



GREENSPECTOR

L'écoconception pour les
développeurs

Khalil Boulkenafet
Rapport de stage ingénieur

Étude et développement d'un dispositif de mesure physique, étude iOS pour l'outil GREENSPECTOR

Tuteur école :
Jean-Yves MARTIN

Encadrants entreprise :
Thierry LEBOUcq
Olivier PHILIPPOT



Remerciements

Je tiens à remercier ABDOU.

Table des matières

Remerciements	1
Table des matières	2
Table des figures	4
Introduction	5
Présentation de l'entreprise	6
Cadre général	7
1 Éco-conception logicielle	7
2 Présentation de l'outil <i>Greenspector</i>	8
2.1 Architecture	8
2.2 Intelligence	10
3 Travail demandé	12
3.1 Dispositif de mesure physique de consommation d'énergie	12
3.2 Étude et développement iOS	12
3.3 Tâches supplémentaires	12
Dispositif de mesure physique de consommation d'énergie sur mobiles Android	13
1 Introduction : à propos de la mesure logicielle de consommation énergétique	13
2 Cadre	14
2.1 Présentation du prototype de sonde physique	14
2.2 Projets impactés, langages de programmation	16
2.3 Spécification	16
3 Implémentation du module de mesure physique	17
3.1 <code>go-testapi</code>	17
3.2 <code>testrunner</code>	21
4 Adaptation du dispositif	22
5 Analyse des résultats	22
Étude de faisabilité iOS	23
1 Introduction	23
2 Cadre	23
2.1 Appareil	23
2.2 Sonde physique	23
2.3 Technologies utilisées	23
3 Tests automatisés	23
3.1 Premiers essais : <code>libimobiledevice</code> et <code>ios-deploy</code>	24

3.2	Vers la solution : <code>WebDriverAgent</code>	25
3.3	Écriture de scripts de benchmark	27
4	Implémentation dans <code>go-testapi</code> et <code>testrunner</code>	28
4.1	<code>go-testapi</code>	28
4.2	<code>testrunner</code>	31
5	Ajout de métriques supplémentaires	32
Tâches supplémentaires		33
1	Benchmark web Android : mesure de référence	33
2	Calibration SAMSUNG GALAXY S9	34
A Tableau de bord : exemple d'analyse du site de l'École Centrale de Nantes		35
B Intégration de la mesure physique dans <code>testrunner</code>		37
C Scripts de tests automatisés sur iOS		40
1	Benchmark web	40
2	Benchmark d'application	42
Bibliographie		44

Table des figures

1	Architecture et flux (version Cloud)	8
2	Architecture et flux (version On premises)	9
3	Arduino INA219 High Side DC Current Sensor	14
4	Arduino Uno Rev3	14
5	Sonde physique : câblage des composants électroniques	15
6	Sonde physique : circuit pour la mesure d'un Raspberry Pi	15
7	Déclaration de la structure <code>hardwareMeasureLauncher</code>	17
8	Déclaration de la structure <code>APKJobExecutor</code>	18
9	Méthode <code>InitAPKJobExecutor</code>	18
10	Intégration de la mesure physique dans la méthode <code>RunIterations</code>	19
11	Intégration des résultats de la mesure physique	20
12	Déclaration de la structure <code>HardwareMeasureLauncherConfig</code>	20
13	Instanciation de <code>GetTimeDelta</code> dans le cas Android	21
14	Instanciation de la configuration du port série dans le programme principal du <code>testrunner</code>	21
15	Connecteurs batterie	22
16	Exemples d'utilisation de <code>ios-deploy</code>	24
17	<code>WebDriverAgent</code> : affichage de l'adresse du serveur dans la console	26
18	Exemples d'utilisation de <code>WebDriverAgent</code> grâce à <code>cURL</code>	27
19	Méthodes de l'interface <code>Device</code>	29
20	Déclaration des structures <code>IosDeviceConfig</code> et <code>IosDevice</code>	29
21	Déclaration de l'interface <code>JobLauncher</code>	30
22	Une méthode de test unitaire pour <code>IosJobLauncher</code>	31
23	Page web blanche pour la mesure de référence	34
A.1	Tableau de bord. Vue 1	35
A.2	Tableau de bord. Vue 2	36
A.3	Tableau de bord. Vue 3	36

Introduction

Les technologies de l'information et des communications (TIC) ont permis de réduire l'impact environnemental de nombreux secteurs d'activités et cela de diverses manières (dématérialisation, smart grids, gestion des ressources et de la production...). L'usage croissant de ces technologies implique cependant une empreinte écologique conséquente [1], entre la consommation de ressources des équipements (matières premières, consommation d'énergie¹, recyclage difficile et coûteux) et la pollution (gaz à effet de serre, pollution liée à l'extraction des métaux). [3]

Dans le contexte actuel de réchauffement climatique et de raréfaction des énergies fossiles et métaux rares, la réduction des impacts environnementaux des appareils et services du numérique représente un enjeu de taille.

La *conception responsable* (ou éco-conception) des services numériques est un des leviers d'action possibles, et présente un potentiel très important. Elle encourage la modération et la sobriété dans la conception des TIC. Selon le Benchmark Green IT 2017 [4], elle permettrait d'utiliser « de l'ordre de 2 à 100 fois moins de ressources informatiques [...] à tous les niveaux du système d'information ».

Greenspector est la première solution outillée pour l'éco-conception des logiciels. Intégrée dans les outils du développeur, elle permet aux DSI de mesurer et contrôler la consommation de ressources des logiciels qu'elles produisent ou réceptionnent. Ces informations rendent alors possibles des gains d'autonomie des appareils mobiles et objets connectés, des gains de performance des applications et des économies dans les datacenters.

Ce stage chez GREENSPECTOR a permis d'aborder plusieurs sujets : développement d'une fonctionnalité de l'outil, mise en place de tests unitaires, comparaisons des mesures physique et logicielle de consommation d'énergie, investigations pour étendre la portée de l'outil. J'ai suivi une démarche générale de recherche et développement, tout en étant intégré dans le quotidien de l'équipe.

1. En 2015, en France, 12% de la consommation électrique du pays provient du numérique, d'après une étude de l'association négaWatt. [2]

Présentation de l'entreprise

En novembre 2010, Thierry LEBOUQC et Martin DARGENT créent KALITERRE, entreprise spécialisée dans les missions de conseil et formation en RSE et Green IT¹, notamment auprès de l'ADEME², Nantes Métropole, et le Ministère de la Culture et la Communication. Martin DARGENT quitte KALITERRE en 2011 — il en reste actionnaire et partenaire — et deux nouveaux associés, Thomas CORVAISIER et Olivier PHILIPPOT, rejoignent l'entreprise en tant que co-gérants en 2013.

Entre 2010 et 2014, KALITERRE finance sur fonds propres ses projets de R&D sur l'éco-conception, en participant à divers projets de recherche français et européens. Ces projets débouchent sur la création d'une suite logicielle, *Greenspector*, permettant de mesurer la consommation énergétique des applications et détecter les objets consommateurs dans le code source des logiciels. Des propositions de corrections pour les développeurs complètent ces mesures et analyses.

En 2016 l'entreprise devient GREENSPECTOR et lance la commercialisation de sa solution, soutenue à hauteur de 300 000 € par le fonds d'investissement breton NESTADIO CAPITAL. *Greenspector* s'adresse principalement aux compagnies de services du numériques, aux acteurs de l'industrie mobile ainsi qu'aux grands comptes qui produisent ou achètent du logiciel.

Partenariats et distinctions

- *Greenspector* est la première solution logicielle labellisée par le European Code of Conduct for Datacenters.
- GREENSPECTOR est partenaire du projet WEBENERGYARCHIVE³, une première « étiquette énergétique » pour les sites webs.
- L'entreprise a fondé le GREEN CODE LAB, la communauté française des logiciels éco-conçus⁴.

1. La démarche *Green IT* vise à réduire l'empreinte écologique, économique et sociale des technologies de l'information et de la communication. Elle est rattachée aux Directions des Systèmes d'Information. [5]

2. ADEME : Agence de l'environnement et de la maîtrise de l'énergie.

3. <https://wea.greencodelab.org>

4. Le GREEN CODE LAB est notamment auteur du livre *Green Patterns* [6], manuel d'éco-conception logicielle à destination des développeurs

Cadre général

1 Éco-conception logicielle

En dépit de leur caractère immatériel, les logiciels ont besoin d'équipements informatiques pour fonctionner (ordinateurs, écrans, serveurs...). L'analyse du cycle de vie (ACV) de ces équipements révèle que toutes les étapes présentent une empreinte écologique conséquente. Indirectement, les logiciels ont donc un impact non négligeable.

Ils représentent même une des principales causes d'obsolescence des équipements. En effet, les besoins des logiciels en ressources matérielles (mémoire vive, puissance, espace disque) ne cessent d'augmenter¹. Contrôler et diminuer les besoins en ressources des logiciels constitue la première étape naturelle d'une réduction efficace de l'empreinte écologique des équipements matériels.

L'éco-conception appliquée aux logiciels est une démarche d'optimisation des logiciels dès leur conception, dans le but de :

- diminuer la consommation énergétique du logiciel pendant sa phase d'utilisation
- améliorer la performance des applications et sites web ainsi que l'autonomie des appareils mobiles pour une meilleure expérience utilisateur
- allonger la durée de vie des équipements informatiques

Elle consiste dans l'application de bonnes pratiques de développement, et cherche l'équilibre entre diminution du besoin en ressources et maintien de la performance tout en répondant au besoin des utilisateurs. Quelques exemples de bonnes pratiques :

- optimiser le code source pour réduire les ressources nécessaires
- évaluer et comparer les bibliothèques
- optimiser les fonctionnalités en se débarrassant du superflu
- augmenter la scalabilité des applications et du matériel
- mesurer précisément la consommation des applications et sites web
- optimiser l'architecture

Exemples d'application Un audit d'une application Android réalisé par GREENSPECTOR a conduit des corrections dans le code source, permettant un **gain de 69% sur la consommation énergétique** de l'application et une **augmentation de 8h d'autonomie** du mobile.

L'utilisation de l'outil *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**.

1. Par exemple, chaque version de la paire Windows-Office requiert le double de ressources par rapport à la version précédente. Ainsi, écrire un simple texte avec Windows 7 et Office 2010 requiert 70 fois plus de mémoire vive qu'avec Windows 98 et Office 97 !

2 Présentation de l'outil *Greenspector*

L'objectif de *Greenspector* est de mettre à disposition des développeurs en informatique un outil pertinent, facile d'utilisation et intégré dans leurs environnements, pour les aider à coder des applications et sites webs optimisés.

La description suivante concerne la version 2.0.0 sortie au cours de mon stage en juin 2018.

2.1 Architecture

Greenspector permet de benchmarker une application par une analyse dynamique pendant son fonctionnement, grâce à des sondes mesurant l'énergie, la mémoire, le trafic réseau, etc...

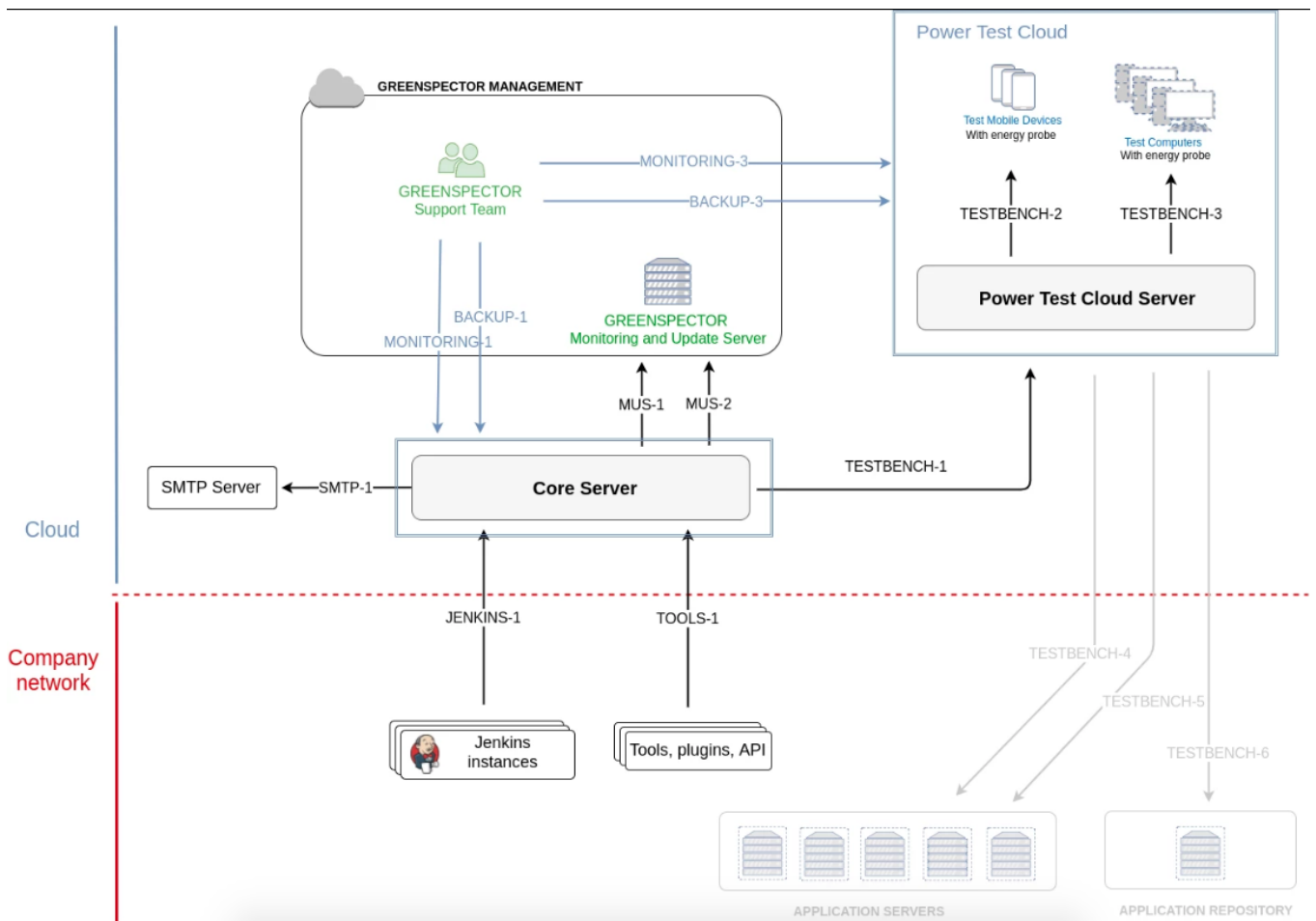


Figure 1 – Architecture et flux (version Cloud)

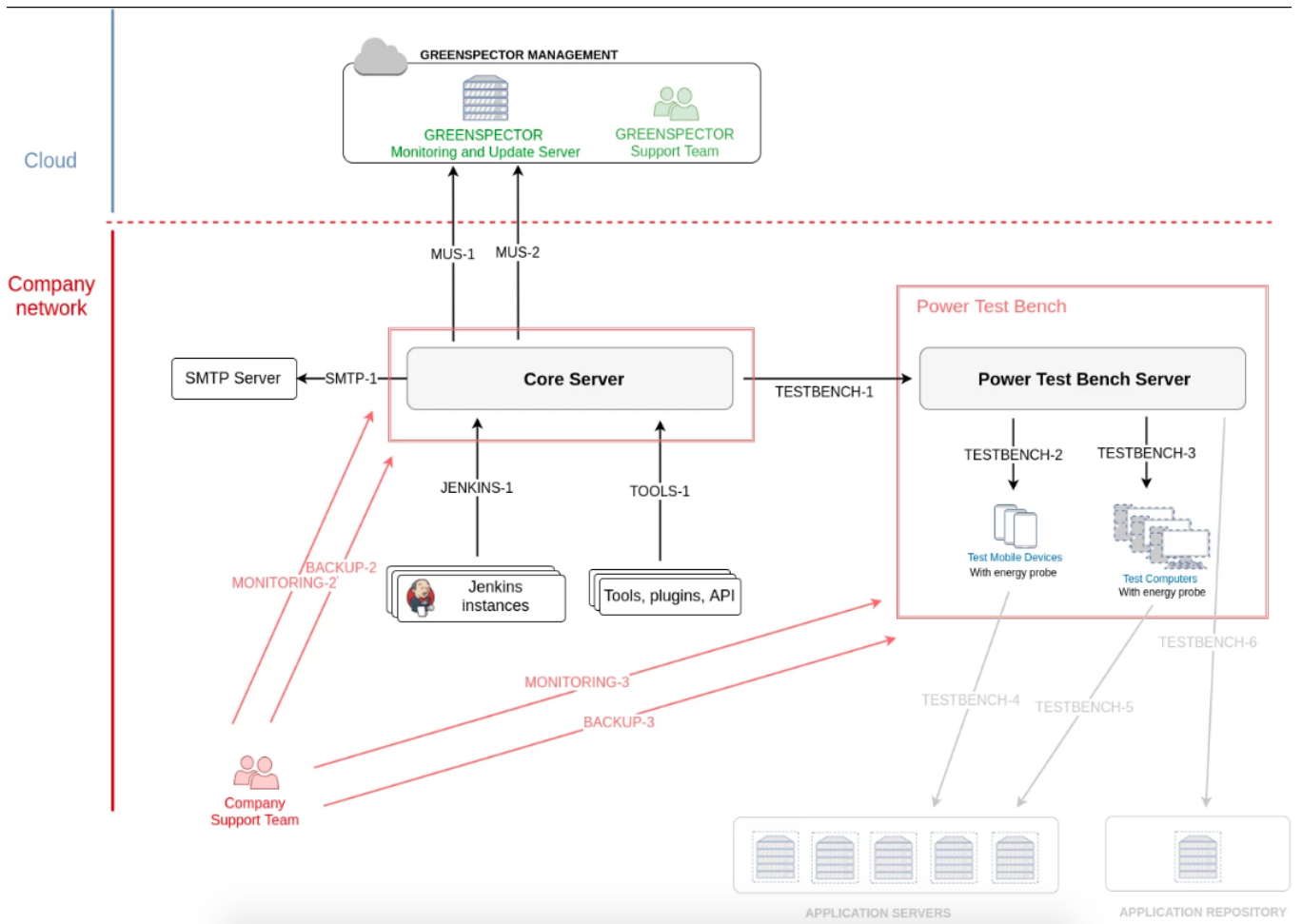


Figure 2 – Architecture et flux (version On premises)

Les figures 1 et 2 présentent un schéma de l'architecture et des flux de données pour une utilisation **cloud** ou **on premises**².

Le serveur *Greenspector* (*Core Server*) est au coeur de cette architecture ; il détient l'intelligence de l'outil. Le Core Server est relié à un serveur de supervision (*Monitoring Server*) qui veille au bon fonctionnement de l'outil. Dans le cas d'une installation on premises (i.e. lorsque le Core Server est installé chez le client), un serveur de licence (*Update Server*) vérifie la validité de la licence en cours et permet les mises à jour de *Greenspector*.

L'architecture présente également un banc de test (*Power Test Cloud Server* en cloud ou *Power Test Bench Server* on premises) comprenant les appareils sur lesquels les tests et les mesures sont effectués.

Greenspector s'intègre dans les environnements de développement, de test et d'intégration continue. Le développeur dispose sur son poste d'un accès à une interface web permettant de lancer des mesures et d'en visualiser les résultats. Au niveau des environnements de test et d'intégration continue, d'autres outils peuvent être utilisés :

- des modules ou des API accompagnées de sondes (sonde Android) pour effectuer des mesures dynamiques en s'intégrant des les outils de test comme UIAutomator, Espresso, Xamarin
- des modules greffés dans des outils d'intégration continue comme Jenkins pour automatiser le lancement de mesures

2. Il existe également une configuration **cloud with power test bench** dans laquelle le client possède seulement un banc de test dans ses locaux.

L'utilisateur dispose également de modules indépendants utilisables en ligne de commande pour lancer des mesures, exécuter des tests standards et envoyer les résultats au serveur central *Greenspector*. L'utilisation en ligne de commande facilite l'intégration au sein de processus continus et automatisés.

2.2 Intelligence

2.2.1 Référentiel de règles et de mesures

Greenspector utilise un référentiel de règles divisé en deux³ domaines principaux :

- 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, mémoire...) sur une plateforme côté client⁴ (mobile).

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'éco-conception logicielle.

On associe à ces règles des indicateurs issus de comparaisons – énergie, mémoire, performance, temps de correction – entre du code *vert* (respectant la règle) et du code *gris* (enfreignant la règle). Ces indicateurs sont classés selon 4 niveaux de criticité :

- bonne pratique
- mineur
- majeur
- critique

ce qui permet aux développeurs de prioriser les actions à mener.

2.2.2 Moteur d'analyse

L'analyse dynamique d'une application permet de distinguer les règles respectées des règles violées. Le moteur d'analyse dynamique examine les mesures effectuées pour une application en fonction du référentiel de règles dynamiques. Une règle dynamique comprend :

- **Une métrique.** Par exemple, le nombre de requêtes HTTP échangées entre client et serveur.
- **Un ensemble de seuils** permettant d'attribuer un *éco-score* à la règle. 75% du score maximal si on compte entre 5 et 10 requêtes, 100% pour moins de 5 requêtes, etc...

2.2.3 Interface utilisateur

Le **tableau de bord** (dashboard) de l'interface web *Greenspector* résume et présente à l'utilisateur l'ensemble des informations utiles issues de la mesure de son application ou site web. Les figures mentionnées dans cette section sont visibles dans l'annexe A.

Dans la première vue du dashboard (fig. A.1) on retrouve l'*éco-score* global de l'application. Il s'agit d'une note entre 0 et 100 indiquant le niveau d'éco-conception de l'application, et correspond à la moyenne des éco-scores de chaque domaine (réseau et ressources). Ces scores spécifiques sont évalués en sommant les points obtenus pour chacune des règles associées au domaine.

La section *Autonomie* fournit à l'utilisateur une estimation de l'impact de son application sur la batterie d'un smartphone (si l'application tourne en boucle sur le téléphone). Cette estimation est

3. On pouvait précédemment compter un troisième domaine, le domaine code pour l'analyse statique du code source des logiciels, applications et sites web. Mais l'analyse de code a été abandonnée par GREENSPECTOR.

4. La mesure de plateformes PC a également été abandonnée, tandis que le développement de la mesure des objets connectés est en cours.

déduite des données de mesure ainsi que de la capacité de la batterie de l'appareil utilisé pour les tests.

La figure A.2 présente un bilan des budgets⁵ définis préalablement par l'utilisateur. Les budgets correspondent à des exigences concernant différents aspects (comme l'autonomie) que les développeurs doivent respecter pour une application plus *green* ou plus performante.

La section *Test details* renseigne sur la consommation de ressources de l'application. Elle permet de visualiser différentes métriques (décharge de la plateforme, mémoire, CPU, réseau...) au cours des différentes étapes des tests effectués. Il s'agit de tests standardisés fournis par l'outil *Greenspector*. Par exemple, les tests d'une page web (ayant permis de tester le site de l'école) comportent 5 étapes : référence, chargement de la page, inactivité avec navigateur au premier plan, défilement, inactivité avec navigateur en arrière plan.

La figure A.3 présente enfin la section *Règles* qui liste chacune des règles analysées en précisant :

- le score obtenu (maximal si la règle est respectée)
- le gain potentiel de la règle (nombre de points à gagner pour arriver au score maximal)
- la priorité de correction, basée sur le gain potentiel

On dispose également d'un bilan des ressources consommées pendant le test.

L'interface utilisateur permet un suivi de l'évolution d'une application. Pour chaque valeur affichée sur le tableau de bord, la différence entre la valeur actuelle et celle de la précédente version est indiquée.

Les autres onglets (**Meter**, **Test results**, **Budgets**, **Evolution**) fournissent différentes précisions sur l'application et sa mesure. En particulier l'onglet **Meter** détaille l'ensemble des étapes de mesure à l'aide de graphiques. Tous les graphiques de mesure présentés dans ce rapport proviennent de cet onglet.

2.2.4 Tests automatisés

L'outil *Greenspector* propose 3 types de tests pour ses mesures sur mobile : les benchmarks web et apk (qui sont des tests standardisés) et custom tests.

Benchmark web Le benchmark de site internet est divisé en 7 étapes, dont 5 sont mesurées :

- Effacement des données de navigation (caches, historique)
- Lancement du navigateur et **étape de référence** : le mobile est inactif sur l'accueil du navigateur pendant 20 secondes
- Lancement et chargement du site à mesurer (**loading**) : 20 secondes
- Inactivité au premier plan (**idle foreground**) : le mobile est inactif sur la page web pendant 20 secondes
- Inactivité en arrière plan (**idle background**) : le mobile est inactif sur l'écran d'accueil (avec le navigateur en arrière plan) pendant 20 secondes
- Fermeture de l'onglet (sans cette action la page consultée se recharge lors du lancement du navigateur au test suivant) et fermeture du navigateur

5. La fonctionnalité budgets a d'ailleurs fait l'objet d'une amélioration intéressante que j'ai pu suivre au cours de mon stage.

Benchmark apk Le benchmark d'application (ou **apk**, qui désigne l'extension de l'exécutable d'une application Android) comporte 6 étapes :

- Installation de l'application grâce à **ios-deploy**
- **Étape de référence** sur l'écran d'accueil
- **Loading**
- **Idle background**
- Fermeture de l'application
- Désinstallation grâce à **ios-deploy**

Tests custom Le mode custom permet aux développeurs d'utiliser leurs propres tests automatisés, ce qui a pour avantage de pouvoir effectuer des mesures dans des cas d'utilisation plus précis qu'avec les tests de benchmark.

3 Travail demandé

GREENSPECTOR souhaitait approfondir certains sujets afin d'étendre la portée de son outil et diversifier son offre. J'ai donc été chargé de tâches de recherche et développement, dans le but de progresser le plus possible sur ces sujets avant que l'entreprise envisage un développement plus poussé et une commercialisation des fonctionnalités résultantes.

3.1 Dispositif de mesure physique de consommation d'énergie

L'équipe de développement a travaillé sur un prototype de mesure physique de consommation d'énergie pour les objets connectés. J'ai été chargé d'adapter ce prototype et d'implémenter la mesure physique sur mobiles Android (qui n'étaient mesurés jusque-là que par des sondes logicielles), puis de comparer les mesures physiques et logicielles. L'objectif de cette fonctionnalité est de proposer une solution de mesure supplémentaire aux clients, pour affiner leur vision sur la consommation de leurs applications.

3.2 Étude et développement iOS

Avant mon arrivée, les mesures de *Greenspector* ne concernaient que les mobiles Android. La mesure des appareils iOS représente pour l'entreprise une opportunité intéressante pour étendre son marché cible. J'ai eu pour mission d'explorer différentes pistes pour évaluer la possibilité de porter les fonctionnalités de l'outil aux iPhones et iPads, puis d'implémenter le benchmark dans *Greenspector*.

3.3 Tâches supplémentaires

Il m'a également été demandé de travailler sur d'autres tâches moins conséquentes. Enfin j'ai été chargé de m'intégrer au processus de développement de l'équipe (outils et méthodes agiles...).

Dispositif de mesure physique de consommation d'énergie sur mobiles Android

1 Introduction : à propos de la mesure logicielle de consommation énergétique

À mon arrivée chez GREENSPECTOR, j'ai pu prendre connaissance des moyens mis en oeuvre pour mesurer la consommation d'énergie d'une application sur les appareils Android.

Le principe de la mesure est basé sur des informations fournies par l'API Android ou accessibles dans les fichiers système de l'appareil. Chaque modèle présente des valeurs relatives à la batterie (charge restante, tension, courant...) mises à jour plus ou moins fréquemment (de quelques centaines de millisecondes à quelques dizaines de secondes). La sonde logicielle développée par GREENSPECTOR et installée sur le smartphone est chargée de récupérer et traiter la grandeur (ou métrique) la plus pertinente parmi celles qui sont accessibles.

La détermination de la grandeur la plus pertinente est effectuée dès la réception d'un nouveau modèle de téléphone, préalablement à l'installation d'une sonde. Il s'agit de trouver une métrique avec une fréquence de mise à jour satisfaisante (quelques centaines de millisecondes). Si les sondes existantes ne permettent pas le traitement, il est nécessaire d'en développer une nouvelle.

La sonde traite alors la grandeur choisie dans le but de fournir une valeur de consommation d'énergie en milliampère-heure (mAh). Il s'agit d'une unité de charge qui quantifie la capacité d'une batterie. Par exemple si la sonde utilise une mesure de courant, la décharge de la batterie correspond à l'intégration de cette intensité sur l'intervalle de temps considéré¹. Si la sonde utilise la charge de la batterie, le traitement consiste à faire la différence entre deux valeurs consécutives pour obtenir la décharge sur l'intervalle de temps considéré.

S'ensuit un processus de calibration, qui consiste à décharger progressivement la batterie du mobile pour comparer les résultats de mesure de la sonde avec la capacité de la batterie. On vérifie typiquement que pour une décharge de 10%, la sonde fournit effectivement une valeur correspondant à 10% de la charge totale de la batterie².

GREENSPECTOR souhaitait aller plus loin sur ce sujet. Pour ce faire, l'équipe de développement a mis au point un prototype de sonde physique dans le but de contrôler plus précisément la mesure de consommation d'énergie des mobiles de son banc de test. Ce sujet constitue la première tâche de mon stage.

1. On intègre simplement en multipliant l'intensité par le temps car on suppose que la tension aux bornes de la batterie est constant.

2. Dans de nombreux cas la calibration permet d'identifier un intervalle de niveau de batterie sur lequel la sonde est fiable (de 20% à 80% par exemple).

2 Cadre

2.1 Présentation du prototype de sonde physique

2.1.1 Matériel

La sonde physique développée par GREENSPECTOR est basée sur deux composants :

- Un **shunt INA219**³ (figure 3) permettant de mesurer le courant avec une précision à 1%.
- Une carte **Arduino Uno**⁴ (figure 4), programmée pour récupérer et traiter la donnée du courant mesuré par le shunt.

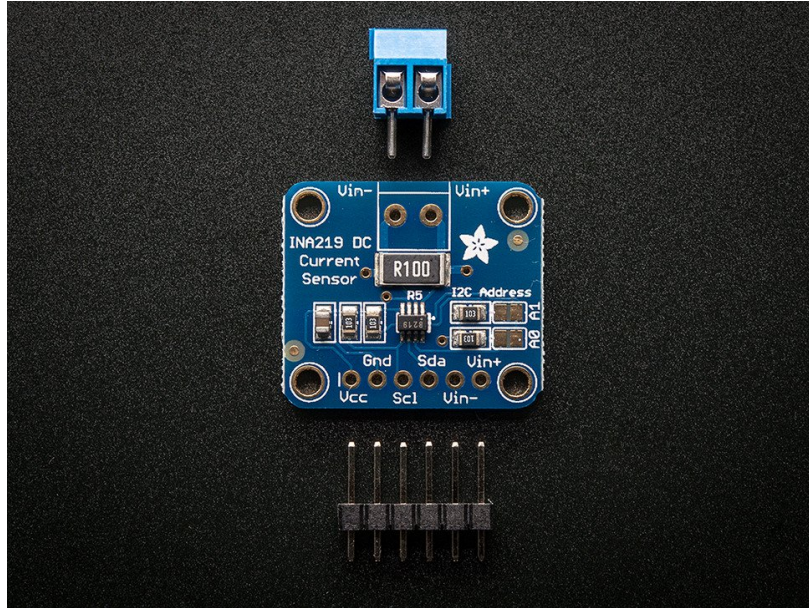


Figure 3 – *Arduino INA219 High Side DC Current Sensor*



Figure 4 – *Arduino Uno Rev3*

Le câblage est présenté sur la figure 5 :

3. <https://www.adafruit.com/product/904>

4. <https://store.arduino.cc/arduino-uno-rev3>

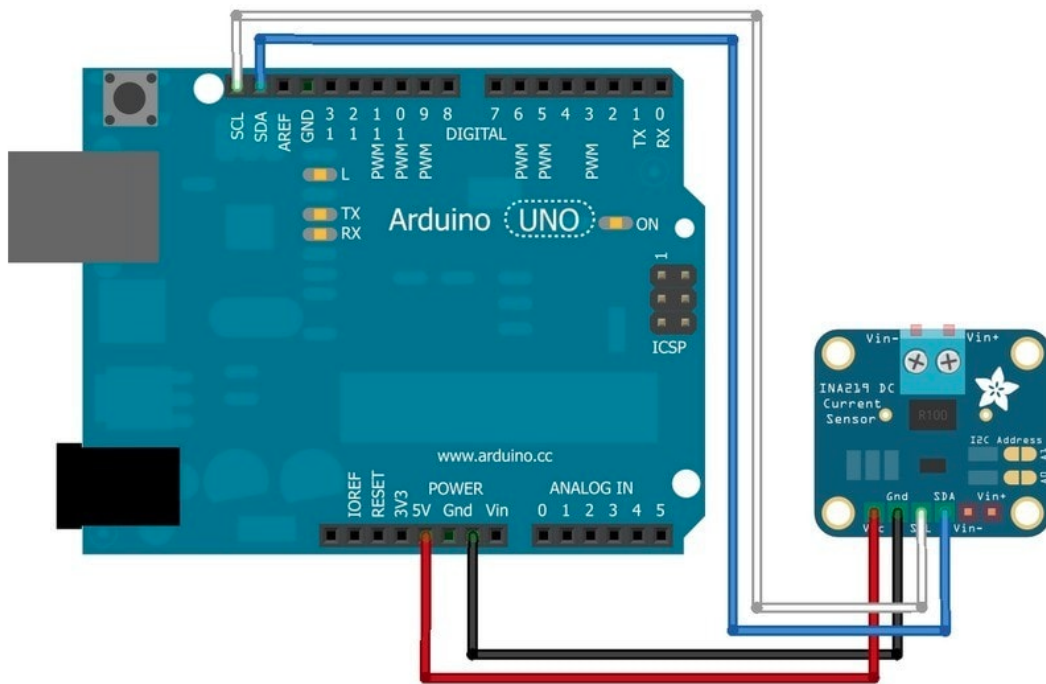


Figure 5 – Sonde physique : câblage des composants électroniques

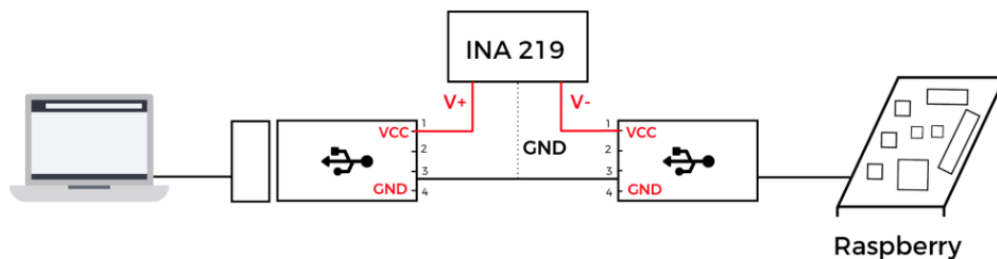


Figure 6 – Sonde physique : circuit pour la mesure d'un Raspberry Pi

La figure 6 montre le circuit complet pour la mesure physique d'un Raspberry Pi, alimenté en USB. Le shunt est inséré entre l'ordinateur et l'appareil au niveau du câble d'alimentation. Ce circuit fournit effectivement des résultats, cependant il n'est pas pertinent pour la mesure sur mobile Android, l'appareil mesuré étant en charge pendant les tests. J'ai donc dû adapter le dispositif à la mesure sur mobile ; cette étape est détaillée dans la suite du rapport.

2.1.2 Programme Arduino

Le programme de la carte **Arduino Uno**, écrit à partir de la librairie **Adafruit_INA219**, permet de lire une mesure du courant passant par le shunt à intervalle régulier (200ms). Ce courant est simplement intégré sur l'intervalle de mesure (en supposant une tension constante) pour obtenir la décharge, comme pour la mesure logicielle évoquée dans l'introduction de cette partie.

La mesure est initiée en écrivant l'octet correspondant au caractère 'a' sur le port série. Le caractère 'b' permet d'arrêter la mesure.

2.2 Projets impactés, langages de programmation

L'implémentation de la mesure physique a mis en jeu différents projets informatiques que j'ai dû prendre en main et modifier pour intégrer cette nouvelle fonctionnalité. Ces projets utilisent différents langages de programmation auxquels je me suis adapté.

2.2.1 Arduino

Le programme de la carte Arduino Uno présentée précédemment a été écrit en langage **Arduino**, très proche du **C**.

2.2.2 go-testapi

Pour intégrer la fonctionnalité de mesure physique, j'ai travaillé sur le projet **go-testapi**, une API **go** développée par GREENSPECTOR. Cette API définit les structure et méthodes permettant de mener des tests et mesures sur mobiles Android. La majeure partie de mon travail sur cette tâche concerne **go-testapi**.

2.2.3 testrunner

Le **testrunner** est un module également codé en **go** qui permet à l'utilisateur de lancer des tests standards automatisés sur mobiles Android. Le but du **testrunner** est de fournir une interface en ligne de commande (CLI) pour utiliser l'API de test **go-testapi**. Son rôle est de faire la correspondance entre des paramètres en ligne de commande (issus de fichiers de configuration), et les fonctions de l'API pour lancer des tests (standard URL, standard APK, custom APK).

2.2.4 test-bench-launcher-android

test-bench-launcher-android est une application Android écrite en **Java**. Elle est notamment lancée par **go-testapi**, avec les arguments nécessaires à son fonctionnement. De plus, elle définit les scénarios des tests standards automatisés. Les méthodes permettant l'interaction avec l'écran du téléphone testé sont également contenues dans **test-bench-launcher-android**, et utilisent le framework de test **UIAutomator**.

2.3 Spécification

Avant le début du développement de la fonctionnalité, une spécification m'a été fournie par le scrum master. La fonctionnalité doit répondre aux exigences suivantes :

- La mesure logicielle est conservée et doit être réalisée en parallèle de la mesure physique
- Pour la sonde matérielle, on utilise une nouvelle métrique d'énergie nommée **AH_HARD** (ampère-heure, hardware), par opposition à la métrique logicielle **AH_PL** (ampère-heure, plateforme)
- On conserve la plateforme **iot** pour la mesure physique hors exécution des tests (fonctionnalité existante à mon arrivée). Pour la mesure avec exécution des tests, on utilise la plateforme mobile.
- Les fichiers de configuration du test runner doivent permettre d'activer et désactiver le dispositif de mesure matérielle

3 Implémentation du module de mesure physique

Dans un premier temps, j'ai été chargé d'implémenter la mesure physique de sorte que l'on puisse effectuer une mesure pendant les tests automatisés, et remonter les résultats sur l'interface *Greenspector*.

3.1 go-testapi

3.1.1 Package hardware

À mon arrivée, les structures et méthodes permettant la mesure physique étaient contenues dans le package `iot`. Je les ai déplacées dans un package `hardware` pour plus de cohérence.

Ce package contient un autre package nommé `controller` dans lequel on trouve une structure `SerialController` qui fait le lien entre le port série (carte **Arduino**) et le projet. Cette structure permet notamment de démarrer/stopper la mesure et lire les informations contenues sur le port série. `SerialController` implémente l'interface `Controller`. On utilise une interface pour faciliter les tests automatisés. En effet, en créant un mock qui implémente l'interface on peut facilement imiter le comportement du `SerialController` et vérifier le fonctionnement du code dans lequel il intervient.

Dans le package `hardware` on retrouve également une structure `HardwareMeasureLauncher` (interface `MeasureLauncher`), présentée sur la figure 7. Elle contient notamment les informations du dispositif de mesure grâce à une instance de `Controller`, et permet de stocker (dans `hardwareMeasureResults`) les résultats lus sur le port série (méthode `startMeasure`). On reviendra sur le champ `config` dans la suite.

```
type HardwareMeasureLauncher struct {  
    config          HardwareMeasureLauncherConfig  
    controller      controller.Controller  
    stopped         chan (bool)  
    stopMeasure     chan (bool)  
    hardwareMeasureResults []common.MeasureTick  
}
```

Figure 7 – Déclaration de la structure `hardwareMeasureLauncher`

3.1.2 Implémentation du module

Pour implémenter la fonctionnalité, je me suis intéressé aux structures et méthodes qui lancent les tests sur le mobile. On trouve dans le package `android` une structure `JobLauncher`, associée aux différentes méthodes de lancement des tests. Il y a une méthode par mode : par exemple pour tester une page web, c'est la méthode `LaunchURLJob` de notre instance de `JobLauncher` qui est appelée.

Au sein de ces méthodes de test, un `executor` est initialisé et permet de mener les tests. La structure dont cet objet est l'instance dépend du type de test (par exemple, on utilise la structure `URLJobExecutor` lors d'un test de site web). Toutes ces structures "héritent"⁵ de `APKJobExecutor` (figure 8).

5. En `go`, il n'y a pas vraiment de relation d'héritage. En réalité, les instances `executor` présentent un champ qui est une instance de `APKJobExecutor`.

```

type APKJobExecutor struct {
    Iterations          int
    TestTimeout         time.Duration
    WorkingDirectory    string
    EnableScreenshots   bool
    EnableAssertions    bool
    Modules              APKJobExecutorModules
    Results              common.JobResult
    UsbHub              usbhub.UsbHub
}

```

Figure 8 – *Déclaration de la structure `APKJobExecutor`*

J’ai créé une structure `HardwareProbeModule` contenant (en plus des attributs d’un module : activé/désactivé...) une instance de `MeasureLauncher`, et je l’ai ajoutée à la structure `APKJobExecutorModules`. Dans la méthode de `JobLauncher` qui initialise l’instance `executor`, j’ai ajouté l’initialisation du module de mesure physique. Le paramètre `job`, qui regroupe les besoins de l’utilisateur, permet d’activer ou non le module tandis que l’instance `j` de `JobLauncher` fournit l’instance de `MeasureLauncher` (figure 9).

```

func (j *JobLauncher) InitAPKJobExecutor(job common.Job) APKJobExecutor {
    ...
    return APKJobExecutor{
        EnableAssertions: settings.GetBool("enableAssertions"),
        EnableScreenshots: settings.GetBool("enableScreenshots"),
        UsbHub:             j.config.UsbHub,
        Iterations:        job.Iterations,
        TestTimeout:       job.TestTimeout,
        Modules: APKJobExecutorModules{
            ...
            Hardware: HardwareProbeModule{
                Module: Module{
                    Enabled: job.Modules["hardware_probe"],
                },
                measureLauncher: j.measureLauncher,
            },
        },
        WorkingDirectory: job.WorkingDirectory,
    }
}

```

Figure 9 – *Méthode `InitAPKJobExecutor`*

J’ai ensuite intégré la mesure physique au sein des méthodes des différents types d’`executor`. À chaque itération, si le module est activé, on démarre la mesure avant les tests et on l’arrête après. La figure 10 illustre cette intégration dans la méthode `RunIterations` de `CommonAPKJobExecutor`, la structure relative aux tests d’applications Android.

```

func (e *CommonAPKJobExecutor) RunIterations(instrumentation string, arguments
↳ []string) error {
    firstIteration := true
    for iteration := 0; iteration < e.Iterations; iteration++ {
        logger.Infoln(fmt.Sprintf("Running iteration %d/%d", iteration+1,
↳ e.Iterations))
        if e.Modules.Hardware.Enabled {
            logger.Infoln(fmt.Sprintf("Starting hardware measurement
↳ module"))
            if err := e.Modules.Hardware.measureLauncher.Start(); err != nil {
                return err
            }
        }

        ...

        if e.Modules.Hardware.Enabled {
            logger.Infoln(fmt.Sprintf("Stopping hardware measurement module"))
            if err := e.Modules.Hardware.measureLauncher.Stop(); err != nil {
                return err
            }
        }

        e.retrieveResults(false)
        ...
    }

    ...

    return nil
}

```

Figure 10 – *Intégration de la mesure physique dans la méthode `RunIterations`*

L'étape suivante a consisté dans la récupération des résultats de la sonde physique et leur fusion avec les résultats des sondes logicielles.

3.1.3 Récupération des résultats

La méthode `retrieveResults` (qui est appelée fig 10) permet de récupérer les mesures et construire les résultats des tests. Elle appelle une méthode `CompleteTestResultsWithMeasures` destinée à compléter les résultats provenant des sondes logicielles du téléphone avec les mesures issues de la sonde réseau. J'ai modifié cette méthode pour inclure la mesure physique, en ajoutant à ses arguments les résultats ainsi qu'un booléen précisant si on veut inclure ou pas ces résultats.

```

func CompleteTestResultsWithMeasures(testResults []common.TestResultV2,
→ hardwareProbeData []common.MeasureTick, hardwareModule bool)
→ ([]common.TestResultV2, error) {
    ...

    if hardwareModule && len(hardwareProbeData) > 0 {
        mergedTestResults := make([]common.TestResultV2, 0)
        for _, testResult := range mergedTestResultsNetwork {
            err := CompleteTestResultMeasures(&testResult,
→ hardwareProbeData)
            if err != nil {
                logger.Warn(err.Error())
            }
            mergedTestResults = append(mergedTestResults, testResult)
        }
        return mergedTestResults, nil
    } else {
        return mergedTestResultsNetwork, nil
    }
}

```

Figure 11 – *Intégration des résultats de la mesure physique*

À ce stade j’ai rencontré une difficulté : la fusion des résultats ne fonctionnait pas. En debugant j’ai réalisé que les dates (`timestamps`) des points de mesure logicielle et matérielle n’étaient pas cohérents. En effet les dates des points de mesure physique sont issues du poste sur lequel le programme tourne tandis les points de mesure logicielle sont datés à partir de l’heure du mobile. Il existait un léger décalage (~30s) qui empêchait la fusion des résultats.

Pour résoudre ce problème, j’ai ajouté le champ `config` de type `HardwareMeasureLauncherConfig` au sein de `HardwareMeasureLauncher`. Ce champ contient un champ `GetTimeDelta` qui est une fonction donnant la différence entre l’heure du poste et celle du téléphone (figure 12). L’avantage de cette solution est que pour les autres plateformes (iot et iOS) qui ne présentent pas ce problème, on peut simplement renvoyer 0, et éviter d’avoir autant d’implémentations que de plateformes. La figure 13 présente l’implémentation dans le cas d’un mobile Android⁶.

```

type HardwareMeasureLauncherConfig struct {
    GetTimeDelta func() (int64, error)
}

```

Figure 12 – *Déclaration de la structure `HardwareMeasureLauncherConfig`*

6. Ce bout de code apparaît dans `testrunner`

```

GetTimeDelta: func() (int64, error) {
    adbCommandLauncher := adb.NewAdbCommandLauncher()
    deviceTime, err := adbCommandLauncher.DeviceTime()
    if err != nil {
        return int64(0), err
    }
    return util.Now() - deviceTime, nil
}

```

Figure 13 – *Instanciation de `GetTimeDelta` dans le cas Android*

La fonction est appelée lors de la mesure physique, dans la méthode `startMeasure` : on prend en compte le delta temporel lorsqu'on stocke les points de mesure dans le tableau `hardwareMeasureResults`.

3.2 testrunner

Le `testrunner` a pour rôle de créer les objets nécessaires à la réalisation des tests, selon les besoins de l'utilisateur. J'ai donc dû modifier ce projet pour finir d'implémenter la fonctionnalité de mesure physique.

J'ai modifié le fichier principal pour créer une instance de `SerialControllerConfig` (figure 14). Cette instance contient les informations relatives au port série utilisé pour la mesure (carte **Arduino**). L'adresse du port série est renseignée par l'utilisateur dans le fichier de configuration `config.yml`⁷. Les autres informations ne sont pas modifiables.

```

address := settings.GetString("target.hardware.address")
if address == "" {
    address = "/dev/ttyACM0"
}

controllerConfig := controller.SerialControllerConfig{
    Address: address,
    Baudrate: 115200,
    DataBits: 8,
    Parity: "N",
    StopBits: 2,
}

```

Figure 14 – *Instanciation de la configuration du port série dans le programme principal du `testrunner`*

Le fichier principal appelle la méthode `LaunchTestRunnerJob` du package `android`. J'ai ajouté l'objet `controllerConfig` à ses paramètres. `LaunchTestRunnerJob` instancie un `Job` (grâce aux informations du fichier de configuration `job.yml`) et un `JobLauncher` avec notamment son `MeasureLauncher` pour la mesure physique. Ensuite elle appelle la méthode de `go-testapi` correspondante au mode (`url`, `apk` ou `custom`). Le code de la méthode se trouve dans l'annexe B.

7. Si cette information n'est pas renseignée, on utilise une valeur par défaut correspondant au port série des postes Linux

4 Adaptation du dispositif

Comme évoqué dans le paragraphe 2.1, il a fallu adapter la sonde physique dans le but de mesurer la consommation d'énergie sur mobile. L'idée est de placer le shunt entre le téléphone et la batterie et de mesurer le courant sortant de cette dernière vers l'appareil.

Pour cette tâche, nous avons choisi d'utiliser un **Samsung Galaxy J3** car sa batterie est facilement accessible.

La figure 15 présente les connecteurs que nous avons utilisés. Ces connecteurs établissent le contact entre les bornes de la batterie et les fils électriques d'un côté, ainsi qu'entre le téléphone et les fils de l'autre.

Nous avons soudé les fils électriques à chaque borne des connecteurs. De chaque côté, les bornes + et - sont reliées au shunt, de sorte que ce dernier est placé entre la batterie et le téléphone. Chacune des bornes restantes du côté de la batterie relie la borne correspondante du côté du mobile grâce un fil, sans passer par le shunt.

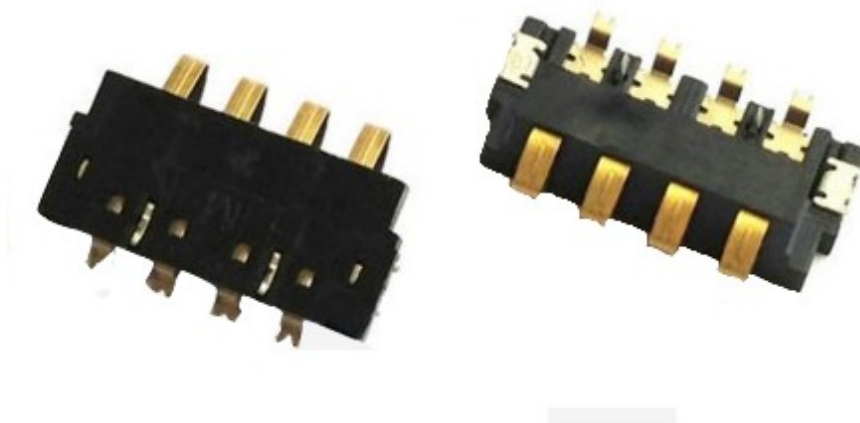


Figure 15 – *Connecteurs batterie*

5 Analyse des résultats

Étude de faisabilité iOS

1 Introduction

Pour la seconde tâche majeure de mon stage, il m'a été demandé d'étudier la possibilité d'étendre la solution *Greenspector* aux appareils iOS. Cette fonctionnalité représente pour l'entreprise l'opportunité de cibler un marché plus large. Cependant l'équipe de développement sait que sa réalisation est plus difficile que pour les appareils Android et demande du temps. C'est pourquoi j'ai été chargé d'avancer le plus possible sur le sujet avant qu'il soit repris par l'équipe GREENSPECTOR.

2 Cadre

2.1 Appareil

J'ai travaillé sur mon téléphone personnel (**iPhone 6s**) au début de l'étude, puis un **iPhone 7** (iOS 11.2) a été mis à ma disposition par l'entreprise pour la suite.

Nous avons envisagé d'utiliser un appareil débridé (jailbroken) pour contourner les restrictions et sécurités d'Apple et accéder facilement aux informations intéressantes. Cependant, pour des raisons de stabilité¹, nous avons abandonné cette idée.

2.2 Sonde physique

Compte tenu de la difficulté présumée de récupérer des métriques logicielles sur iOS, il a été décidé au préalable que la sonde physique serait utilisée pour mesurer la consommation énergétique de l'appareil.

2.3 Technologies utilisées

En plus des projets **go-testapi** et **test runner** (écrits en go), j'ai dû prendre en main l'environnement **Xcode** d'Apple ainsi que **Swift**, le langage de programmation également développé par Apple.

3 Tests automatisés

La première étape de l'étude a consisté dans l'automatisation des différentes étapes des tests standards automatisés². J'avais pour objectif de trouver une solution pour interagir avec l'appareil, notamment lancer des applications à distance (Wi-Fi seulement) via le terminal ou un programme, le but final étant d'effectuer une mesure pendant l'exécution des tests (benchmark).

1. Chaque mise à jour du système d'exploitation iOS améliore sa sécurité. Pour mettre à jour un appareil débridé, il faut donc réussir à exploiter de nouvelles failles à chaque version, ce qui ne peut être garanti.

2. Voir la section sur l'interface graphique

3.1 Premiers essais : libimobiledevice et ios-deploy

Dans un premier temps j’ai cherché à lancer une application sur l’appareil en ligne de commande. J’ai essayé deux outils : `ios-deploy` et `libimobiledevice`.

3.1.1 ios-deploy

`ios-deploy`³ est un outil open source permettant de déployer et lancer des applications sur les appareils iOS en ligne de commande. L’outil présente des fonctionnalités intéressantes, dont quelques exemples sont présentés sur la figure 16.

```
// Installer une app et la lancer en debug
ios-deploy --debug --bundle my.app

// Installer et lancer d une application puis quitter le debugger
ios-deploy --justlaunch --debug --bundle my.app

// Télécharger les fichiers de l app
ios-deploy --bundle_id <bundle.id> --download --to MyDestinationFolder

// Lister les fichiers d une app
ios-deploy --bundle_id <bundle.id> --list

// Désinstaller puis réinstaller et lancer
ios-deploy --uninstall --debug --bundle my.app

// Vérifier l existence d un package
ios-deploy --exists --bundle_id com.apple.mobilemail

// Désinstaller une app
ios-deploy --uninstall_only --bundle_id my.bundle.id

// Lister les app installées sur l appareil
ios-deploy --list_bundle_id
```

Figure 16 – *Exemples d’utilisation de ios-deploy*

J’ai testé les possibilités offertes par l’outil, ce qui m’a permis d’évaluer son utilité pour la fonctionnalité.

Avantages `ios-deploy` présente quelques avantages utiles pour l’implémentation d’un benchmark iOS :

- L’installation de l’outil est rendue très simple⁴ grâce au node package manager :
\$ `npm install -g ios-deploy`
- L’outil permet d’installer/désinstaller l’application à tester à partir du fichier `.app` obtenu après compilation du projet Xcode.

3. <https://github.com/ios-control/ios-deploy>

4. Il est important de considérer cette problématique lors de la conception pour alléger au maximum l’installation de *Greenspector*. Les éventuels pré-requis demandés aux clients doivent être simples et peu contraignants.

- La plupart des actions et commandes (pas toutes !) sont utilisables avec l'appareil connecté en Wi-Fi.

Points bloquants Cependant, l'utilisation de `ios-deploy` est limitée :

- Le lancement d'une application est effectué en mode debug. Les applications natives ainsi que celles de l'AppStore ne possèdent pas cette propriété, et ne peuvent donc pas être lancées par l'outil. En particulier, cela rend la mesure de sites webs impossible puisqu'on ne peut pas ouvrir un navigateur.
- De plus, il n'est pas possible de lancer une application lorsque l'appareil est connecté en Wi-Fi.

3.1.2 libimobiledevice

`libimobiledevice`⁵ est une bibliothèque multiplateformes permettant de communiquer avec des appareils iOS (iPhone, iPod Touch, iPad, Apple TV). Elle permet notamment d'accéder aux fichiers système, de gérer les applications installées... `libimobiledevice` fonctionne avec les appareils sous iOS 10 (et versions antérieures).

J'ai testé quelques fonctionnalités de cet outil pour comparer son utilisation à celle de `ios-deploy`.

Avantage La commande `idevicesyslog` permet d'afficher les logs de l'appareil en temps réel, ce qui peut être intéressant pour remonter des erreurs ou autres informations utiles aux tests.

Points bloquants `libimobiledevice` comporte les mêmes points bloquants que `ios-deploy`.

J'ai choisi d'envisager l'utilisation de `ios-deploy` pour installer/désinstaller des applications, plus commode que `libimobiledevice`. Cependant cet outil seul ne suffit pas pour implémenter la fonctionnalité.

3.2 Vers la solution : WebDriverAgent

Il manquait un moyen d'interagir avec l'appareil et mener les tests de façon automatisée. Mes recherches et mes collègues m'ont amené à m'intéresser l'outil d'automatisation de tests `Appium`⁶, qui permet effectivement de mener des tests automatisés sur appareils iOS (entre autres). J'ai entrepris d'étudier cet outil open source afin d'en comprendre le fonctionnement et l'adapter à mon problème. Pour les tests iOS, `Appium` utilise l'outil `WebDriverAgent`.

3.2.1 WebDriverAgent

`WebDriverAgent`⁷ est un outil développé par FACEBOOK. Il s'agit d'un serveur `WebDriver` pour iOS permettant de contrôler des appareils à distance. Il permet notamment de lancer des applications (y compris celles de l'App Store) et effectuer des actions sur l'écran (scroller, taper du texte...), ce qui répond à notre besoin.

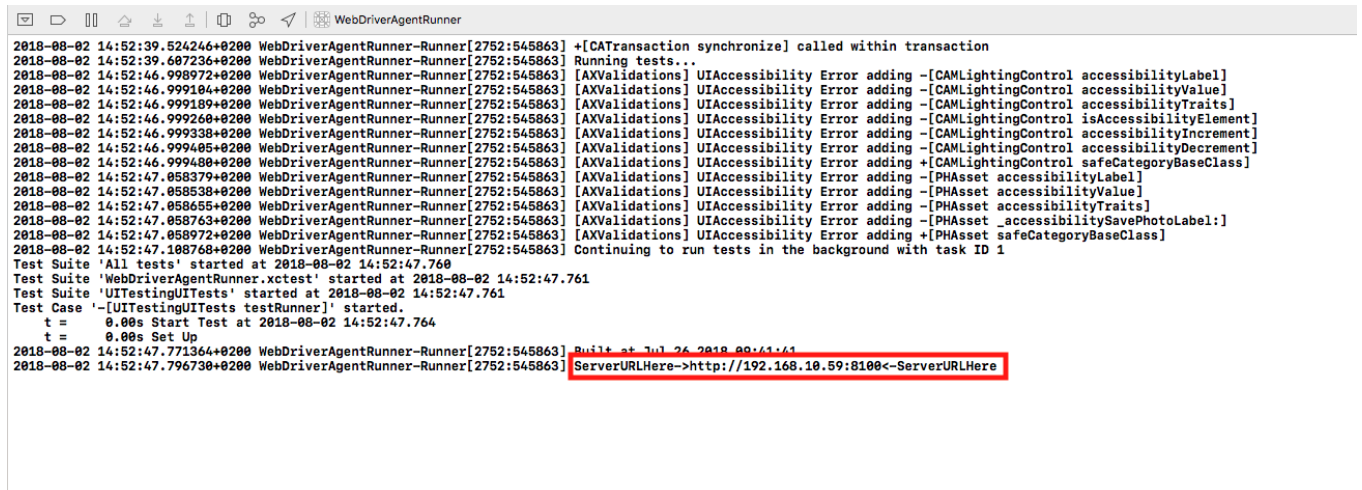
5. www.libimobiledevice.org

6. <https://github.com/appium/>

7. <https://github.com/facebook/WebDriverAgent>

3.2.2 Fonctionnement

`WebDriverAgent` est lancé sur l'appareil depuis `Xcode` en mode test. Suite au lancement, le serveur fournit son URL (souvent `http://<IP_TÉLÉPHONE>:8100` — voir fig. 17). Un certain nombre de routes définies dans le projet permettent au serveur de mener des actions sur le mobile (un clic par exemple) en fonction de la requête effectuée.



```
2018-08-02 14:52:39.524246+0200 WebDriverAgentRunner-Runner[2752:545863] +[CATransaction synchronize] called within transaction
2018-08-02 14:52:39.607236+0200 WebDriverAgentRunner-Runner[2752:545863] Running tests...
2018-08-02 14:52:46.998972+0200 WebDriverAgentRunner-Runner[2752:545863] [AXValidations] UIAccessibility Error adding -[CAMLightingControl accessibilityLabel]
2018-08-02 14:52:46.999104+0200 WebDriverAgentRunner-Runner[2752:545863] [AXValidations] UIAccessibility Error adding -[CAMLightingControl accessibilityValue]
2018-08-02 14:52:46.999189+0200 WebDriverAgentRunner-Runner[2752:545863] [AXValidations] UIAccessibility Error adding -[CAMLightingControl accessibilityTraits]
2018-08-02 14:52:46.999268+0200 WebDriverAgentRunner-Runner[2752:545863] [AXValidations] UIAccessibility Error adding -[CAMLightingControl isAccessibilityElement]
2018-08-02 14:52:46.999338+0200 WebDriverAgentRunner-Runner[2752:545863] [AXValidations] UIAccessibility Error adding -[CAMLightingControl accessibilityIncrement]
2018-08-02 14:52:46.999408+0200 WebDriverAgentRunner-Runner[2752:545863] [AXValidations] UIAccessibility Error adding -[CAMLightingControl accessibilityDecrement]
2018-08-02 14:52:46.999480+0200 WebDriverAgentRunner-Runner[2752:545863] [AXValidations] UIAccessibility Error adding -[CAMLightingControl safeCategoryBaseClass]
2018-08-02 14:52:47.058379+0200 WebDriverAgentRunner-Runner[2752:545863] [AXValidations] UIAccessibility Error adding -[PHAsset accessibilityLabel]
2018-08-02 14:52:47.058538+0200 WebDriverAgentRunner-Runner[2752:545863] [AXValidations] UIAccessibility Error adding -[PHAsset accessibilityValue]
2018-08-02 14:52:47.058655+0200 WebDriverAgentRunner-Runner[2752:545863] [AXValidations] UIAccessibility Error adding -[PHAsset accessibilityTraits]
2018-08-02 14:52:47.058763+0200 WebDriverAgentRunner-Runner[2752:545863] [AXValidations] UIAccessibility Error adding -[PHAsset _accessibilitySavePhotoLabel:]
2018-08-02 14:52:47.058972+0200 WebDriverAgentRunner-Runner[2752:545863] [AXValidations] UIAccessibility Error adding -[PHAsset safeCategoryBaseClass]
2018-08-02 14:52:47.188768+0200 WebDriverAgentRunner-Runner[2752:545863] Continuing to run tests in the background with task ID 1
Test Suite 'All tests' started at 2018-08-02 14:52:47.760
Test Suite 'WebDriverAgentRunner.xctest' started at 2018-08-02 14:52:47.761
Test Suite 'UITestingUITests' started at 2018-08-02 14:52:47.761
Test Case '-[UITestingUITests testRunner]' started.
t = 0.00s Start Test at 2018-08-02 14:52:47.764
t = 0.00s Set Up
2018-08-02 14:52:47.771364+0200 WebDriverAgentRunner-Runner[2752:545863] Built at Jul 26 2018 09:41:41
2018-08-02 14:52:47.796730+0200 WebDriverAgentRunner-Runner[2752:545863] ServerURLHere->http://192.168.10.59:8100<-ServerURLHere
```

Figure 17 — `WebDriverAgent` : affichage de l'adresse du serveur dans la console

Le serveur tourne sur le mobile (même verrouillé) tant que ce dernier est allumé et connecté en Wi-Fi.

3.2.3 Prise en main

J'ai pris en main `WebDriverAgent` à l'aide des exemples fournis sur github. J'ai écrit les premières requêtes en utilisant `cURL`, une interface en ligne de commande qui permet cela.

L'écriture des requêtes met en jeu 3 variables :

- L'adresse du serveur évoquée dans la section 3.2.2 : `DEVICE_URL`
- L'entête qui permet de travailler avec des requêtes et réponses en JSON : `JSON_HEADER='-H "Content-Type: application/json"'`
- Un identifiant de session `SESSION_ID`, fourni au lancement d'une application. Par exemple : `D15E12F6-CA23-4CD4-89F9-E5C5EA6F4FAD`

La figure 18 présente des requêtes utiles que j'ai testées :

- Accéder à l'écran d'accueil
- Démarrer une session et lancer une application⁸ (la réponse à cette requête fournit la variable `SESSION_ID` utile pour les requêtes qui suivent)
- Rechercher un élément sur l'écran à l'aide de son nom (la réponse à cette requête fournit l'identifiant de l'élément recherché permettant l'interaction avec ce dernier)
- Cliquer sur un élément
- Taper du texte
- Défiler sur la page
- Terminer une session et fermer l'application

8. Il est possible d'installer l'application avant de la lancer. Cependant le lancement est effectué directement après l'installation, ce qui empêche la réalisation de l'étape de référence.

```

curl -X POST $JSON_HEADER -d "" $DEVICE_URL/wda/homescreen

curl -X POST $JSON_HEADER \
-d "{\"desiredCapabilities\":{\"bundleId\":\"com.apple.mobilesafari\"}}" \
$DEVICE_URL/session

curl -X POST $JSON_HEADER \
-d "{\"using\":\"link text\",\"value\":{\"name=URL\"}}" \
$DEVICE_URL/session/$SESSION_ID/elements

curl -X POST $JSON_HEADER -d "" $DEVICE_URL/session/$SESSION_ID/element/5/click

curl -X POST $JSON_HEADER \
-d "{\"value\":\"e\",\"c\",\"n\",\".\",\"f\",\"r\",\"\n\"]}" \
$DEVICE_URL/session/$SESSION_ID/element/5/value

curl --silent -X POST $JSON_HEADER \
-d "{\"fromX\":\"100\",\"fromY\":\"500\",\"toX\":\"100\",\"toY\":\"80\", \"duration\":\"0.2\"}" \
$DEVICE_URL/session/$SESSION_ID/wda/dragfromtoforduration

curl -X DELETE $JSON_HEADER $DEVICE_URL/session/$SESSION_ID

```

Figure 18 – Exemples d'utilisation de *WebDriverAgent* grâce à *cURL*

3.3 Écriture de scripts de benchmark

Après avoir pris en main ces commandes j'ai entrepris d'écrire des scripts de tests automatisés afin de mieux maîtriser l'enchaînement des différentes actions. Il s'agissait d'effectuer les bonnes requêtes avec les informations adéquates. Par exemple, pour fermer une application il faut effectuer la requête avec le bon identifiant de session, il faut donc s'assurer d'avoir accès à ce dernier en le récupérant lors du lancement. Pour les étapes de test je me suis basé sur les tests existants sur Android (voir le paragraphe 2.2.4 de la présentation de l'outil) en réduisant le temps de test. Les scripts sont présentés dans l'annexe C.

3.3.1 Benchmark de site web

Le script de benchmark de site internet prend en argument l'URL du serveur *WebDriverAgent* et l'URL de la page testée.

3.3.2 Benchmark d'application

Ici on passe en argument le chemin vers le fichier `.app` et l'identifiant de l'application (par exemple, pour le navigateur d'Apple : `com.apple.mobilesafari`) en plus de l'URL du serveur *WebDriverAgent*.

Je me suis servi de ces scripts de benchmark comme base de travail pour l'implémentation de la fonctionnalité iOS dans `go-testapi` et `testrunner`.

4 Implémentation dans `go-testapi` et `testrunner`

4.1 `go-testapi`

Pour implémenter la fonctionnalité iOS, j'ai principalement travaillé sur `go-testapi`. Pour cela j'ai créé au sein du projet un package `ios`, contenant tous les objets et méthodes utiles à la réalisation des tests et des mesures.

4.1.1 Package `device`

J'ai commencé par créer un package `device` avec une interface du même nom dans le package `ios`. Cette interface met en jeu les méthodes d'interaction avec l'appareil, présentées dans la figure 19. Les noms de méthodes commençant par une minuscule correspondent aux méthodes "utilitaires" privées et appelées par les autres méthodes.

Ensuite, dans un autre fichier, j'ai créé les structures `IosDeviceConfig` (contenant les informations relatives à un appareil) et `IosDevice` (contenant une instance de `IosDeviceConfig`) qui implémente l'interface `Device` (déclaration fig. 20). Pour implémenter les méthodes de l'interface, j'ai adapté mes tests de `WebDriverAgent` en effectuant les différentes requêtes nécessaires grâce à des méthodes `go`, ce qui rend les interactions plus commodes. En effet, un appel de méthode est beaucoup plus simple et explicite qu'une ligne de commande `cURL`.

J'ai également créé une version mockée de `IosDevice` pour en simuler le comportement lors de tests unitaires⁹.

9. L'utilisation de mocks pour les tests unitaires automatisés justifie la création d'une interface.

```

package device

import "gitlab.kali/kaliterre/go-testapi/common"

type Device interface {
    openApp(bundleId string) (string, error)
    killApp(sessionId string) error
    scroll(sessionId string, fromY int, toY int) error
    findElementByName(sessionId string, name string) (string, error)
    clickOnElement(sessionId string, elementId string) error
    typeInElement(sessionId string, elementId string, content string) error
    homeScreen() error
    getDefaultSession() (string, error)

    LaunchAgent() error
    ClearWebData() error
    CloseSafariTabAndKillApp() error
    RunWebBenchmark(websiteURL string) (common.Steps, error)
    RunAppBenchmark(path string) (common.Steps, error)
    InstallApp(path string, bundleId string) error
    UninstallApp(bundleId string) error
    GetDeviceName() string
}

```

Figure 19 – Méthodes de l'interface Device

```

type IosDeviceConfig struct {
    DeviceName string

    //Identifiant du device iOS (sert pour ios-deploy)
    UDID string

    //URL de l'appareil pour faire des requêtes
    URL string

    //Chemin d'accès à ios-deploy si pas dans le path
    PathToIosDeploy string

    //Chemin vers WebDriverAgent
    PathToWDA string
}

type IosDevice struct {
    Config IosDeviceConfig
}

```

Figure 20 – Déclaration des structures IosDeviceConfig et IosDevice

4.1.2 Autres entités du package ios

Lancement des tests sur l'appareil Pour lancer la procédure de test sur un appareil, j'ai créé une interface `JobLauncher` comportant deux méthodes (fig. 21). Ces dernières sont destinées à mener toutes les actions de test et de mesures nécessaires selon le besoin de l'utilisateur (contenu dans l'objet `job`, qui présente notamment l'application à mesurer, le nombre d'itérations etc...) et renvoyer les résultats.

```
type JobLauncher interface {  
    LoadURLJob(job common.Job) (common.JobResult, error)  
    LoadAppJob(job common.Job) (common.JobResult, error)  
}
```

Figure 21 – *Déclaration de l'interface JobLauncher*

La structure `IosJobLauncher` — qui contient un `device` et un `measureLauncher` (pour déclencher la mesure physique) — implémente l'interface `JobLauncher`. L'algorithme général de l'implémentation des méthodes de `JobLauncher` se présente comme suit :

- Lancement de `WebDriverAgent` sur l'appareil
- Démarrage de la mesure physique
- Déroulement des tests sur une ou plusieurs itérations¹⁰
- Arrêt de la mesure physique
- Construction et renvoi des résultats

En plus des méthodes de l'interface, `IosJobLauncher` comprend des méthodes auxiliaires comme `buildResults` qui organise les résultats de mesure en fonction des étapes de test. C'est d'ailleurs l'utilité du type `common.Steps` renvoyé par les méthodes `RunWebBenchmark` et `RunAppBenchmark` de `Device` (fig. 19). Il consiste en un tableau de `common.Step`, comprenant le nom de l'étape de test ainsi que ses dates de début et de fin.

Tests unitaires Afin de tester un minimum le code, j'ai entrepris d'écrire des tests unitaires. J'ai créé deux méthodes de test : chacune d'elle permet de tester une méthode de `JobLauncher`. Pour cela j'ai créé, au sein de chaque méthode de test, une instance de `IosJobLauncher` avec des mocks comme attributs (`device` et `measureLauncher`), ainsi qu'un `job`. Ensuite on compare les résultats du faux test avec les résultats attendus. La figure 22 présente la méthode `TestURLjob` qui teste le lancement d'un benchmark web.

10. On note que dans le cas d'un benchmark d'application, l'installation et la désinstallation ne sont pas effectuées à chaque itération mais seulement avant et après l'exécution de tous les tests.

```

func TestURLJob(t *testing.T) {
    t.Parallel()

    // Job URL
    deviceInstance := device.NewMockedIosDevice(device.MockedIosDeviceConfig{
        DeviceName: "iPhone7",
    })
    jobLauncher := &IosJobLauncher{
        device:          deviceInstance,
        measureLauncher: hardware.NewMockedMeasureLauncher(),
    }

    modules := make(map[string]bool)
    modules["hardware_probe"] = true

    job := common.Job{
        Mode: "url",
        GreenspectorApp: common.GreenspectorApp{
            Name:      "ApplicationTest",
            Version:    "2",
        },
        Iterations: 5,
        URLs:       []string{"www.greenspector.com"},
        Modules:    modules,
    }
    jobResult, err := jobLauncher.LoadURLJob(job)

    assert.Equal(t, err, nil)
    assert.Equal(t, len(jobResult.Results), 5)
    assert.Equal(t, jobResult.Results[0].Application, "ApplicationTest")
    assert.Equal(t, jobResult.Results[0].Version, "2")
    assert.Equal(t, jobResult.Results[0].Devicename, "iPhone7")
    assert.Equal(t, jobResult.Results[0].Iterations, 5)
    assert.Equal(t, len(jobResult.Results[0].Measures), 1)
}

```

Figure 22 – Une méthode de test unitaire pour *IosJobLauncher*

4.2 testrunner

J'ai ensuite travaillé sur **testrunner** pour créer les objets nécessaires au déroulement des tests. J'ai créé au sein du projet un package **ios** avec un unique fichier. Ce fichier contient une seule fonction **LaunchIosJob**, appelée par la fonction principale de **testrunner**.

Le fonctionnement peut se résumer en 5 étapes :

- Dans la fonction principale, on lit l'attribut **platform** du fichier de configuration **config.yml**. S'il s'agit de **ios**, la fonction principale appelle **LaunchIosJob**.

- Dans `LaunchIosJob`, on instancie un `device` et un `measureLauncher` grâce aux informations de `config.yml`. Ces instances permettent de créer une instance de `IosJobLauncher`.
- Le fichier de configuration `job.yml` qui contient les informations relatives aux tests à mener permet d’instancier un `job`.
- On appelle ensuite la bonne méthode de notre instance de `IosJobLauncher` selon le mode (`url` ou `app`).
- On récupère les résultats des tests pour mener la suite des actions (sauvegarde locale ou envoi sur l’interface web *Greenspector*).

5 Ajout de métriques supplémentaires

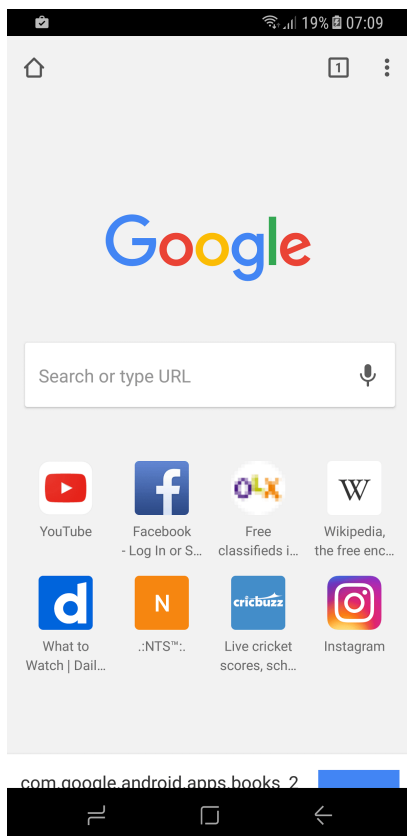
Pour la suite de l’étude, j’ai été chargé de trouver des solutions pour remonter des métriques de performance (CPU, mémoire...) des appareils iOS pendant les tests.

Tâches supplémentaires

Parallèlement à mes sujets principaux, j'ai été chargé d'effectuer deux petites tâches.

1 Benchmark web Android : mesure de référence

La mesure de référence pour les tests de page web était basée sur l'écran d'accueil du navigateur **Chrome**. Nous avons remarqué que cet écran pouvait changer selon l'appareil. Les icônes également pouvaient changer (lorsque le logo d'un site présent est mis à jour par exemple). Pour stabiliser un peu plus les mesures, j'ai été chargé de modifier le projet `testbench-launcher-android` pour charger la page `about:blank` avant la mesure de référence.



Pour cela j'ai ajouté deux lignes à la méthode qui effectue les différentes étapes du test automatisé, juste avant l'étape de référence (figure 23).

```
try {  
    this.device.wait(10000);  
    if (testName.equals("initial")) {  
        this.device.goToUrlInBrowser("about:blank");  
        this.device.pressKey(KeyEvent.KEYCODE_ENTER);  
        runReferenceMeasure("url");  
    }  
}
```

Figure 23 – *Page web blanche pour la mesure de référence*

2 Calibration Samsung Galaxy S9

Annexe A

Tableau de bord : exemple d'analyse du site de l'École Centrale de Nantes

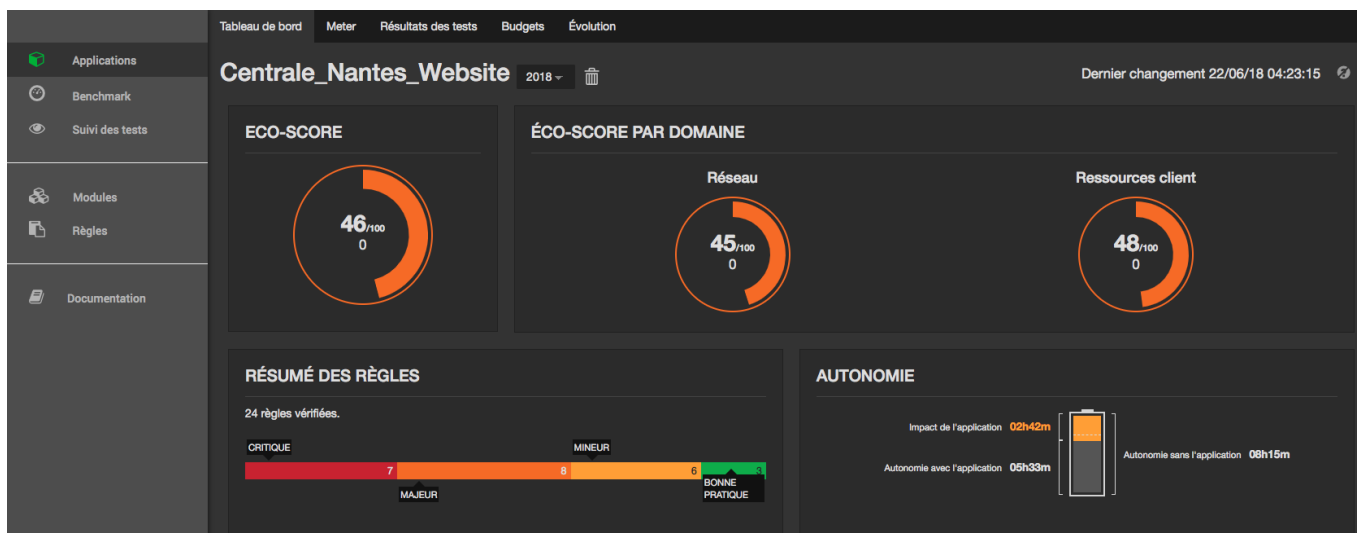


Figure A.1 – Tableau de bord. Vue 1



Figure A.2 – Tableau de bord. Vue 2

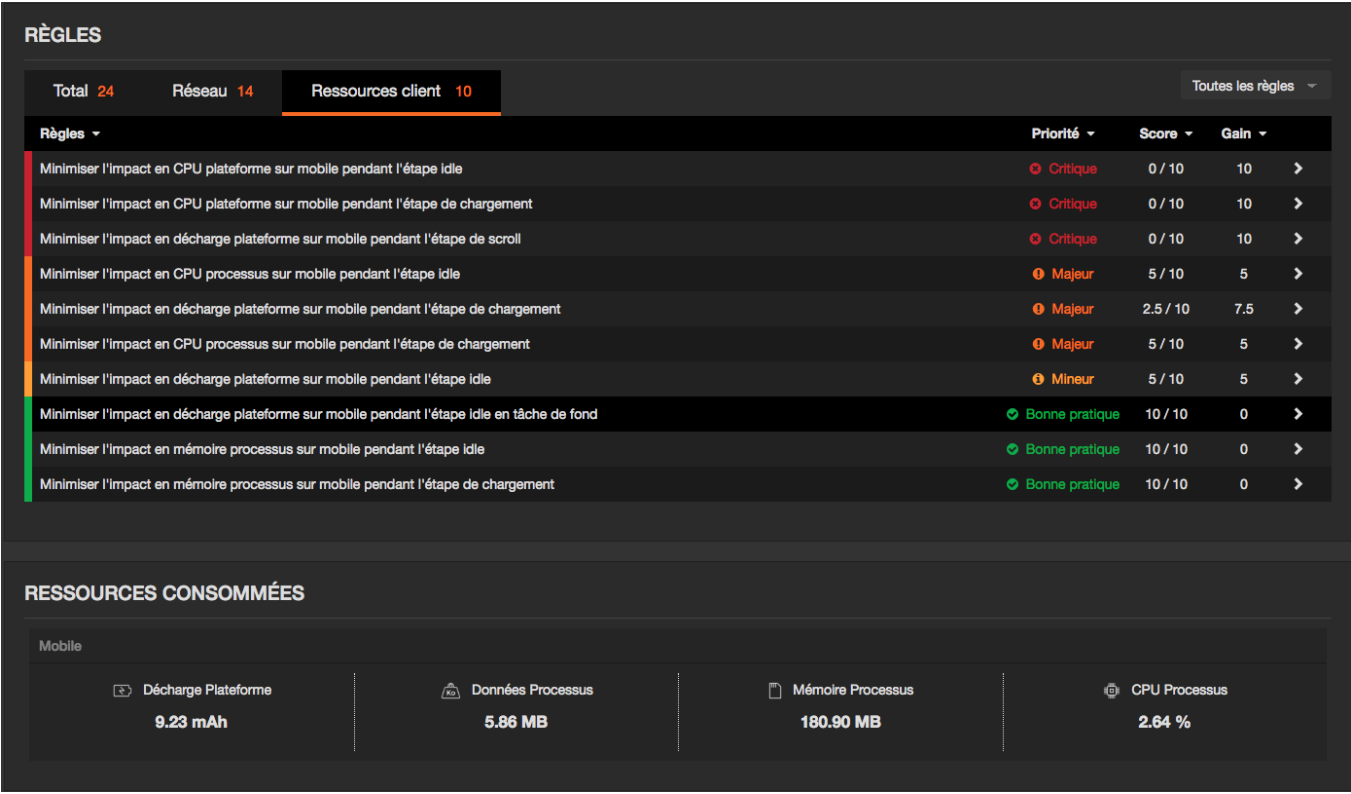


Figure A.3 – Tableau de bord. Vue 3

Annexe B

Intégration de la mesure physique dans testrunner

```
func LaunchTestRunnerJob(UsbHub usbhub.UsbHub, controllerConfig
↪ controller.SerialControllerConfig) error {
    mode := settings.Job.GetString("mode")

    defer adb.Disconnect()

    urls := settings.Job.GetStringSlice("job.urls")

    if len(urls) == 0 {
        urls = []string{settings.Job.GetString("job.url")}
    }

    job := common.Job{
        URLs:          urls,
        LocationType:   settings.Job.GetString("job.locationType"),
        TestPackages:   settings.Job.GetStringSlice("job.testPackages"),
        MonitoredPackage: settings.Job.GetString("job.monitoredPackage"),
        Iterations:     settings.Job.GetInt("job.iterations"),
        TestTimeout:    settings.Job.GetDuration("job.testTimeout"),
        Scenario:       settings.Job.GetString("job.scenario"),
        AuthenticationForm: common.AuthenticationForm{
            Actions:
                ↪ settings.Job.GetStringSlice("job.authentication.actions"),
        },
        Cache: settings.Job.GetBool("job.cache"),
        Browser: settings.Job.GetString("job.browser"),
        Modules: settings.Job.GetStringMapBool("job.modules"),
        Mode:    mode,
        Online: settings.Job.GetBool("job.online"),
        GreenspectorApp: common.GreenspectorApp{
            Name:
                ↪ settings.Job.GetString("greenspector.application.name"),
            Version:
                ↪ settings.Job.GetString("greenspector.application.version"),
        },
    }
```

```

    },
}

var jobResult common.JobResult
var err error
jobLauncher := jobs.NewJobLauncher(
    jobs.JobLauncherConfig{
        UsbHub: UsbHub,
        MeasureLauncherConfig:
            ↪ hardware.HardwareMeasureLauncherConfig{
                GetTimeDelta: func() (int64, error) {
                    adbCommandLauncher :=
                        ↪ adb.NewAdbCommandLauncher()
                    deviceTime, err :=
                        ↪ adbCommandLauncher.DeviceTime()
                    if err != nil {
                        return int64(0), err
                    }
                    return util.Now() - deviceTime, nil
                },
            },
        SerialConfig: controllerConfig,
    },
)

switch mode {
case "url":
    jobResult, err = jobLauncher.LaunchURLJob(job)
case "apk":
    jobResult, err = jobLauncher.LaunchStandardApkJob(job)
case "apkCustom":
    return fmt.Errorf("`apkCustom` mode has been deprecated. Please
        ↪ change it to `custom`.")
case "custom":
    jobResult, err = jobLauncher.LaunchCustomTestsJob(job)
default:
    return fmt.Errorf("unknown job mode: %s", mode)
}
if err != nil {
    return err
}

testFailed := false

for _, testStatus := range jobResult.TestsStatus {
    if testStatus.Status == "failed" {
        testFailed = true
        break
    }
}

```

```

}

if testFailed {
    for test, testStatus := range jobResult.TestsStatus {
        if testStatus.Status == "success" {
            logger.Info("%s: passed", test)
        } else {
            logger.Error("%s: failed. Reason: %s", test,
                ↪ testStatus.StackTrace)
        }
    }
}

return results.HandleResults(job, jobResult)
}

```


Annexe C

Scripts de tests automatisés sur iOS

1 Benchmark web

```
#!/bin/bash

DEVICE_URL=$1
URL=$2
JSON_HEADER='-H "Content-Type:application/json"'

clear_history_cache() {
    echo "Suppression de l'historique et des caches"
    SETTINGS_SESSION_ID=$(curl --silent -g -X POST $JSON_HEADER -d
    ↪ '{"desiredCapabilities":{"bundleId":"com.apple.Preferences"}}'
    ↪ $DEVICE_URL/session | jq -r '.sessionId')

    SCROLL1=$(curl --silent -X POST $JSON_HEADER -d
    ↪ '{"fromX\\":\\"100\\",\\"fromY\\":\\"600\\",\\"toX\\":\\"100\\",\\"toY\\":\\"80\\",
    ↪ \\"duration\\":\\"0.2\\"}'
    ↪ $DEVICE_URL/session/$SETTINGS_SESSION_ID/wda/dragfromtoforduration)

    SCROLL2=$(curl --silent -X POST $JSON_HEADER -d
    ↪ '{"fromX\\":\\"100\\",\\"fromY\\":\\"600\\",\\"toX\\":\\"100\\",\\"toY\\":\\"80\\",
    ↪ \\"duration\\":\\"0.2\\"}'
    ↪ $DEVICE_URL/session/$SETTINGS_SESSION_ID/wda/dragfromtoforduration)

    SCROLL3=$(curl --silent -X POST $JSON_HEADER -d
    ↪ '{"fromX\\":\\"100\\",\\"fromY\\":\\"600\\",\\"toX\\":\\"100\\",\\"toY\\":\\"300\\",
    ↪ \\"duration\\":\\"0.2\\"}'
    ↪ $DEVICE_URL/session/$SETTINGS_SESSION_ID/wda/dragfromtoforduration)

    SAFARI_ELEMENT=$(curl --silent -g -X POST $JSON_HEADER -d '{"using":"link
    ↪ text","value":"name=Safari"}'
    ↪ $DEVICE_URL/session/$SETTINGS_SESSION_ID/elements | jq -r '.value |
    ↪ to_entries[1] | {"value": .value.ELEMENT}' | jq -r '.value')

    SAFARI=$(curl --silent -g -X POST $JSON_HEADER -d ""
    ↪ $DEVICE_URL/session/$SETTINGS_SESSION_ID/element/$SAFARI_ELEMENT/click)
```

```

SCROLL1=$(curl --silent -X POST $JSON_HEADER -d
↳ '{"fromX\\":\\"100\\",\\"fromY\\":\\"600\\",\\"toX\\":\\"100\\",\\"toY\\":\\"50\\",
\\"duration\\":\\"0.2\\"}'
↳ $DEVICE_URL/session/$SETTINGS_SESSION_ID/wda/dragfromtoforduration)

CLEAR_ELEMENT=$(curl --silent -g -X POST $JSON_HEADER -d '{"using":"link
↳ text","value":"name=Clear History and Website Data"}'
↳ $DEVICE_URL/session/$SETTINGS_SESSION_ID/elements | jq -r '.value |
↳ to_entries[0] | {"value": .value.ELEMENT}' | jq -r '.value')

CLEAR=$(curl --silent -g -X POST $JSON_HEADER -d ""
↳ $DEVICE_URL/session/$SETTINGS_SESSION_ID/element/$CLEAR_ELEMENT/click)

CONF_ELEMENT=$(curl --silent -g -X POST $JSON_HEADER -d '{"using":"link
↳ text","value":"name=Clear History and Data"}'
↳ $DEVICE_URL/session/$SETTINGS_SESSION_ID/elements | jq -r '.value |
↳ to_entries[0] | {"value": .value.ELEMENT}' | jq -r '.value')

CONF=$(curl --silent -g -X POST $JSON_HEADER -d ""
↳ $DEVICE_URL/session/$SETTINGS_SESSION_ID/element/$CONF_ELEMENT/click)
sleep 1

CLOSE=$(curl --silent -X DELETE $JSON_HEADER
↳ $DEVICE_URL/session/$SETTINGS_SESSION_ID)

echo "Terminé"
}

clear_history_cache

echo 'Reference step';
sleep 10

echo 'Launching Safari';
SESSION_ID=$(curl --silent -g -X POST $JSON_HEADER -d
↳ '{"desiredCapabilities":{"bundleId":"com.apple.mobilesafari"}}'
↳ $DEVICE_URL/session | jq -r '.sessionId')

echo 'Loading ' $URL;
URL_ELEMENT=$(curl --silent -g -X POST $JSON_HEADER -d '{"using":"link
↳ text","value":"name=URL"}' $DEVICE_URL/session/$SESSION_ID/elements | jq -r
↳ '.value | to_entries[0] | {"value": .value.ELEMENT}' | jq -r '.value')
ACCES_URL=$(curl --silent -g -X POST $JSON_HEADER -d '{"value":["'"$URL"\n"']}'
↳ $DEVICE_URL/session/$SESSION_ID/element/$URL_ELEMENT/value)
sleep 10

echo 'Foreground idle';

```

```

sleep 10

echo 'Scroll';
sleep 5
SCROLL1=$(curl --silent -X POST $JSON_HEADER -d
↳ '{"fromX\\":\\"100\\",\\"fromY\\":\\"500\\",\\"toX\\":\\"100\\",\\"toY\\":\\"80\\",
\\"duration\\":\\"0.2\\"}' $DEVICE_URL/session/$SESSION_ID/wda/dragfromtoforduration)
SCROLL2=$(curl --silent -X POST $JSON_HEADER -d
↳ '{"fromX\\":\\"100\\",\\"fromY\\":\\"500\\",\\"toX\\":\\"100\\",\\"toY\\":\\"80\\",
\\"duration\\":\\"0.2\\"}' $DEVICE_URL/session/$SESSION_ID/wda/dragfromtoforduration)
SCROLL3=$(curl --silent -X POST $JSON_HEADER -d
↳ '{"fromX\\":\\"100\\",\\"fromY\\":\\"500\\",\\"toX\\":\\"100\\",\\"toY\\":\\"300\\",
\\"duration\\":\\"0.2\\"}' $DEVICE_URL/session/$SESSION_ID/wda/dragfromtoforduration)
sleep 5

echo 'Background idle';
IDLE_BACK=$(curl --silent -X POST $JSON_HEADER -d "" $DEVICE_URL/wda/homescreen)
sleep 10

echo 'Closing Safari'
CLOSE=$(curl --silent -X DELETE $JSON_HEADER $DEVICE_URL/session/$SESSION_ID)

echo 'Done'

```

2 Benchmark d’application

```

#!/bin/bash

DEVICE_URL=$1
APP=$2
BUNDLE_ID=$3
JSON_HEADER='-H "Content-Type:application/json"'

echo 'Installation du package "'"$BUNDLE_ID"'"';
ios-deploy -b $APP

echo 'Reference'
sleep 3

echo 'Lancement'
SESSION_ID=$(curl --silent -g -X POST $JSON_HEADER -d
↳ '{"desiredCapabilities":{"bundleId":"'"$BUNDLE_ID"'"}}' $DEVICE_URL/session |
↳ jq -r '.sessionId')
sleep 5

echo 'Idle en arrière plan';

```

```
IDLE_BACK=$(curl --silent -X POST $JSON_HEADER -d "" $DEVICE_URL/wda/homescreen)
sleep 5

echo 'Fermeture'
CLOSING=$(curl --silent -X DELETE $JSON_HEADER $DEVICE_URL/session/$SESSION_ID)

echo 'Désinstallation du package "'$BUNDLE_ID'";
UNINSTALL=$(ios-deploy --uninstall_only --bundle_id $BUNDLE_ID)

echo 'Terminé'
```

Bibliographie

- [1] *Livre vert : Éco-conception des logiciels et services numériques*. Syntec Numérique, 2013.
- [2] négaWatt. La révolution numérique fera-t-elle exploser nos consommations d'énergie? *decrypterlenergie.org*, 2017.
- [3] Francis Vivat. *Impacts environnementaux des TIC*. 2011.
- [4] GreenIT.fr. Benchmark numérique responsable. Technical report, Club Green IT, 2017.
- [5] GreenIT.fr. Définition.
- [6] Olivier Philippot, Frédéric Bordage, and Thierry Leboucq. *Green Patterns, Manuel d'éco-conception des logiciels*. Green Code Lab, 2012.