abord consisté à prendre en main les différents outils cités au paragraphe précédent. J'ai commencé par utiliser Apache JMeter sur quelques plans de test simples, en me basant sur un tutoriel en ligne [2]. Cette prise en main de l'outil m'a permis de créer un plan de test que j'ai par la suite utilisé pour mesurer les règles PHP.

3. <http://www.easyvirt.com>

4. <https://github.com/Spirals-Team/powerapi>

5. ESX est un hyperviseur (natif ou de type 1, c'est-à-dire s'exécutant directement sur une plateforme matérielle) développé par VMware pour déployer et gérer des machines virtuelles

6. interface de programmation applicative

Ce plan de test comprend un élément principal appelé groupe d'unités, dans lequel on configure le nombre d'utilisateurs et le nombre de boucles par utilisateur, comme représenté sur la figure 3. Cet élément contient plusieurs sous-éléments :

- un échantillon qui permet de définir le type de requêtes échangées avec le serveur et de les émettre lors du test, ici une requête HTTP avec les informations suivantes : nom du serveur et chemin d'accès vers le fichier (sur le serveur), comme représenté sur la figure 4 ;
- un récepteur dont le rôle est de récupérer et d'afficher les résultats d'un test, ici le module GREENSPECTOR (détaillé dans la section 6.2) qui sert à intégrer la mesure de consommation du serveur défini précédemment.

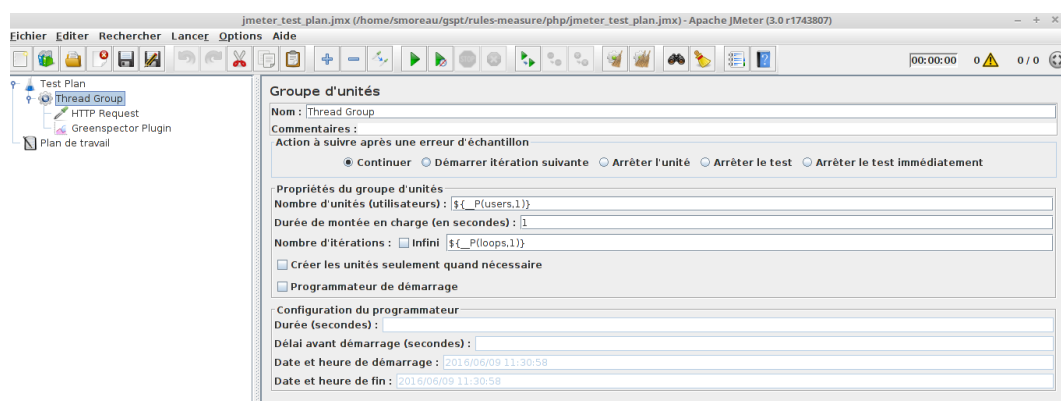


Figure 3 – JMeter : groupe d'unités

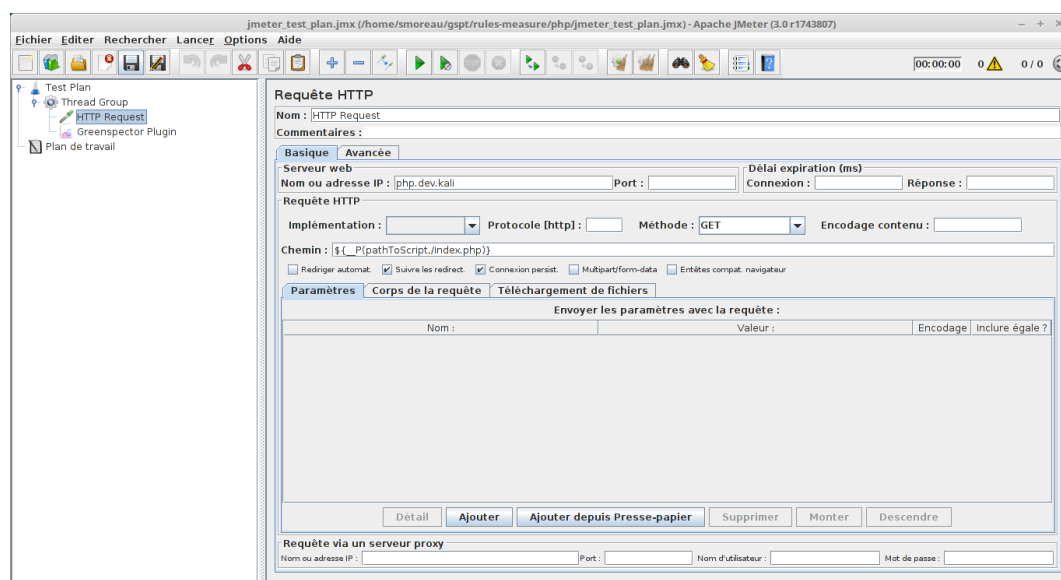


Figure 4 – JMeter : requête HTTP

Afin d'automatiser les tests, il est plus pratique d'utiliser Apache JMeter en ligne de commande en indiquant le chemin vers le fichier .jmx correspondant au plan de test.

Cependant, certains éléments du plan de test doivent pouvoir être modifiés facilement, notamment :

- le nombre d'utilisateurs;
- le nombre de boucles par utilisateur;
- le chemin d'accès du fichier de la requête HTTP.

Pour cela, j'ai utilisé une fonctionnalité proposée par Apache JMeter consistant à variabiliser certains paramètres. Dans le plan de test, il est possible de définir un paramètre de la façon suivante (en indiquant le nom de la variable, suivi d'une valeur par défaut) :

```
${__P(pathToScript,/index.php)}
```

Ainsi, JMeter peut être lancé avec une commande similaire à :

```
./jmeter -n -t pathToTest_plan.jmx -JpathToScript="/monDossier/monScript.php"
```

Dans cet exemple, `-n` est l'option pour le mode sans interface graphique, `-t` l'option pour préciser le chemin vers le plan de test, et l'option `-JpathToScript` instancie la variable `pathToScript` à la valeur `/monDossier/monScript.php`:

Comparaison des sondes de mesure Je me suis ensuite concentrée sur la comparaison des sondes logicielles DCScope et PowerAPI présentées au paragraphe précédent. L'enjeu était de choisir une sonde permettant d'obtenir des mesures de la consommation en ressources de la machine virtuelle utilisée comme serveur web pour un usage à la fois interne (au sein de GREENSPECTOR pour tester les règles de bonnes pratiques de développement), et à la fois externe (quelle(s) sonde(s) proposer aux clients).

Une première étape a donc consisté à étudier l'environnement dans lequel ces sondes seront amenées à être utilisées :

- il n'y a pas forcément d'ESX installé sur les serveurs (ce qui est notamment le cas pour le serveur interne de GREENSPECTOR) car c'est un produit propriétaire payant;
- dans le cas où un ESX est installé, y avoir accès peut être compliqué (or la sonde DCScope a besoin de cet accès);
- les serveurs n'ont pas forcément de sonde physique installée (la sonde DCScope nécessite d'être étalonnée en mesurant la puissance maximale du serveur avec une sonde physique, en revanche, la sonde PowerAPI propose des modules avec ou sans sonde physique);
- la communication avec la sonde (pour lancer une mesure et/ou récupérer les résultats d'une mesure) se fait par HTTP pour la sonde DCScope, et peut se faire par le protocole de communication SSH avec une sonde PowerAPI installée sur une machine virtuelle.

D'autres éléments à prendre en compte dans cette comparaison sont : la fréquence à laquelle les données de mesure sont faites ainsi que les métriques récupérées. La table 1 présente un récapitulatif de ces paramètres.

Sonde	Fréquence de mesure	Métriques
DCScope - driver ESX	20 secondes	CPU, RAM, puissance
DCScope - driver Unix	3 secondes	CPU, RAM
PowerAPI	définissable par l'utilisateur (à partir de 10 ms)	puissance

Table 1 – Comparaison des sondes DCScope et PowerAPI

Au début de mon stage, l'outil GREENSPECTOR intégrait la sonde DCSope uniquement avec le pilote ESX (moyennant un petit travail de mise à jour à effectuer) et une ancienne version de la sonde PowerAPI (utilisée uniquement pour faire des mesures sur PC côté client).

Afin d'utiliser ces différentes sondes, un travail préliminaire d'intégration des sondes dans GREENSPECTOR a été nécessaire et est présenté dans la section 6.1.

Une fois cette intégration effectuée, j'ai lancé des mesures et récupéré les informations fournies par les sondes pour les analyser. Les objectifs étaient de :

- tester la cohérence des différents modules proposés par PowerAPI ;
- tester le bon fonctionnement des sondes sur une machine virtuelle ;
- comparer les mesures fournies par PowerAPI et DCScope ;
- enfin, décider quelle sonde utiliser.

La figure 5 présente une comparaison de 3 modules proposés par PowerAPI. Comme expliqué précédemment, ces modules se basent sur des interfaces différentes pour récupérer les informations liées à l'activité du processeur. Il s'agit donc de les mettre en opposition pour vérifier leur cohérence. Pour cela, j'ai mesuré la consommation en puissance du processus *firefox* sur un PC avec les modules *procfs-cpu-simple*, *sigar-cpu-simple* et *rapl*. Sur cette figure, on peut observer que les données fournies par chacun des modules sont très proches (*rapl* a tendance à indiquer des puissances légèrement plus élevées) et suivent la même évolution.

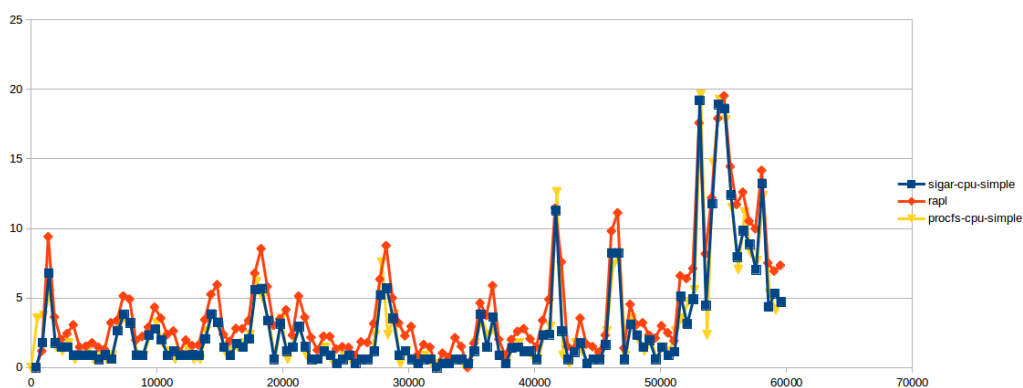


Figure 5 – Mesure d'un processus (*firefox*) avec 3 modules de PowerAPI : en abscisse, le temps en ms, en ordonnée, la puissance en mW

Étant donné que le module *procfs-cpu-simple* ne fonctionne que sur un système de type Unix et que tous les processeurs ne sont pas compatibles avec le pilote RAPL d'Intel, le choix s'est porté sur le module *sigar-cpu-simple*.

Ensuite, j'ai comparé des mesures fournies par PowerAPI et par la sonde DCScope. La figure 6 présente une comparaison du module *sigar-cpu-simple* de PowerAPI avec la sonde DCScope (pour Unix). La mesure concerne ici un test de charge sur un serveur (exécution de scripts PHP en boucle). On peut observer que la sonde PowerAPI semble suivre de plus près la montée en charge du serveur (notamment car les points de mesure sont beaucoup plus nombreux).

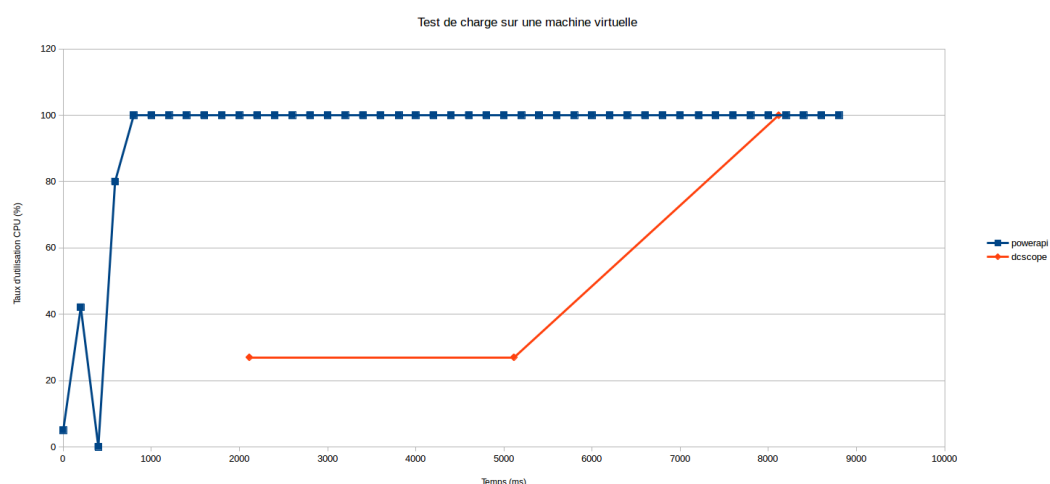


Figure 6 – Mesure d'une machine virtuelle avec le module *sigar-cpu-simple* de PowerAPI et la sonde DCScope pour Unix : en abscisse, le temps en ms, en ordonnée, le taux d'utilisation CPU en %

Sonde	Points positifs	Points négatifs
DCScope - driver ESX	données CPU et RAM	intervalle de mesure très grand, nécessite un ESX
DCScope - driver Unix	données CPU et RAM	intervalle de mesure grand
PowerAPI	intervalles de mesure très fins, installation simple	pas de données RAM

Table 2 – Bilan de la comparaison des sondes DCScope et PowerAPI

Finalement, la décision finale a été de choisir la sonde PowerAPI à la fois pour les mesures et tests en interne, et à la fois comme sonde à proposer aux clients. En effet, l'installation est simple et la fréquence de mesure est élevée. De plus, ce choix permet d'utiliser la même sonde pour des mesures sur PC côté client et des mesures sur serveur (DCScope ne peut pas être utilisée sur un PC).

Scripts de test des règles de bonnes pratiques de développement pour le langage PHP Nous avons été accompagnés par un consultant externe pour travailler sur des règles PHP :

- identification de règles intéressantes ;
- rédaction d'exemples de codes *vert* et *gris* implémentant respectivement les bonnes et mauvaises pratiques ;

- mesures d'énergie et de performance et analyse de ces mesures ;
- rédaction d'une description des règles retenues (car apportant des gains d'énergie, de performance) pour le problème à résoudre et la solution proposée.

Mon travail a consisté plus particulièrement à récupérer les exemples de codes et à les intégrer à un environnement de test (envoi de requête HTTP avec Apache JMeter et utilisation de la sonde PowerAPI) afin d'obtenir des mesures d'énergie et de performance. J'ai ensuite analysé ces mesures afin d'identifier les règles les plus pertinentes.

Les figures 8, 8 et 9 montrent des exemples de codes utilisés pour mesurer et tester les règles de bonnes pratiques de développement. La règle en question concerne l'import de fichiers : en PHP, l'inclusion de fichiers est effectuée à l'exécution (et non au moment de la compilation), ce qui demande des ressources. Lors de l'appel à la fonction *include()* sans indiquer le chemin précis du fichier, PHP cherche ce fichier dans une liste de dossiers définie par la directive de configuration *include_path*. Or si le chemin absolu du fichier est indiqué, PHP n'a aucun traitement supplémentaire à exécuter au moment de l'exécution.

```
$numberOfLoops = 100000;
for ($x = 1; $x <= $numberOfLoops; $x++) {
    include __DIR__.'/index.php';
}
```

Figure 7 – Exemple de code vert pour la règle : Utiliser un chemin absolu pour un import de fichier

```
$numberOfLoops = 100000;
for ($x = 1; $x <= $numberOfLoops; $x++) {
    include './index.php';
}
```

Figure 8 – Exemple de code gris pour la règle : Utiliser un chemin absolu pour un import de fichier

```
$numberOfLoops = 100000;
for ($x = 1; $x <= $numberOfLoops; $x++) {
    include 'index.php';
}
```

Figure 9 – Exemple de code gris pour la règle : Utiliser un chemin absolu pour un import de fichier

Certaines règles identifiées pour le langage PHP font intervenir des requêtes SQL : pour pouvoir faire des mesures il était donc nécessaire d'avoir une base de données sur laquelle faire des requêtes. En me basant sur les exemples de code *vert* et *gris*, j'ai créé un modèle de base de données adéquat, ainsi que des scripts pour alimenter cette base.

Comme mentionné précédemment, j'ai utilisé JMeter comme outil de simulation de requêtes HTTP vers un serveur web. Le scénario de test était le suivant : un nombre X

d'utilisateurs *simulés* effectuent Y fois en boucle une requête vers un script de code PHP stocké sur la machine virtuelle mise en place sur un serveur interne de GREENSPECTOR. Ce script PHP est composé de 3 parties :

- une partie d'initialisation (variables, connexion à la base de données, etc) ;
- une boucle de Z itérations sur la partie de code à tester (c'est-à-dire le code qui varie entre la version *verte* et la version *grise*) ;
- une partie finale (déconnexion de la base de données, etc).

Le paramètre Z doit être suffisamment élevé afin que les parties d'initialisation et finale aient un impact négligeable sur la mesure globale.

Afin d'avoir un panel de mesures le plus complet possible, j'ai fait varier les valeurs des paramètres X, Y et Z.

Résultats

Mesure des règles La table 3 répertorie les principales règles identifiées, en indiquant les gains en énergie et temps d'exécution observés. Les gains indiqués correspondent aux valeurs suivantes des paramètres X, Y et Z précédemment cités :

- X = 1 utilisateur ;
- Y = 1 requête par utilisateur ;
- Z = 10 000 ou 100 000 itérations par script.

Le paramètre Z est variable selon les règles car certaines sont beaucoup plus *lourdes* que d'autres : effectuer une requête SQL vers une base de données n'a pas le même impact qu'incrémenter une simple variable par exemple. Cependant, la valeur de ce paramètre est considéré suffisamment élevée pour négliger l'impact de la *surcouche globale* (envoi de la requête HTTP au serveur, récupération de la réponse, etc) lors du calcul des gains unitaires (où le gain global est divisé par Z).

Les mesures ont été effectuées avec la version 5.6 de PHP.

Règle	Gain relatif (énergie)	Gain relatif (performance)	Commentaire
Utiliser un chemin absolu pour un import de fichier	26%	5%	Règle validée et implémentée
Préférer l'utilisation de 'foreach'	23%	11%	Règle validée et implémentée
Préférer les regex PCRE aux regex POSIX	36%	31%	Règle validée et implémentée
Éviter d'utiliser SELECT * dans les requêtes SQL	40%	38%	Règle validée et implémentée
Éviter les méthodes dans les conditions des boucles	56%	52%	Règle validée et implémentée
Éviter d'effectuer des requêtes SQL à l'intérieur d'une boucle	79%	79%	Règle validée et implémentée
Gérer les erreurs dans le code	71%	70%	Règle validée mais pas encore implémentée
Utiliser PHP 7	-%	40-60%	Règle validée (selon des mesures effectuées par Zend, [4], comparant PHP 5.6 et PHP 7 selon le critère de performance) mais pas encore implémentée
Préférer le JSON au XML	96%	94%	Règle validée mais pas encore implémentée
Ne pas utiliser les références	18%	-2%	Mesures non concluantes, à étudier pour voir s'il y a des gains sur d'autres critères
Préférer les 'Array' aux 'Object'	0%	0%	Mesures non concluantes, à étudier pour voir s'il y a des gains sur d'autres critères

Table 3 – Liste des règles de bonnes pratiques de développement pour PHP

Les gains absolus calculés lors de la mesure des règles font intervenir un certain nombre

de paramètres, tels que :

- la plateforme matérielle sur laquelle sont faites les mesures ;
- le nombre d'itérations (le paramètre Z défini précédemment) à l'intérieur du script de test ;
- d'autres paramètres comme le nombre d'utilisateurs et le nombre de requêtes par utilisateurs dans le cas de règles mesurées côté serveur.

Afin de calculer la criticité des règles (c'est-à-dire les indicateurs entre 1 et 4 mentionnées dans la section 5), il est nécessaire de comparer les règles entre elles sur la base des gains mesurés. En effet, la criticité la plus forte doit correspondre aux règles dont les gains absolus sont les plus élevés parmi l'ensemble des gains mesurés.

Il paraît donc indispensable de fixer l'ensemble des paramètres pouvant intervenir dans la mesure des règles, dans l'objectif de comparer ce qui est comparable. Cependant, il n'est pas toujours possible de se placer à iso-périmètre pour la mesure de toutes les règles : par exemple, la mesure des règles PHP (scripts s'exécutant côté serveur) ne peut pas se faire sur un mobile.

De plus, se baser sur des gains relatifs (afin de s'abstraire des paramètres variables de la mesure) ne paraît pas envisageable non plus pour les raisons suivantes :

- il est plus difficile de comparer les règles entre elles, car une règle peut avoir un fort gain en valeur absolue mais un faible gain en valeur relative (et inversement) ;
- la notion de gain absolu doit être conservée afin de faire les calculs de gains sur l'application globale (en additionnant les gains absolus de chacune des règles enfreintes, et ce à chaque fois que la règle est enfreinte).

Enfin, il ne faut pas oublier que les scripts utilisés pour la mesure peuvent être plus ou moins longs selon la règle testée : par exemple, incrémenter une variable ou effectuer une requête vers une base de données n'a pas le même impact ! Une règle faisant référence à du code dont le temps d'exécution est long aura donc plus de poids qu'une règle dont le temps d'exécution est plus faible.

L'ensemble des mesures effectuées et les résultats associés montrent que la mesure de règles dépend d'un grand nombre de paramètres et d'hypothèses. Notamment, les gains relatifs en énergie et performance selon le nombre d'utilisateurs (paramètre X) et le nombre de requêtes par utilisateur (paramètre Y) peuvent varier de façon assez significative. Même si les valeurs exactes des gains sont à interpréter avec réserve, il est indéniable que certaines règles apportent un gain de façon systématique.

Description des règles Pour chacune des règles implémentées dans le moteur d'analyse statique de GREENSPECTOR, une description est fournie à l'utilisateur pour l'aider à comprendre le principe de la règle et l'aider à appliquer la bonne pratique préconisée. Parmi les règles intéressantes retenues pour l'intégration dans le moteur d'analyse statique, certaines règles étaient déjà implémentées, d'autres non. Il a donc fallu rédiger les descriptions des nouvelles règles, et réviser les autres. Un exemple de description est fourni en annexe A.

6. Évolution de l'outil Meter

L'outil *Meter* de GREENSPECTOR permet d'intégrer des mesures de consommation de ressources (énergie, CPU, mémoire, etc) à des environnements de tests, d'intégration continue. Ces mesures peuvent être faites sur PC, mobile ou serveur. Cet outil est décliné en plusieurs modules, listés dans la table 4.

Module	Plateforme(s) de mesure	Usage
<i>Standalone Meter</i>	PC, serveur	en ligne de commande
<i>API Meter Java</i>	PC, serveur	dans un environnement Java
<i>API Meter JavaScript</i>	PC, serveur	dans un environnement JavaScript
<i>API Meter Android</i>	mobile	dans un environnement Android
<i>Plugin JMeter</i>	PC, serveur	dans un environnement de test Apache JMeter
<i>TestRunner</i>	PC, serveur, mobile	tests standardisés multi-plateformes

Table 4 – Déclinaisons de l'outil Meter

La figure 10 présente le fonctionnement global des différents modules de l'outil *Meter*. Les mesures sont lancées depuis un PC principal (le PC où est installé le TestRunner, ou le PC où est installé Apache JMeter, ou le PC depuis lequel l'*API Meter Java* est appelée, etc). Selon les modules, il est possible d'effectuer des mesures en local, sur un mobile, ou sur un serveur distant. À la fin des mesures, les résultats sont centralisés avant d'être envoyés vers le serveur GREENSPECTOR.

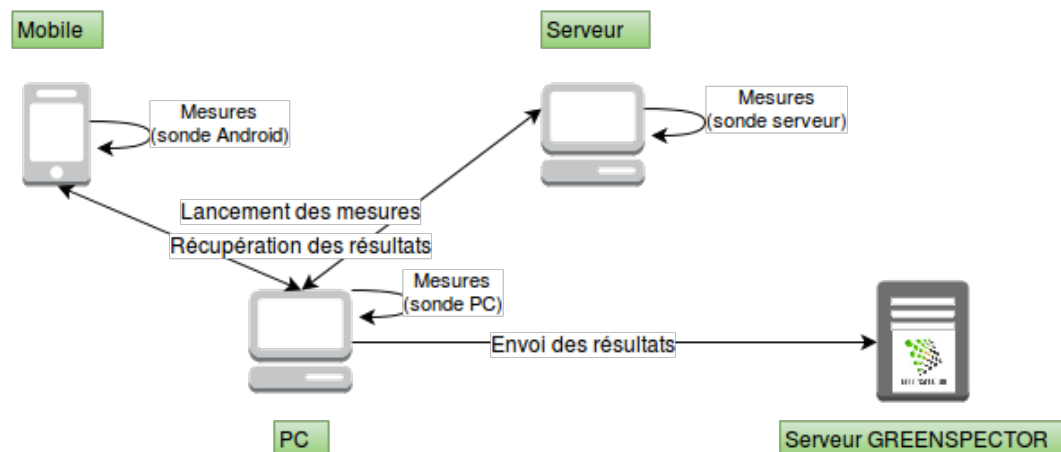


Figure 10 – Fonctionnement général des modules de l'outil Meter

6.1. Intégration de nouvelles sondes dans l'API Meter Java

Comme expliqué dans la section 5, l'outil GREENSPECTOR intégrait au début de mon stage la sonde DCSope uniquement avec le pilote ESX (moyennant un petit travail de mise à jour à effectuer) et une ancienne version de la sonde PowerAPI (utilisée uniquement pour faire des mesures sur PC côté client). Afin d'utiliser les différentes sondes décrites dans la section 5 pour pouvoir les comparer, un travail préliminaire d'intégration des sondes dans GREENSPECTOR a été nécessaire.

Spécifications

Intégrer les sondes dans l'API Meter Java consiste à implémenter la communication entre l'API et les sondes :

- configuration de la sonde ;
- lancement de la mesure ;
- arrêt de la mesure ;
- récupération des résultats.

La figure 11 propose un diagramme de séquence représentant le lien entre les méthodes de l'API Meter Java et les méthodes des classes implémentant une interface *Probe*.

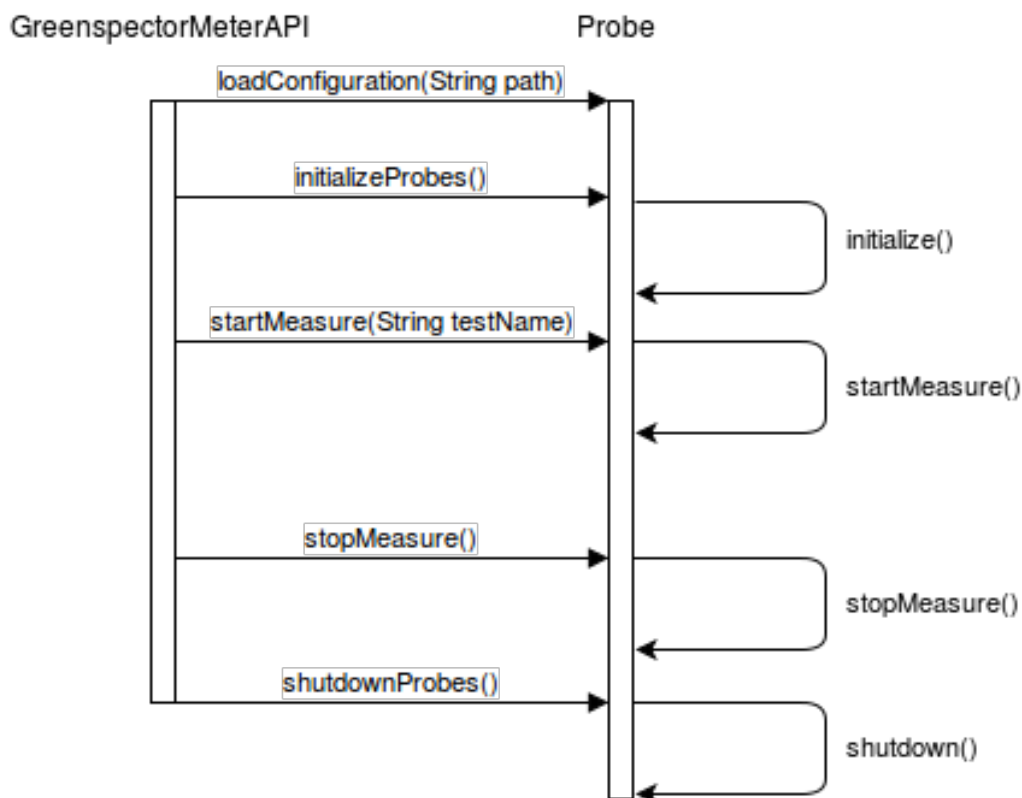


Figure 11 – Diagramme de séquence de l'interface Probe

La communication avec les sondes DCScope se fait par requête HTTP vers la machine virtuelle où est installée la sonde : la sonde effectue des mesures en continu et stocke les

données en base. Une requête HTTP en passant indiquant dans le corps de la requête les paramètres listés ci-dessous permet de récupérer un fichier JSON avec les résultats de la mesure :

- l'identifiant unique universel (UUID) de la machine virtuelle ;
- l'instant de début de la mesure ;
- l'instant de fin de la mesure.

Le fichier JSON récupéré est un tableau d'éléments contenant chacun les informations suivantes :

- instant de la mesure en ms ;
- nom de la machine virtuelle ;
- identifiant unique universel de la machine virtuelle ;
- taux d'utilisation du CPU en % ;
- taux d'utilisation de la RAM en %.

La sonde PowerAPI existe en deux modes : un mode *cli* et un mode *daemon*. Le mode *cli* consiste en une seule commande qui se lance via un terminal, à laquelle on passe des arguments (fréquence des mesures, processus à mesurer, durée de la mesure, etc). Tant que la mesure n'est pas terminée, le processus *powerapi* lancé en ligne de commande ne rend pas la main. Le mode *daemon* propose plusieurs commandes (*start*, *stop*) et le processus ainsi lancé s'exécute en arrière-plan. La configuration doit alors être écrite dans un fichier au préalable. Certains modules de mesure nécessitent les droits de super-utilisateur pour ce mode.

Pour l'intégration dans l'*API Meter Java*, il est nécessaire de pouvoir lancer facilement en ligne de commande la sonde PowerAPI en gérant facilement les paramètres de configuration. De plus, le prérequis consistant à avoir les droits de super-utilisateur sur les serveurs et PC où seront installées les sondes est trop contraignant dans les environnements des clients de GREENSPECTOR. C'est donc le mode *cli* qui a été retenu pour l'intégration dans l'*API Meter Java*.

L'ensemble des paramètres possibles (dont certains sont optionnels) pour la sonde PowerAPI est listé ci-dessous :

- le module de mesure (*procfs-cpu-simple*, *sigar-cpu-simple*, etc) ;
- la fréquence de mesure ;
- les identifiants des processus à mesurer ;
- les noms des processus à mesurer ;
- la façon dont les données sont agrégées sur un intervalle de mesure (valeur maximale, minimale, somme, moyenne, etc) ;
- la sortie voulue pour les résultats (sortie standard, écriture dans un fichier, affichage d'un graphe) ;
- la durée de la mesure.

La figure 12 montre un exemple de mesure du processus *firefox* sur PC à une fréquence de 500 ms.

```
more@ubuntu:~/Lenovo-650-70 /tools/powerapi-ctl (master *) $ ./bin/powerapi modules sigar-cpu-simple monitor --frequency 500 --apps firefox --agg mean --console
muid=961fcee8-eb93-4feb-abdd-clf9bc23cbf0;timestamp=1473793208340;targets=firefox;devices=cpu;power=0.0 mW
muid=961fcee8-eb93-4feb-abdd-clf9bc23cbf0;timestamp=1473793208845;targets=firefox;devices=cpu;power=50.0 mW
muid=961fcee8-eb93-4feb-abdd-clf9bc23cbf0;timestamp=1473793209355;targets=firefox;devices=cpu;power=0.0 mW
muid=961fcee8-eb93-4feb-abdd-clf9bc23cbf0;timestamp=1473793209855;targets=firefox;devices=cpu;power=0.0 mW
muid=961fcee8-eb93-4feb-abdd-clf9bc23cbf0;timestamp=1473793210355;targets=firefox;devices=cpu;power=0.0 mW
muid=961fcee8-eb93-4feb-abdd-clf9bc23cbf0;timestamp=1473793210855;targets=firefox;devices=cpu;power=0.0 mW
muid=961fcee8-eb93-4feb-abdd-clf9bc23cbf0;timestamp=1473793211355;targets=firefox;devices=cpu;power=0.0 mW
muid=961fcee8-eb93-4feb-abdd-clf9bc23cbf0;timestamp=1473793211855;targets=firefox;devices=cpu;power=0.0 mW
muid=961fcee8-eb93-4feb-abdd-clf9bc23cbf0;timestamp=1473793212355;targets=firefox;devices=cpu;power=0.0 mW
muid=961fcee8-eb93-4feb-abdd-clf9bc23cbf0;timestamp=1473793212855;targets=firefox;devices=cpu;power=0.0 mW
muid=961fcee8-eb93-4feb-abdd-clf9bc23cbf0;timestamp=1473793213355;targets=firefox;devices=cpu;power=0.0 mW
muid=961fcee8-eb93-4feb-abdd-clf9bc23cbf0;timestamp=1473793213855;targets=firefox;devices=cpu;power=538.4615384615385 mW
muid=961fcee8-eb93-4feb-abdd-clf9bc23cbf0;timestamp=1473793214355;targets=firefox;devices=cpu;power=787.2538868103627 mW
muid=961fcee8-eb93-4feb-abdd-clf9bc23cbf0;timestamp=1473793214854;targets=firefox;devices=cpu;power=270.6185567018309 mW
muid=961fcee8-eb93-4feb-abdd-clf9bc23cbf0;timestamp=1473793215355;targets=firefox;devices=cpu;power=279.2746115989637 mW
muid=961fcee8-eb93-4feb-abdd-clf9bc23cbf0;timestamp=1473793215855;targets=firefox;devices=cpu;power=214.28571428571428 mW
muid=961fcee8-eb93-4feb-abdd-clf9bc23cbf0;timestamp=1473793216355;targets=firefox;devices=cpu;power=328.125 mW
muid=961fcee8-eb93-4feb-abdd-clf9bc23cbf0;timestamp=1473793216855;targets=firefox;devices=cpu;power=187.6923876923877 mW
muid=961fcee8-eb93-4feb-abdd-clf9bc23cbf0;timestamp=1473793217355;targets=firefox;devices=cpu;power=945.0 mW
muid=961fcee8-eb93-4feb-abdd-clf9bc23cbf0;timestamp=1473793217855;targets=firefox;devices=cpu;power=765.625 mW
muid=961fcee8-eb93-4feb-abdd-clf9bc23cbf0;timestamp=1473793218354;targets=firefox;devices=cpu;power=949.748743718593 mW
muid=961fcee8-eb93-4feb-abdd-clf9bc23cbf0;timestamp=1473793218855;targets=firefox;devices=cpu;power=633.1058291457286 mW
muid=961fcee8-eb93-4feb-abdd-clf9bc23cbf0;timestamp=1473793219355;targets=firefox;devices=cpu;power=53.83838383838383 mW
muid=961fcee8-eb93-4feb-abdd-clf9bc23cbf0;timestamp=1473793219855;targets=firefox;devices=cpu;power=158.29145728643215 mW
muid=961fcee8-eb93-4feb-abdd-clf9bc23cbf0;timestamp=1473793220355;targets=firefox;devices=cpu;power=0.0 mW
muid=961fcee8-eb93-4feb-abdd-clf9bc23cbf0;timestamp=1473793220855;targets=firefox;devices=cpu;power=0.0 mW
muid=961fcee8-eb93-4feb-abdd-clf9bc23cbf0;timestamp=1473793221355;targets=firefox;devices=cpu;power=52.76381909547739 mW
muid=961fcee8-eb93-4feb-abdd-clf9bc23cbf0;timestamp=1473793221854;targets=firefox;devices=cpu;power=0.0 mW
muid=961fcee8-eb93-4feb-abdd-clf9bc23cbf0;timestamp=1473793222355;targets=firefox;devices=cpu;power=887.6923876923877 mW
muid=961fcee8-eb93-4feb-abdd-clf9bc23cbf0;timestamp=1473793222855;targets=firefox;devices=cpu;power=326.42487846632124 mW
muid=961fcee8-eb93-4feb-abdd-clf9bc23cbf0;timestamp=1473793223354;targets=firefox;devices=cpu;power=0.0 mW
muid=961fcee8-eb93-4feb-abdd-clf9bc23cbf0;timestamp=1473793223855;targets=firefox;devices=cpu;power=430.7692387692388 mW
muid=961fcee8-eb93-4feb-abdd-clf9bc23cbf0;timestamp=1473793224355;targets=firefox;devices=cpu;power=852.7918781725887 mW
muid=961fcee8-eb93-4feb-abdd-clf9bc23cbf0;timestamp=1473793224855;targets=firefox;devices=cpu;power=795.4545454545455 mW
muid=961fcee8-eb93-4feb-abdd-clf9bc23cbf0;timestamp=1473793225355;targets=firefox;devices=cpu;power=261.1940298507463 mW
muid=961fcee8-eb93-4feb-abdd-clf9bc23cbf0;timestamp=1473793225855;targets=firefox;devices=cpu;power=53.29949238578679 mW
```

Figure 12 – Exemple de mesure du processus firefox sur PC avec la sonde PowerAPI

L'API Meter doit permettre à l'utilisateur de définir :

- les informations liées à l'application qu'il souhaite mesurer (nom de l'application, version, nom des différents tests et leur nombre d'itérations associé, nom des plateformes de mesure);
- quelle(s) sonde(s) il souhaite utiliser;
- pour la sonde DCScope :
 - les identifiants des machines virtuelles à mesurer;
 - l'URL d'accès à la sonde;
- pour la sonde PowerAPI :
 - le chemin vers le binaire de PowerAPI;
 - les éventuels identifiants ou noms des processus pour PowerAPI (par défaut, tous les processus sont mesurés);
 - les informations de connexion SSH dans le cas d'une mesure sur un serveur distant (nom d'hôte, utilisateur);
- les informations liées au compte GREENSPECTOR (adresse email et URL du serveur).

Ainsi, l'utilisateur sera capable d'effectuer des mesures :

- en local (sur la machine où l'API Meter Java est installée) avec la sonde PowerAPI;
- sur un serveur distant (avec la sonde PowerAPI ou la sonde DCScope).

Méthodes utilisées

La sonde PowerAPI en mode *cli* nécessite de lancer un processus en ligne de commande sans bloquer l'exécution du programme Java : pour gérer ces différents processus, je me suis donc servie de la classe abstraite *java.lang.Process* pour gérer les processus. Cette classe fournit des méthodes pour attendre la terminaison d'un processus, récupérer la valeur de sortie, ... Deux méthodes existent pour créer un nouveau processus. Il est possible d'instancier une sous-classe de *Process* avec la méthode *Runtime.exec()*.

```
Process proc = Runtime.getRuntime().exec("ls");
```

Il est aussi possible d'instancier la classe *ProcessBuilder* puis d'appeler la méthode *ProcessBuilder.start()* pour instancier une sous-classe de *Process*.

```
ProcessBuilder pb = new ProcessBuilder("ls");
Process proc = pb.start();
```

J'ai également utilisé d'autres classes Java telles que *OutputStream* pour gérer les flux de données sur l'entrée et la sortie standard des processus créés, ou *BufferedReader* concernant la lecture des résultats que la sonde PowerAPI écrit dans un fichier.

Travail effectué

Configuration La première étape a été de modifier la structure des éléments de configuration de l'*API Meter* que l'utilisateur doit renseigner. Cette configuration se fait dans 2 fichiers JSON, et leur structure est ensuite directement traduite par des classes et attributs dans l'*API Meter Java*.

Sonde DCScope J'ai ensuite entamé l'intégration dans l'*API Meter Java* de la sonde DCScope avec pilote Unix. Pour cela, j'ai commencé par identifier les éléments communs aux deux versions des sondes (pilote ESX et pilote Unix) pour ne pas les dupliquer. Puis, j'ai intégré les nouveaux éléments spécifiques à la sonde avec pilote Unix, à savoir :

- la récupération différente des identifiants des machines virtuelles;
- la création de la requête HTTP pour récupérer les résultats;
- la récupération des résultats des mesures (moins de métriques avec la sonde pour Unix).

Un attribut de la classe implémentant la sonde DCScope permet d'indiquer quel est le mode choisi (*VMWare* pour le pilote ESX et *Unix* pour le pilote Unix). L'exemple suivant montre la construction de la requête HTTP selon le mode choisi par l'utilisateur.

```
private JSONArray retrieveIdMetricsWithTime(String epoch,
String idMachine, Long startTime, Long endTime)
throws IOException {
String url = this.easyvirtUrl;
if (this.mode.equals("VMWare")) {
url += "?command=listidMetrics";
} else { // this.mode = "Unix"
url += "?command=listidMetricsUnix";
}
url += "&epoch=" + epoch +
"&id=" + idMachine +
"&startTime=" + startTime +
"&endTime=" + endTime;
return readMetricsFromURL(url);
}
```

Sonde PowerAPI Enfin, j'ai modifié le code intégrant la sonde PowerAPI pour utiliser la dernière version de la sonde qui comporte plus de fonctionnalités.

— Configuration de la sonde (méthode *initialize*)

Il s'agit tout d'abord d'affecter des valeurs aux attributs de la classe *PowerAPIProbe*. Ces valeurs sont celles indiquées dans les fichiers de configuration par l'utilisateur. Puis, il faut lancer la sonde pour qu'elle soit prête au moment où l'utilisateur souhaitera démarrer une mesure. Pour cela, un processus *bash* est lancé (mesure locale) ou une connexion SSH est démarrée (mesure distante). Une fois ce processus lancé, il faut lui envoyer sur son entrée standard les commandes pour se placer dans le dossier d'installation de PowerAPI puis exécuter le binaire pour démarrer la sonde (en indiquant les processus à mesurer et le nom du fichier dans lequel écrire les résultats).

— Lancement de la mesure (méthode *startMeasure*)

Il s'agit ici de démarrer une mesure : le fichier dans lequel la sonde écrit les résultats est effacé. Pour une mesure locale, les méthodes fournies par la classe *File* de Java sont utilisées pour vérifier l'existence d'un fichier et le supprimer. Pour une mesure distante, le même principe que précédemment est appliqué : une connexion SSH est ouverte, à laquelle on envoie un flux sur l'entrée standard correspondant à une commande *rm* pour effacer le fichier voulu.

— Arrêt de la mesure (méthode *stopMeasure*)

Au moment où l'utilisateur souhaite stopper une mesure, le fichier contenant les résultats est lu et parsé pour stocker les valeurs.

— Arrêt de la sonde et envoi des résultats (méthode *shutdown*)

Il s'agit ici de stopper l'exécution du binaire *powerapi* et d'envoyer les résultats au serveur GREENSPECTOR ou d'écrire sur le disque les résultats, selon la configuration indiquée par l'utilisateur.

Résultats

En annexe C est présenté un exemple d'utilisation de l'*API Meter Java*. Elle est ici intégrée dans un test Selenium⁷. Selenium fournit un pilote permettant d'automatiser des tests avec un navigateur web. Le scénario de test est le suivant :

- lancer le navigateur Firefox ;
- charger la page <https://www.youtube.com/> ;
- effectuer une recherche avec le mot-clé *youtube* (*test_Search*) ;
- ne rien faire (*test_Idle*) ;
- fermer le navigateur.

6.2. Mise à jour du plugin pour JMeter

Le plugin pour JMeter permet d'intégrer des mesures de consommation à des plans de test JMeter. L'objectif est également d'utiliser l'outil JMeter comme *sonde* afin d'enrichir les résultats fournis par GREENSPECTOR. Ce plugin appelle les méthodes de l'*API Meter Java*.

7. <http://docs.seleniumhq.org/>

Spécifications

Un plugin pour JMeter existait déjà et mon travail a consisté à le reprendre pour y ajouter des fonctionnalités :

- permettre de paramétrer le test (nom du test, plateforme, noms des processus à mesurer) ;
- récupérer certains résultats fournis par JMeter (temps de réponse d'une requête) pour les ajouter aux mesures de consommation provenant des sondes ;
- permettre l'utilisation du module sans sonde (DCScope ou PowerAPI) pour ne récupérer que les métriques données par JMeter.

Les résultats fournis par JMeter comme le temps de réponse d'une requête ont un format différent des résultats provenant des mesures faites avec les sondes DCScope ou PowerAPI. Pour ces dernières, les résultats sont un ensemble de valeurs chacune associées à un instant donné de la mesure, c'est-à-dire un ensemble de valeurs instantanées tout au long de la mesure. En revanche, les résultats donnés par JMeter correspondent à des valeurs globales sur l'ensemble de la mesure. Il a donc été nécessaire d'adapter le service récupérant les valeurs des mesures pour s'adapter à ces deux types de format.

Structure d'un plugin pour JMeter JMeter suit le patron de conception modèle-vue-contrôleur. Un plugin est ainsi composé d'une classe pour la partie graphique (vue), et d'une autre classe pour la partie contrôleur.

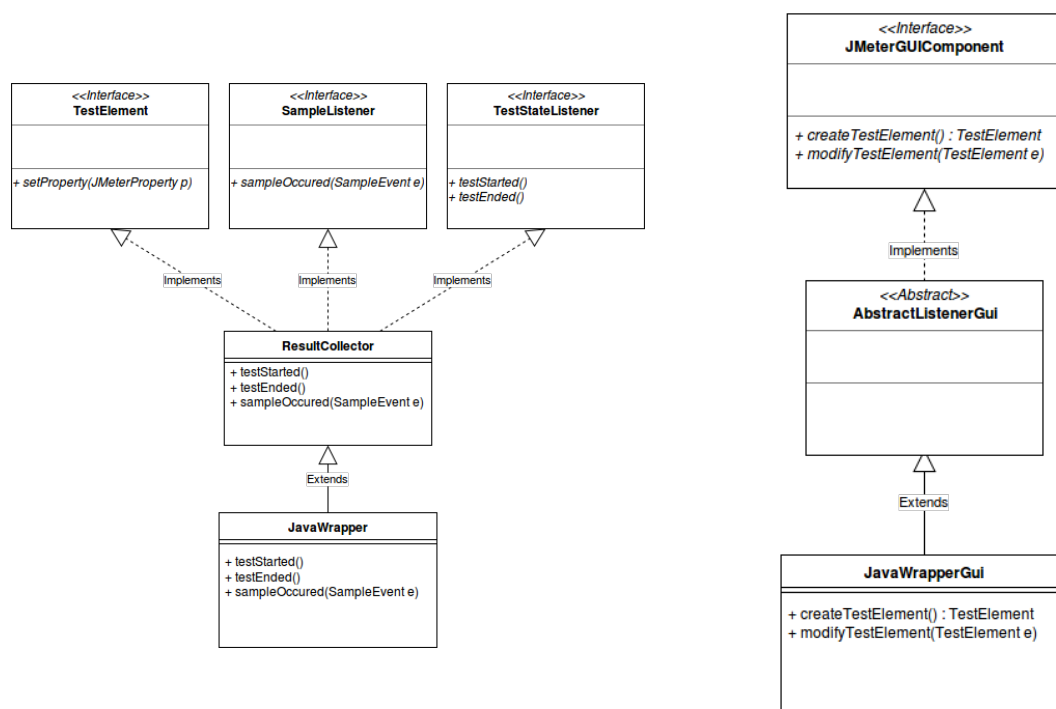


Figure 13 – Diagramme de classes du plugin pour JMeter

La figure 13 présente un diagramme de classes permettant d'expliquer le fonctionnement de ce plugin. La partie vue étend la classe abstraite *AbstractListenerGui*, la partie

contrôleur étend la classe *ResultCollector* : l'objectif du plugin est en effet de collecter les données issues des plans de tests effectués. Les méthodes *testStarted* et *testEnded* sont appelées au début et à la fin du plan de test, la méthode *sampleOccured* est quand à elle appelée à chaque fois qu'un échantillon s'est produit (pour un plan de test constitué de 10 utilisateurs effectuant chacun 3 requêtes en boucle 10 fois, un échantillon correspond à une requête, il y aura donc ici 300 échantillons au total).

Méthodes utilisées

Pour le travail effectué sur ce plugin, je me suis essentiellement basée sur la Javadoc de JMeter [1] et sur le code source de l'outil afin de comprendre comment sont implémentés et utilisés les autres éléments dont la fonctionnalité est similaire à celle souhaitée pour le plugin GREENSPECTOR.

Travail effectué

Mon travail a ici consisté à ajouter des fonctionnalités sur le plugin pour JMeter déjà existant et à intégrer les évolutions de l'*API Meter Java* présentés à la section 6.1.

L'utilisation principale de l'outil de test Apache JMeter se fait dans un environnement de test automatisé, c'est-à-dire en ligne de commande (et non depuis l'interface graphique). J'ai donc implémenté le fait de pouvoir passer en ligne de commande les paramètres dont ont besoin les méthodes de l'*API Meter Java* (nom du test, et éventuellement noms ou identifiants des processus à mesurer). Pour cela, j'ai utilisé l'interface *JMeterProperty* qui permet de définir une propriété à un élément. Ici, l'élément est une instance de la classe *JavaWrapper* (qui hérite de *ResultCollector*). La notion de propriété permet de faire le lien entre un nom et une valeur. Par exemple, j'ai associé la valeur `$_P(gsptTestName,JMeterTest)` au nom `GreenspectorPlugin.test_name`. Ainsi, l'utilisateur pourra définir la valeur de la propriété `GreenspectorPlugin.test_name` en ajoutant le paramètre `-JgsptTestName=leNomDeMonTest` lors du lancement de Apache JMeter en ligne de commande. Les extraits de code ci-dessous expliquent comment ce mécanisme est implémenté.

```
public class JavaWrapper extends ResultCollector {
    ...
    private static final String GREENSPECTOR_TEST_NAME =
        "GreenspectorPlugin.test_name";
    ...
    public JavaWrapper() {
        super();
        setProperty(new StringProperty(GREENSPECTOR_TEST_NAME,
            "${_P(gsptTestName,JMeterTest)}"));
        ...
    }
}
```

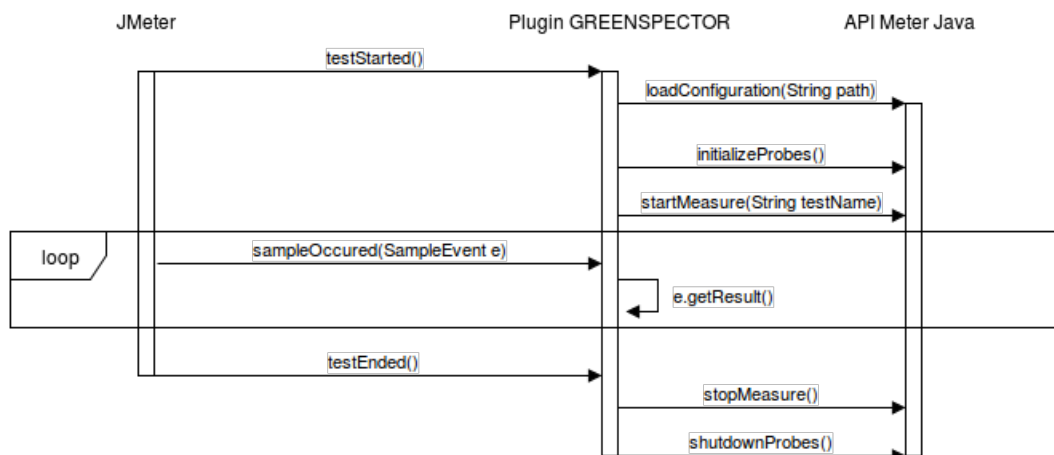


Figure 14 – Diagramme de séquences du plugin pour JMeter

La figure 14 explique les différentes étapes lors de l'exécution d'un test JMeter et comment l'appel aux méthodes de l'API Meter Java y est intégré.

Résultats

Pour utiliser le plugin GREENSPECTOR pour JMeter, il faut l'ajouter au moment de la création d'un plan de test. La figure 15 illustre comment cet ajout se fait à partir de l'interface graphique.

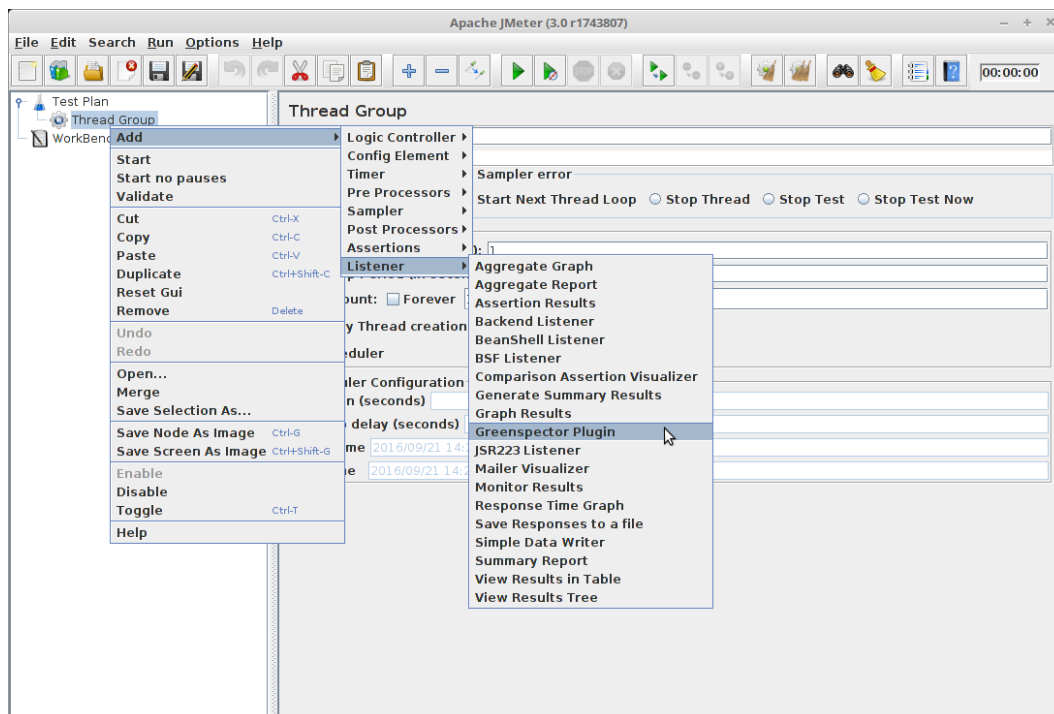
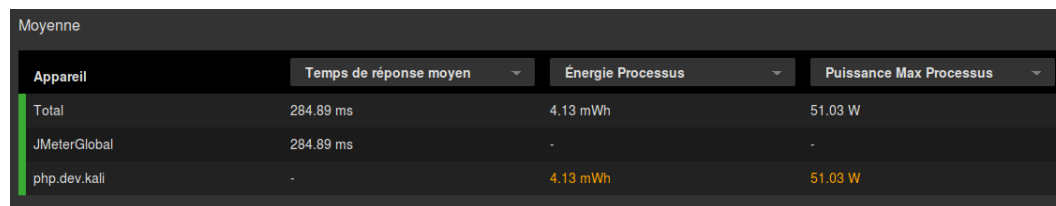


Figure 15 – Utilisation du plugin GREENSPECTOR pour JMeter

Après avoir créé entièrement le plan de test, il suffit de lancer un test (soit en cliquant

sur la flèche verte en haut, soit en lançant JMeter en ligne de commande).

Les résultats sont ensuite affichés sur l'interface web comme le montre la figure 16. Le temps de réponse moyen est une métrique calculée à partir des données fournies par JMeter. L'énergie et la puissance maximale sont calculées à partir des mesures par la sonde PowerAPI.



Appareil	Temps de réponse moyen	Énergie Processus	Puissance Max Processus
Total	284.89 ms	4.13 mWh	51.03 W
JMeterGlobal	284.89 ms	-	-
php.dev.kali	-	4.13 mWh	51.03 W

Figure 16 – Résultats sur l'interface web de mesures utilisant le plugin pour JMeter

6.3. Création d'un module indépendant

Spécifications

Le besoin identifié est ici de pouvoir utiliser l'outil *Meter* en ligne de commande pour l'intégrer à des outils de tests ou d'intégration continue. En effet, l'outil *Meter* n'existait pour l'instant que sous la forme d'API (pour Java, JavaScript) ou de plugin (pour JMeter). Un module indépendant doit donc permettre de démarrer une mesure en ligne de commande, effectuer des tâches, puis stopper la mesure.

L'objectif est ici de pouvoir passer en ligne de commande les paramètres des méthodes de l'API *Meter* :

- le chemin d'accès aux fichiers JSON de configuration (facultatif) ;
- le nom du test ;
- les noms ou identifiants des processus à mesurer par la sonde PowerAPI (facultatif) ;
- le nombre d'itérations effectuées pendant la mesure.

Méthodes utilisées

J'ai utilisé une librairie Java (JCommander⁸) qui permet de simplifier l'analyse des arguments passés en ligne de commande.

J'ai également utilisé un module fourni par Maven (*maven-assembly-plugin*) pour créer un JAR exécutable au moment de la compilation. Ce module s'insère dans le fichier *pom.xml* du projet.

Travail effectué

Ce module *Meter* indépendant est constitué d'une classe principale (GreenspectorMeter) avec une méthode *main(String ... args)*. C'est cette méthode qui est appelée par le JAR après compilation du module. Le paramètre *args* représente les paramètres avec lesquels est exécuté le JAR en ligne de commande.

L'exécution du JAR en ligne de commande s'effectue de la façon indiquée ci-dessous.

8. <http://www.jcommander.org>

```
java -jar greenspector-meter.jar [command] [options]
```

J'ai défini 4 commandes différentes pour le JAR, chacune d'entre elles liées à des méthodes de l'API Meter Java :

- *init*, qui appellera les méthodes *loadConfiguration* puis *initializeProbes*;
- *start*, qui appellera la méthodes *startMeasure*;
- *stop*, qui appellera la méthodes *stopMeasure*;
- *shutdown*, qui appellera la méthode *shutdownProbes*.

Pour chacune de ces commandes, des options peuvent s'ajouter, qui seront l'équivalent des paramètres des méthodes de l'API.

Pour chaque commande, une classe correspondante est définie. L'exemple ci-dessous correspond à la commande *init*.

```
@Parameters(commandDescription = "Load the configuration and initialize  
the probes")  
public class CommandInit {  
  
    @Parameter(names = {"--path", "-p"}, description = "The path to  
configuration files")  
    public String path = GreenspectorMeterAPI.DEFAULT_LOCAL_GREENSPECTOR_PATH;  
  
    @Parameter(names = {"--procNames", "-n"}, variableArity = true,  
description = "The names of the processes to monitor with PowerAPI")  
    public List<String> procNames = new ArrayList<>();  
  
    @Parameter(names = {"--procIds", "-i"}, variableArity = true,  
description = "The pids of the processes to monitor with PowerAPI")  
    public List<String> procIds = new ArrayList<>();  
  
}
```

Les annotations *@Parameters* et *@Parameter* sont utilisées par la librairie JCommander. La méthode *main* de la classe principale est implémentée de la façon suivante :

- instanciation de la classe JCommander;
- ajout des 4 commandes définies plus haut;
- appel de la méthode *parse* de la classe JCommander pour parser les arguments fournis en lignes de commande;
- disjonction de cas selon la valeur de la commande et des éventuelles options associées.

L'extrait de code ci-dessous illustre l'explication précédente.

```
public static void main (String... args) {  
  
    GreenspectorMeterAPI gsptMeterAPI = new GreenspectorMeterAPI();  
    JCommander jc = new JCommander(gsptMeterAPI);
```

```

try {
    CommandInit init = new CommandInit();
    jc.addCommand("init", init);
    CommandStart start = new CommandStart();
    jc.addCommand("start", start);
    CommandStop stop = new CommandStop();
    jc.addCommand("stop", stop);
    CommandShutdown shutdown = new CommandShutdown();
    jc.addCommand("shutdown", shutdown);

    jc.parse(args);

    String cmd = jc.getParsedCommand();

    if (cmd == null) {
        LOGGER.error("Please specify a command to launch the jar file");
        jc.usage();
        System.exit(1);
    }

    switch (cmd) {
        case "init":
            ...
        case "start":
            ...
        case "stop":
            ...
        case "shutdown":
            ...
    }
    ...
}
...
}

```

Enfin, il était nécessaire de sauvegarder *l'état* du module pour les relire lorsque l'utilisateur exécutera le JAR avec une commande différente. Par exemple, il est nécessaire d'enregistrer le fait qu'une mesure a été démarrée ou non (par un booléen qui passe à la valeur *true* après une commande *start*) pour indiquer un message d'erreur adéquat à l'utilisateur s'il essaie de refaire la commande *start* alors qu'une mesure est déjà en cours. Pour cela, j'ai implémenté l'écriture d'un fichier JSON à la fin des différentes actions effectuées pour chaque commande. Ce fichier JSON est ensuite lu au début de l'invocation d'une commande, et un message d'avertissement est éventuellement affiché si l'état décrit dans le fichier ne correspond pas à un état attendu. L'ensemble des informations nécessaires est regroupé dans une classe qui est sérialisée/désérialisée à l'aide de la librairie Gson.

Résultats

Lors de l'exécution du JAR sans aucune commande, un message d'aide apparaît pour expliquer à l'utilisateur l'usage attendu, comme l'illustre la figure 17.

```
smoreau@smoreau-Lenovo-G50-70 ~/Tests (master *) $ java -jar greenspector-meter-1.1.4.jar
[greenspector] Please specify a command to launch the jar file
Usage: <main class> [options] [command] [command options]
Commands:
  init          Load the configuration and initialize the probes
    Usage: init [options]
    Options:
      --path, -p          The path to configuration files
                          Default: ./greenspector/
      --procIds, -i       The pids of the processes to monitor with PowerAPI
                          Default: []
      --procNames, -n     The names of the processes to monitor with PowerAPI
                          Default: []

  start         Start a measure
    Usage: start [options]
    Options:
      --testName, -t      The name of the current test case
                          Default: <empty string>

  stop          Stop the current measure
    Usage: stop [options]
    Options:
      --iterations, -i   The number of iterations for the current test case
                          Default: 1

  shutdown      Shutdown the probes
    Usage: shutdown [options]
```

Figure 17 – Message d'aide au lancement du module indépendant Meter

7. Évolution du tableau de bord utilisateur : implémentation de règles dynamiques sur les ressources

La version 1.7.0 de GREENSECTOR intègre des évolutions majeures de l'outil, notamment un nouveau tableau de bord sur l'interface web, ainsi qu'un outil pour lancer des tests standardisés multi-plateformes (PC Linux ou Windows, serveur, mobile). Cet outil nommé *TestRunner* permet de faire de la mesure dynamique sur des applications web (test d'une URL) ou Android (test d'un APK). Les données issues de ces mesures sont ensuite enregistrées et analysées pour afficher un bilan de l'application testée sur le tableau de bord.

Les figures 18 et 19 montrent l'évolution (respectivement, version 1.6.X et version 1.7.X) du tableau de bord associé à une application sur l'interface web.

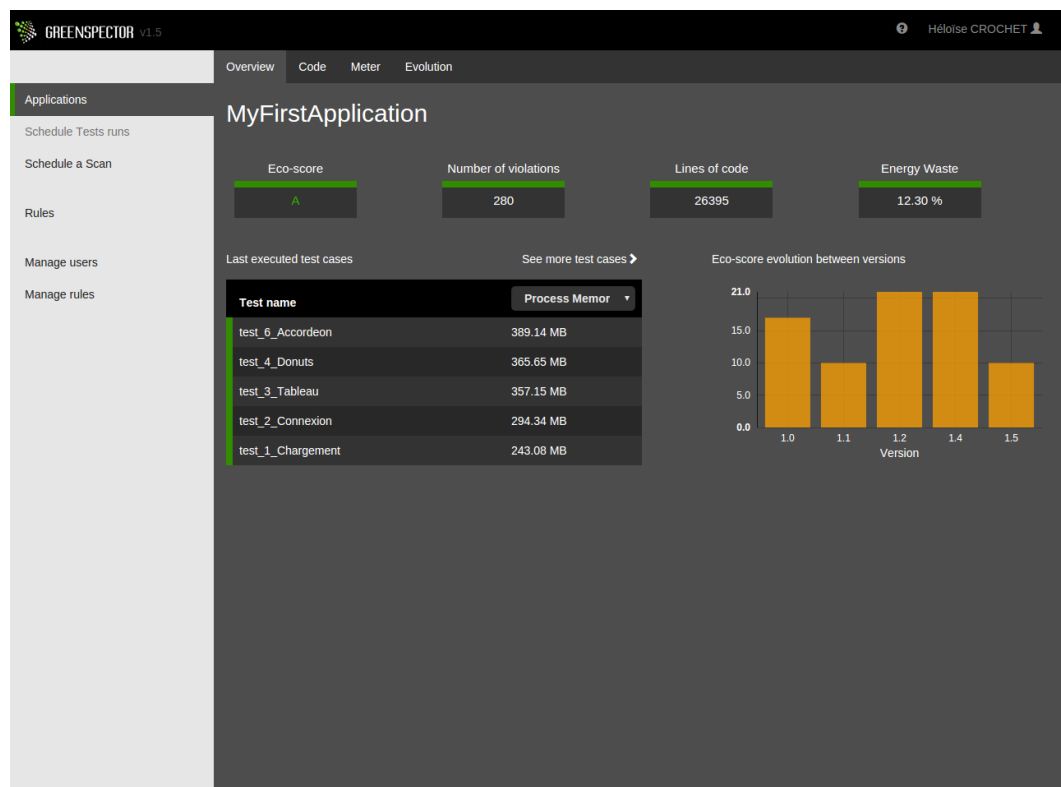


Figure 18 – Ancien tableau de bord

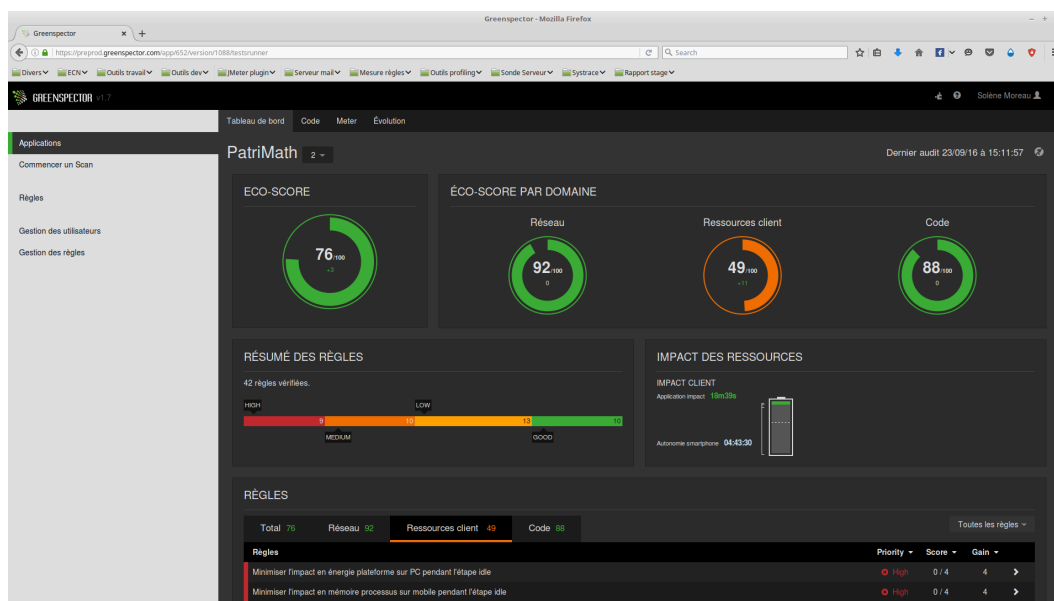


Figure 19 – Nouveau tableau de bord

Comme décrit dans la section 5, ce tableau de bord présente un écoscore global, déduit des écoscores de chaque domaine (code, réseau, ressources). Ces scores sont calculés par le moteur d'analyse statique pour le domaine code, et par le moteur d'analyse dynamique pour les domaines réseau et ressources.

Une des évolutions majeures de la version 1.7.0 concerne le moteur d'analyse dynamique, inexistant dans les précédentes versions. Ce moteur analyse des mesures dynamiques selon un référentiel de règles.

Les données utilisées par les règles dynamiques du domaine réseau sont issues d'un outil externe intégré au *TestRunner* : *phantomas*⁹. Cet outil permet de collecter des métriques pour analyser les requêtes échangées entre un serveur et un client. Voici un extrait des métriques renvoyées par *phantomas* :

- le nombre total de requêtes HTTP échangées ;
- le nombre de requêtes POST ;
- le nombre de réponses de type CSS ;
- le nombre de ressources statiques non gzippées ;
- etc.

Pour le domaine ressources, les données utilisées proviennent des mesures renvoyées par les sondes PC/serveur (PowerAPI) et Android.

Une règle dynamique est notamment composée :

- d'une métrique (par exemple, le nombre de requêtes HTTP envoyées entre le client et le serveur) ;
- d'un ensemble de seuils permettant d'attribuer un écoscore à la règle (par exemple, 75% du score maximal s'il y a entre 5 et 10 requêtes, 100% du score maximal s'il y a moins de 5 requêtes, etc).

9. <https://github.com/macbre/phantomas>

Pendant mon stage, j'ai participé à l'implémentation de ce moteur d'analyse dynamique (plus particulièrement pour le domaine ressources), ainsi qu'à la rédaction des règles associées.

Spécifications

Les règles pour le domaine ressources sont réparties en 3 grandes catégories :

- des règles qui vérifient la présence des données attendues sur chaque plateforme (PC, serveur ou mobile) ;
- des règles qui comparent la consommation de ressources par rapport à la version précédente de l'application ;
- des règles qui comparent la consommation de ressources par rapport à des valeurs de référence (par exemple, consommation d'un navigateur web avant chargement d'une URL).

Sur PC et serveur, les sondes utilisées permettent de mesurer l'énergie de la plateforme (en Wh). Sur un mobile, les sondes Android développées par GREENSPECTOR permettent de mesurer davantage de métriques :

- la décharge de la plateforme, en Ah ;
- les données échangées par le processus, en bytes ;
- la mémoire du processus, en Mo ;
- le taux d'utilisation du CPU pour la plateforme, en % ;
- le taux d'utilisation du CPU pour le processus, en %.

Ceux sont l'ensemble des métriques attendues selon le type de plateforme.

Le *TestRunner* permet de lancer des tests standardisés. Deux types de tests sont actuellement proposés : le test d'une URL (sur un PC ou un mobile) et le test d'un APK ¹⁰ (sur un mobile). Ces tests standardisés sont constitués de plusieurs étapes dont 3 sont toujours présentes :

- une étape *reference* qui permet de mesurer des valeurs de référence pour la plateforme ;
- une étape *load* pendant laquelle l'URL est chargée dans le navigateur, ou bien l'APK est lancé ;
- une étape *idle* pour mesurer la consommation de ressources une fois l'URL complètement chargée, ou bien l'APK complètement lancé.

Méthodes utilisées

Le moteur d'analyse statique est une application implémentée avec l'environnement d'exécution Node.js. Je me suis donc servie à la fois de la documentation de Node.js ainsi que de plusieurs librairies JavaScript comme Lodash ¹¹ ou bluebird ¹².

10. Android Package

11. <https://lodash.com>

12. <http://bluebirdjs.com/docs/getting-started.html>

Lodash est une librairie permettant de simplifier la gestion des tableaux, chaînes de caractères, objets en JavaScript. Elle fournit des méthodes pour itérer sur ces éléments, manipuler et tester les valeurs, ainsi que pour créer des fonctions composées.

La librairie bluebird propose une API pour l'utilisation de promesses. Les promesses sont utilisées en JavaScript pour traiter des actions de façon asynchrone. Elles représentent des opérations asynchrones qui n'ont pas encore été complétées, mais qui sont attendues dans le futur.

Pour travailler sur cette évolution du tableau de bord, une branche git spécifique a été créée sur tous les projets impactés. De plus, nous avons utilisé une machine virtuelle dédiée pour tester nos développements dans un environnement similaire à un environnement de production.

Travail effectué

Une des étapes de ce travail a été d'ajouter une étape de référence dans les tests lancés par le *TestRunner*. J'ai pour cela implémenté l'ajout d'une nouvelle mesure dans les modules qui lancent les tests (un module pour les tests sur PC et un module pour les tests sur mobile). Cette nouvelle mesure s'effectue :

- après le lancement du navigateur pour les tests URL (mesure de la consommation de la plateforme PC ou mobile avec le navigateur «vide»);
- avant le lancement de l'APK pour les tests APK (mesure de la consommation de la plateforme mobile).

L'ensemble des règles (d'analyse statique et dynamique) utilisées par GREENSPECTOR sont stockées en base. Pour permettre au moteur d'analyse d'utiliser de nouvelles règles, j'ai donc complété les scripts SQL d'ajout des règles dans les tables de la base de données.

À la fin de l'exécution du *TestRunner*, une première route est utilisée pour envoyer les résultats des mesures. Puis une autre route est appelée pour lancer le moteur d'analyse dynamique sur l'application en question.

L'algorithme principal du moteur d'analyse dynamique consiste à parcourir la liste des règles dynamiques stockées en base pour vérifier si la règle est respectée ou non. Selon les règles, cette vérification peut être :

- une comparaison des mesures associées à une étape de la mesure entre la version actuelle et la version précédente ;
- une comparaison entre les métriques attendues et les métriques effectivement mesurées ;
- une comparaison entre les mesures associées à l'étape *load* ou *idle* et les mesures de l'étape de référence ;
- une analyse simple d'une métrique.

Pour chaque règle, des seuils permettent d'attribuer un score à la règle. Ce score est proportionnel au «degré de violation» de la règle, et est donc maximal si la règle est respectée.

Par exemple, l'exemple ci-dessous implémente l'algorithme qui vérifie si les métriques attendues sont bien mesurées.

```

var lstMetricsForStep =
  _.keys(runContext.lstTestMetricGroupByStepMetricForCurrentAudit[completeStepName]);
// metriques qui sont absentes
var metricsOffenders = _.difference(rule.metricsToVerify, lstMetricsForStep);
logger.debug('metrics not verified are ' + metricsOffenders);

var nbMetricsToVerify = rule.metricsToVerify.length;
var nbMetricsNonVerified = metricsOffenders.length;
var ecoScore =
  (nbMetricsToVerify - nbMetricsNonVerified) / nbMetricsToVerify * rule.maxPoints;
ecoScore = _.round(ecoScore, 2);
var priority;
if (nbMetricsNonVerified > 0) {
  priority = 1;
} else {
  // all metrics are measured
  priority = 4;
}

```

Le principe ici est de comparer la liste des métriques mesurées avec la liste des métriques attendues en utilisant la méthode *difference* de la librairie *Lodash*. Puis, l'écopoint est calculé de façon proportionnelle au rapport entre nombre de métriques vérifiées et le nombre de métriques attendues. La priorité de la règle est notée sur 4 niveaux (niveaux 1, 2 et 3 pour les règles non respectées, niveau 4 pour les règles respectées).

L'ensemble des informations nécessaires pour chaque règle (*rule.metricsToVerify*, *rule.maxPoints* dans l'exemple précédent) est stocké dans un fichier JSON. Ce fichier contient un objet par domaine d'analyse et est chargé au démarrage du moteur d'analyse dynamique. Chaque objet est composé d'un tableau de règles. Chaque règle est un objet JavaScript associant des valeurs à des clés (identifiant dans la base de données, nom de la règle, nombre maximal de point, nom de l'étape du test associée, etc). Pour que l'accès aux informations liées à une règle soit plus simple, une fonction permet de transformer chaque élément du tableau de règles en un objet avec un unique attribut, dont la clé est l'identifiant dans la base de données, et la valeur l'ensemble des autres informations. Cette conversion est illustrée ci-dessous.

```

function loadMapRulesDescription(objectFromJSON) {
  return _.mapValues(objectFromJSON, function(rulesForDomain) {
    return rulesForDomain.reduce(function(map, obj) {
      map[obj.id] = obj;
      return map;
    }, {});
  });
}

```

Enfin, j'ai également travaillé sur les descriptions des règles dynamiques du domaine ressources, telles qu'elles sont présentées à l'utilisateur sur l'interface web. L'objectif était d'expliquer de façon claire :

- ce que la règle cherche à vérifier ;

- le principe de l'algorithme associé à la règle ;
- où peuvent être retrouvées les métriques utilisées pour l'analyse ;
- les actions à effectuer pour améliorer l'écopcore d'une application.

L'annexe B présente un exemple de description.

Résultats

L'annexe D présente un exemple d'utilisation du *TestRunner* sur une application. Le projet analysé est *PatriMath*, un des projets de groupe de l'option informatique de l'École Centrale de Nantes pour l'année 2015-2016 auquel j'ai participé.

Les écoscores pour le domaine Réseau et Ressources client (figure 23) sont issus des mesures effectuées pendant les tests standardisés lancés par le *TestRunner* sur PC et mobile. L'écopcore du domaine Code a été calculé après avoir lancé une analyse statique sur le code source.

La figure 24 montre le détails de l'analyse du domaine Réseau. La règle *Gérer les entêtes de cache* n'est par exemple pas vérifiée, car plusieurs fichiers n'ont pas d'entête de cache configuré.

La figure 25 permet de visualiser les ressources consommées pendant les différents tests (sur mobile ou PC). Par exemple, on vérifie bien que la consommation en énergie sur PC est plus importante pendant l'étape de chargement que l'étape *idle*, et que la consommation d'énergie pour ces 2 étapes est à chaque fois plus élevée que la valeur de référence.

8. Participation à la mise en place d'une procédure de suivi qualité

Afin d'améliorer la qualité de l'outil GREENSPECTOR, une procédure de suivi qualité plus avancée que celle déjà existante a été mise en place au moment de la qualification de la version 1.7.0. Cette procédure est nécessaire pour :

- évaluer la qualité du produit ;
- séparer les plateformes de test et de développement ;
- identifier de façon formelle les anomalies et permettre de les tracer ;
- vérifier la non-régression des fonctionnalités.

J'ai ainsi pu participé à la fois à la rédaction de tests fonctionnels ainsi qu'à leur exécution.

Spécifications

Pour chaque composant fonctionnel de l'outil, des tests de base et des tests avancés sont définis. Ces tests sont associés à des campagnes de test qui permettent de suivre l'avancement de l'exécution des tests (date et environnement d'exécution, succès ou non du test, commentaires éventuels...). Les campagnes de test sont effectuées à la fin d'une phase de développement.

Méthodes utilisées

Nous avons utilisé l'outil Zephyr intégré à notre outil de gestion des tâches (JIRA).

Travail effectué

J'ai plus particulièrement travaillé sur la rédaction des tests pour les composants fonctionnels sur lesquels j'ai davantage travaillé :

- le plugin pour JMeter ;
- le module indépendant *Meter*.

Résultats

La figure 20 propose un aperçu des étapes du test de base du plugin pour JMeter. Pour chaque étape sont précisées les données nécessaires ainsi que les résultats attendus.

Test Details				
	Test Step	Test Data	Expected Result	
1	Vérifier les prérequis : - avoir au moins Java 7 installé sur la machine sur laquelle sera installé JMeter ; - posséder un compte sur un serveur GREENSPECTOR.		Les prérequis sont satisfaits.	
2	Installer Apache JMeter : télécharger l'archive et l'extraire.	http://jmeter.apache.org/download_jmeter.cgi	L'archive est extraite et contient notamment les dossiers bin et lib.	
3	Vérifier le bon fonctionnement de JMeter : exécuter le binaire jmeter (dossier bin dans l'installation de JMeter).		L'interface graphique de JMeter s'affiche.	
4	Installer le plugin GREENSPECTOR pour JMeter : copier le .jar dans le dossier lib/ext de l'installation de JMeter.		Le .jar est présent dans le dossier lib/ext de l'installation de JMeter.	

Figure 20 – Exemple de test du plugin pour JMeter

9. Tâches transverses liées au développement et à l'édition de logiciels

Ce stage m'a permis d'expérimenter et de découvrir le quotidien d'une entreprise spécialisée dans l'édition de logiciel.

9.1. Méthodes de travail

J'ai ainsi pu découvrir une méthode agile de développement nommée Kanban. Cette méthode s'inspire du système de production de Toyota *Just In Time*¹³.

Le développement d'un logiciel peut être vu comme un système prenant en entrée des demandes de fonctionnalités, et fournissant en sortie un logiciel amélioré, ainsi que le présente le schéma de la figure 21.

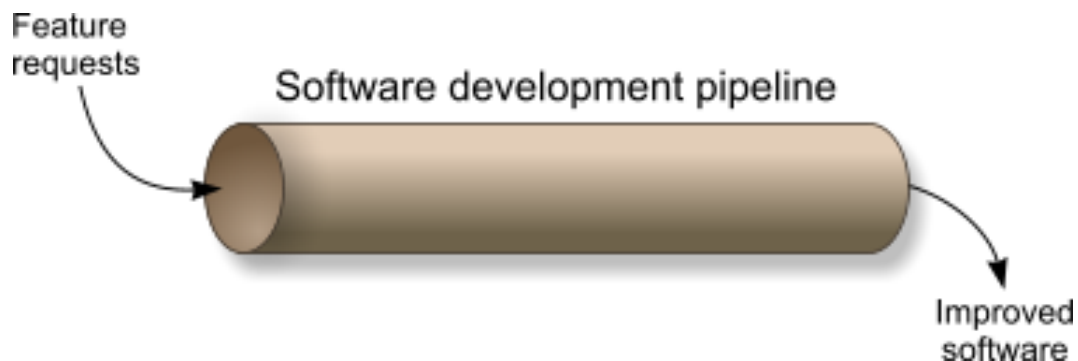


Figure 21 – Vision systémique du processus de développement d'un logiciel (source : [5])

Le processus de développement s'organise en 3 grandes phases :

- analyse du travail à effectuer ;
- développement ;
- tests.

Cependant, si l'une de ces phases est plus lente que les autres, il risque d'y avoir embouteillage : par exemple, si les développeurs développent plus vite que ce que les testeurs ont le temps de tester comme l'illustre la figure 22.

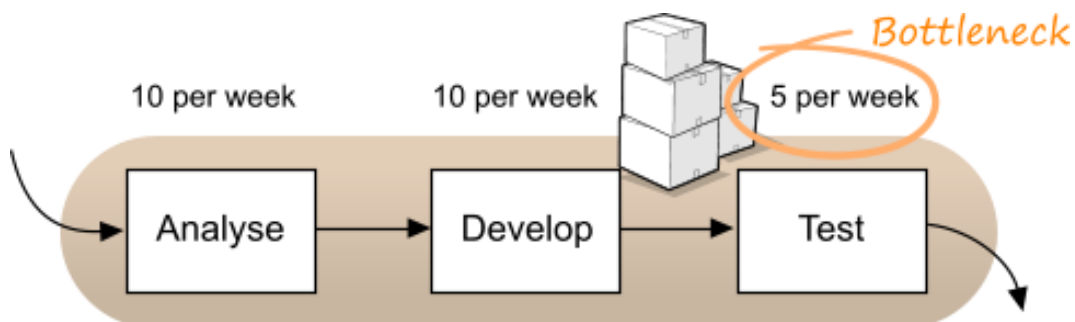


Figure 22 – Goulot d'étranglement (source : [5])

13. *Juste à temps*

Le principe de la méthode Kanban est de limiter la quantité de tâches en cours pour éviter ces embouteillages. Les tâches suivent ainsi un flux de travail défini, dont les principales étapes sont par exemple les suivantes pour GREENSPECTOR :

- à faire ;
- en cours ;
- à qualifier/intégrer ;
- fait.

Tous les matins, nous faisons un tour rapide des tâches de la journée et de l'avancement de chacun : c'est le *daily meeting* qui permet de s'organiser. Le lundi, nous faisons également une rétrospective sur la semaine précédente (l'objectif est ici d'identifier ce qu'on peut améliorer).

Le vendredi est aussi une journée plus particulière que les autres, car c'est à ce moment que nous faisons les démonstrations internes sur les dernières avancées (nouvelles fonctionnalités, améliorations de fonctionnalités existantes, etc). Ces temps de démonstration permettent d'avoir un retour sur l'outil.

9.2. Liens avec les clients

J'ai aussi eu l'occasion de participer à une intervention chez un client :

- installation du plugin GREENSPECTOR pour Apache JMeter ;
- formation du client sur l'installation et l'utilisation.

Cette intervention était très intéressante car discuter avec un client permet de prendre du recul sur le produit et aide à davantage se rendre compte de l'expérience utilisateur. C'était également très gratifiant d'installer chez un client un module pour lequel j'avais contribué au développement.

Un autre lien avec les clients, plus indirect cependant, concerne les bugs remontés. J'ai pu me rendre compte pendant ce stage qu'il n'est pas toujours évident de reproduire les bugs remontés car les environnements de développement sont souvent différents.

9.3. Documentation utilisateur et documentation interne

Une autre tâche à laquelle j'ai participé concerne la documentation utilisateur de l'outil, ainsi que la documentation interne. En parallèle des tâches de développement pour faire évoluer les fonctionnalités de l'outil, il faut mettre à jour la documentation fournie à l'utilisateur. La documentation interne (gérée dans un wiki) se réfère aux spécifications des modules, aux explications sur les calculs de métriques et d'écocores.

9.4. Déploiement et intégration continue

Ce stage m'a également permis de découvrir concrètement l'intégration continue et le déploiement sur des environnements de production.

Chaque projet versionné sous Git contient un script de déploiement vers un dépôt interne. Ce dépôt contient toutes les versions de tous les artefacts déployés (fichiers .class pour Java, archives JAR, fichiers binaires, etc).

Le déploiement des modules se fait ensuite à l'aide de playbooks Ansible. Des fichiers descripteurs des environnements permettent de gérer les versions des modules devant

être déployés sur les différents serveurs. Après que les artefacts sont téléchargés à partir du dépôt interne mentionné précédemment, le déploiement d'un module sur un serveur suit les étapes suivantes :

- envoi des artefacts vers une bibliothèque contenant les versions;
- mise à jour de la configuration ;
- mise à jour des liens actifs vers les modules de la version déployée ;
- redémarrage des services.

10. Conclusion

10.1. Bilan du travail réalisé

L'objectif de ce stage était de participer au développement de fonctionnalités pour l'outil GREENSPECTOR. Dans ce contexte, j'ai pu contribuer à l'amélioration de l'outil sur différents aspects :

- la mise en place d'un environnement de test pour des règles de bonnes pratiques PHP ;
- l'évolution de l'outil *Meter* permettant d'effectuer des mesures dynamiques d'une application ;
- l'évolution du tableau de bord utilisateur par l'ajout de règles dynamiques sur les ressources consommées ;
- la mise en place d'une procédure de suivi qualité.

10.2. Difficultés rencontrées

J'ai eu quelques difficultés pour prendre en main l'architecture globale des modules et comprendre le fonctionnement de toutes les briques de l'outil. La multiplicité des projets, langages, outils utilisés n'était pas toujours évidente à appréhender. Cependant, les différentes tâches sur lesquelles j'ai travaillées m'ont permis de découvrir pas à pas l'écosystème de l'outil.

Une autre difficulté rencontrée concerne le fait de ne pas avoir eu de sujet de stage détaillé et de vision claire dès le début sur les missions attendues. Je n'ai donc pas hésité à aller demander de l'aide quand le besoin se faisait sentir, ou à faire part du fait que je ne savais pas exactement sur quelle tâche travailler. À chaque fois, une personne de l'équipe a pu m'aider ou m'aiguiller. Avec le recul, je me rends compte que j'ai ainsi pu m'intégrer au fur et à mesure à l'équipe et à une méthode de développement agile, sans être « mise de côté » en travaillant sur un sujet annexe.

10.3. Perspectives

La liste des perspectives d'évolution de l'outil GREENSPECTOR est encore très longue et je suis très heureuse d'avoir la possibilité d'intégrer l'équipe à la suite de mon stage pour continuer à participer au développement de l'outil.

Ce stage m'a également permis de confirmer la nécessité pour moi de pouvoir concilier mon activité professionnelle avec des valeurs personnelles d'écologie et de réflexion sur notre impact environnemental. Travailler dans le milieu du Green IT me permet aujourd'hui d'y répondre, car il me permet d'être active dans un domaine qui me plaît (l'informatique) tout en donnant du sens à mon métier d'ingénieur.

10.4. Bilan personnel

D'un point de vue technique, ce stage m'a permis de monter en compétences sur de nombreux aspects. J'ai eu l'occasion de découvrir un nouveau langage de programmation (Go), augmenter ma connaissance de langages déjà connus (Java, JavaScript). Le fait de participer à toutes les étapes de l'édition d'un logiciel (spécifications, développement, tests, intégration, déploiement) m'a aidé à mieux appréhender et comprendre le cycle

de vie d'un logiciel. Je me sens ainsi un peu plus à l'aise dans le vaste écosystème des technologies du numérique, même s'il me reste encore de nombreuses choses à apprendre et découvrir.

D'un point de vue plus humain et relationnel, j'ai énormément apprécié travailler dans une équipe à taille humaine (une quinzaine de personnes), où entraide, écoute et bonne ambiance sont tous les jours au rendez-vous.

A. Description de la règle «Préférer les regex PCRE aux regex POSIX»

A.1. Problème

Depuis PHP 5.3.0, l'extension des regex POSIX est obsolète et a été supprimée dans PHP 7.0.0. Une alternative possible est d'utiliser l'extension PCRE (compatible PERL) qui est souvent plus rapide.

A.2. Description

Par exemple, trouver l'occurrence d'un motif dans une chaîne donne des résultats différents avec les fonctions POSIX ou PCRE :

- POSIX utilise la sous-chaîne correspondante la plus longue comme résultat ;
- PCRE utilise la première sous-chaîne correspondante comme résultat.

Cette différence peut avoir des effets sur le temps de recherche du motif, et donc sur la performance.

A.3. Solution

Préférer les fonctions des regex PCRE aux fonctions des regex POSIX.

Remplacements de fonctions :

POSIX	PCRE
<code>ereg_replace()</code>	<code>preg_replace()</code>
<code>ereg()</code>	<code>preg_match()</code>
<code>eregi_replace()</code>	<code>preg_replace()</code>
<code>eregi()</code>	<code>preg_match()</code>
<code>split()</code>	<code>preg_split()</code>
<code>spliti()</code>	<code>preg_split()</code>
<code>sql_regcase()</code>	pas d'équivalent

A.4. Référence

<http://php.net/manual/en/reference.pcre.pattern.posix.php>

B. Description d'une règle dynamique du domaine

ressources : «Minimiser l'impact en CPU plateforme sur mobile pendant l'étape idle»

B.1. Problème

Il faut éviter les applications ayant un impact en consommation de ressources trop élevé, c'est-à-dire lorsque la consommation de ressources de la plateforme avec l'application est trop importante par rapport à la valeur de référence pour la plateforme seule.

B.2. Description

Un logiciel fonctionne sur un socle (OS, middleware...) qui consomme une certaine quantité de ressources. L'impact d'un logiciel est la surconsommation par rapport à cette consommation de base (ou de référence). Un impact trop important va d'une part consommer trop de ressources, et d'autre part empêcher d'autres applications d'utiliser ces ressources.

Un logiciel est le plus souvent dans un état idle (lorsqu'un utilisateur lit une page web par exemple). Il est donc d'autant plus nécessaire de maîtriser la consommation dans cet état. De plus, en limitant la quantité de ressources consommées, d'autres applications peuvent utiliser plus de ressources pendant l'inactivité du logiciel (partage des ressources équitable).

Le résultat affiché sur le tableau de bord (en déroulant la règle) correspond à l'évolution de la métrique CPU plateforme sur mobile pour l'application pendant l'étape idle par rapport à l'étape de référence. Plus cette évolution est faible, plus l'écopcore augmente. Les valeurs des métriques utilisées pour calculer cette évolution peuvent se retrouver dans la section TESTS DETAILS du tableau de bord, ainsi que dans l'onglet *Meter* (où chacune des étapes de la mesure : référence, chargement, idle, ...) sont détaillées.

Exemple pour une application web (test d'une URL) avec la métrique énergie plateforme :

- mesure de l'application pendant l'étape de référence : 1,25 mWh ;
- mesure de l'application pendant l'étape de chargement : 2,03 mWh.

L'impact de l'application pendant l'étape de chargement est donc +62% (évolution par rapport à l'étape de référence).

B.3. Solution

Lancer des tests dynamiques sur mobile pour avoir des mesures d'une application, et vérifier que l'impact en CPU plateforme pendant l'étape idle n'est pas trop important.

C. Utilisation de l'API Meter Java dans un test Selenium

```
package com.greenspector.meter.api.examples;

import static org.junit.Assert.fail;
import java.util.concurrent.TimeUnit;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.MarionetteDriver;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;

import com.greenspector.meter.api.GreenspectorMeterAPI;

public class SeleniumTestCase {
    private WebDriver driver;
    WebDriverWait wait;

    @Before
    public void setUp() throws Exception {
        driver = new MarionetteDriver();
        driver.manage().timeouts().implicitlyWait(30, TimeUnit.SECONDS);
        driver.get("https://www.youtube.com/");
        wait = new WebDriverWait(driver, 120);
        wait.until(ExpectedConditions.elementToBeClickable(By.id("search-btn")));
        GreenspectorMeterAPI.loadConfiguration("./greenspector/");
        // will measure only firefox process
        GreenspectorMeterAPI.initializeProbes(new String[] {"firefox"});
    }

    @Test
    public void test_YouTubeSearch() {
        driver.findElement(By.id("masthead-search-term")).clear();
        driver.findElement(By.id("masthead-search-term")).sendKeys("Youtube");
        if (!GreenspectorMeterAPI.startMeasure("test_Search")) {
            fail("Couldn't start measure");
        }
        driver.findElement(By.id("search-btn")).click();
        wait.until(ExpectedConditions.elementToBeClickable(By.id("search-btn")));
        try {
            Thread.sleep(4000);
        } catch (Exception ex) {
            fail("Timer interrupted");
        }
    }
}
```

```

        if (!GreenspectorMeterAPI.stopMeasure()) {
            fail("Couldn't stop measure");
        }
        if (!GreenspectorMeterAPI.startMeasure("test_Idle")) {
            fail("Couldn't start measure");
        }
        try {
            Thread.sleep(5000);
        } catch (Exception ex) {
            fail("Timer interrupted");
        }
        if (!GreenspectorMeterAPI.stopMeasure()) {
            fail("Couldn't stop measure");
        }
    }

    @After
    public void tearDown() throws Exception {
        GreenspectorMeterAPI.shutdownProbes();
        driver.quit();
    }
}

```

D. Tableau de bord : exemple d'analyse du projet PatriMath

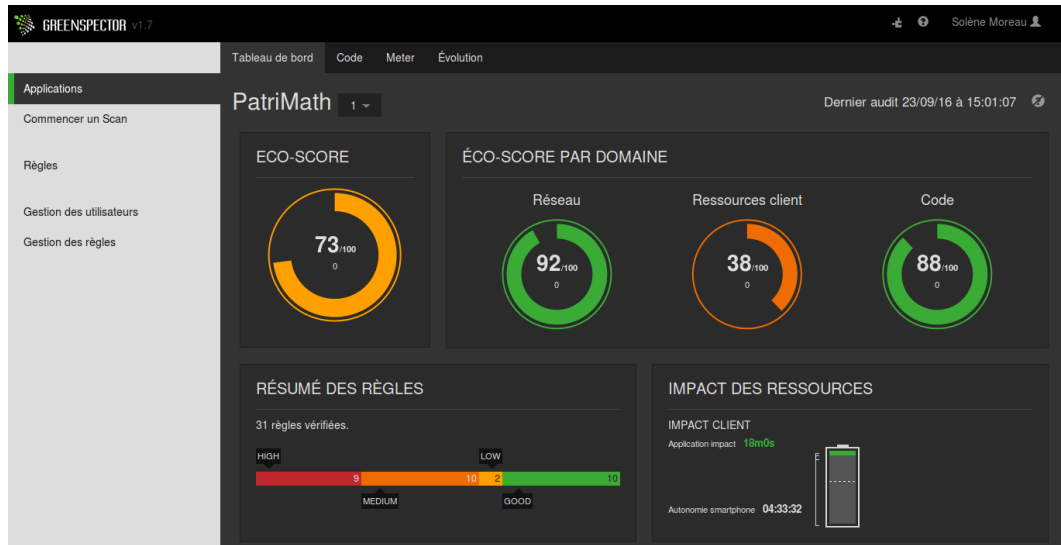


Figure 23 – Tableau de bord - vue 1

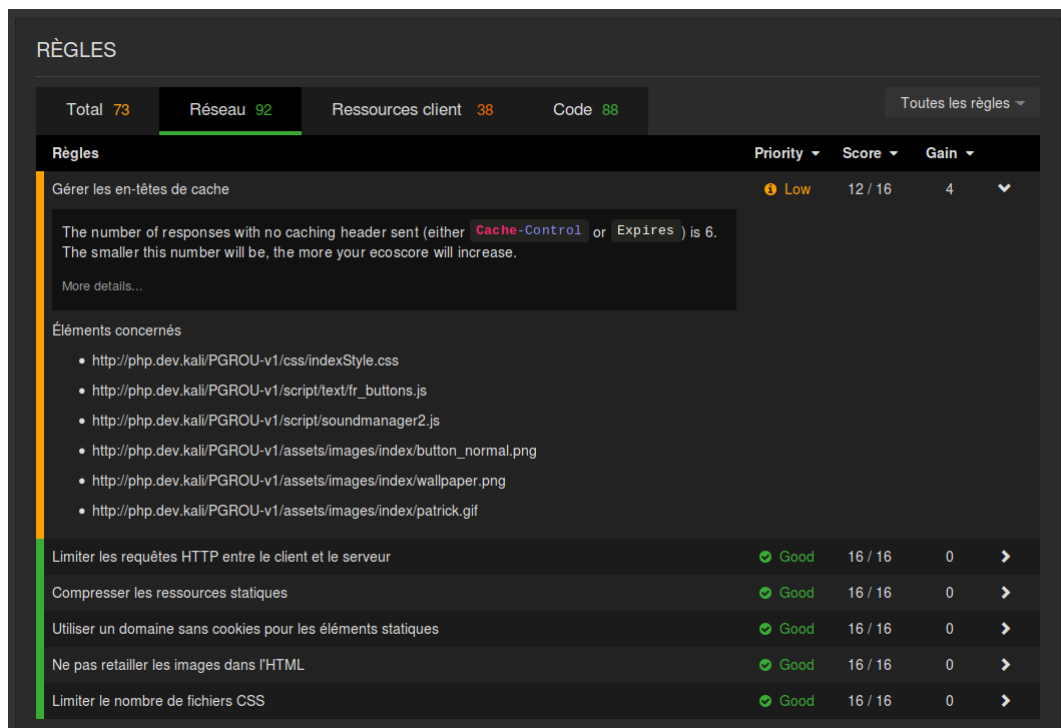


Figure 24 – Tableau de bord - vue 2

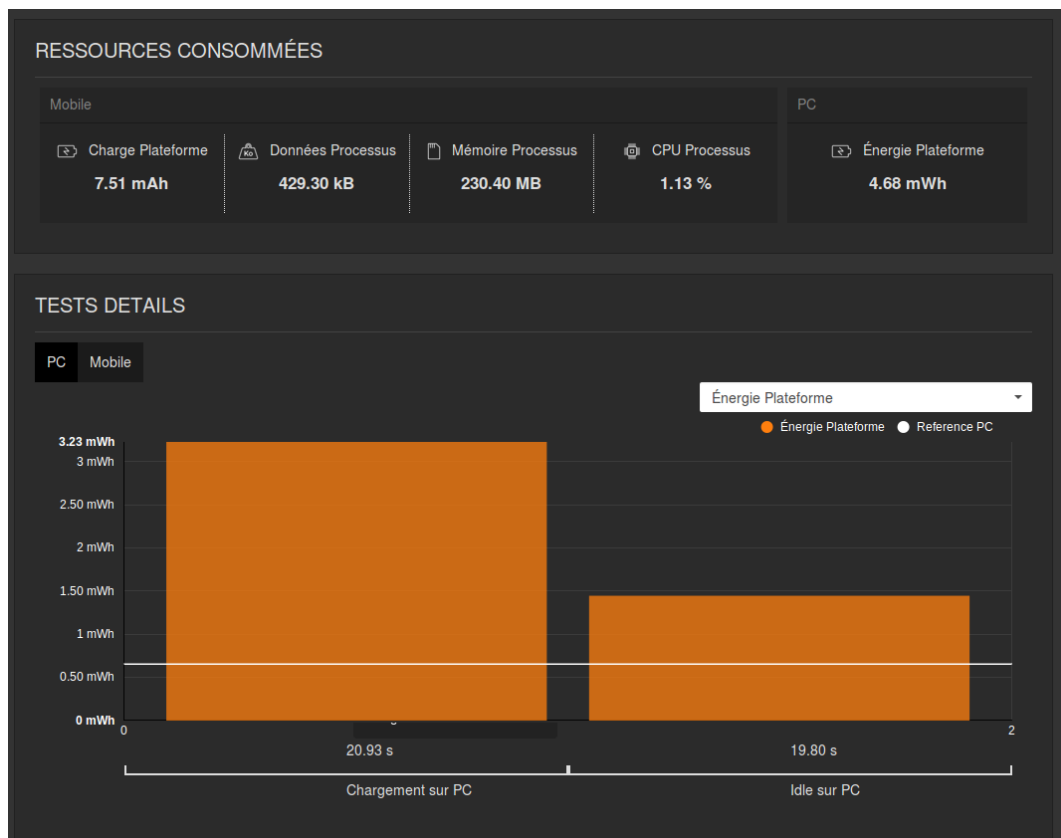


Figure 25 – Tableau de bord - vue 3

Références

- [1] Apache jmeter api. <http://jmeter.apache.org/api/index.html>. accédé le 17 septembre 2016.
- [2] Développer un plan de test avec JMeter. <http://arodrigues.developpez.com/tutoriels/java/performance/developper-plan-test-avec-jmeter/>. accédé le 22 août 2016.
- [3] Informatique durable. https://fr.wikipedia.org/wiki/Informatique_durable. accédé le 27 septembre 2016.
- [4] Turbocharging the Web with PHP7. <https://pages.zend.com/rs/zendtechnologies/images/PHP7-Performance%20Infographic.pdf>. accédé le 11 septembre 2016.
- [5] What is kanban ? <http://kanbanblog.com/explained/>. accédé le 21 septembre 2016.
- [6] ADEME. *Internet, courriels : réduire les impacts*. 2014.
- [7] Frédéric Bordage. Benchmark Green IT 2016. GreenIT.fr, 2016.
- [8] Green Code Lab. *Green Patterns*. 2012.
- [9] Gerhard Fettweis and Ernesto Zimmermann. ICT energy consumption-trends and challenges. In *Proceedings of the 11th International Symposium on Wireless Personal Multimedia Communications*, volume 2, page 6. Lapland, 2008.