Adaptation des interfaces à l'environnement

BOUDAB Jonathan	jonathan.boudab@etu.unice.fr	IAM	AngularJS
CHALTE Thomas	thomas.chalte@etu.unice.fr	IAM	VanillaJS
PEPIN Nicolas	nicolas.pepin@etu.unice.fr	IAM	Ionic
PRESTINI Adrien	adrien.prestini@etu.unice.fr	IAM	Xamarin

SOMMAIRE

Introduction **4**
Comment gérer l'adaptation aux dispositifs ? 5
Développement Responsive Web Design 5
Développement Adaptive Design 8
AngularJS : Tutorial **10**
Qu'est-ce que AngularJS et pourquoi l'utiliser ? 10
Installation et configuration d'AngularJS 10
Comparaisons des performances et analyses 11
WebPageTest 11
Responsive 12
Adaptive 13
PageSpeed Insights 14
Responsive 14
Adaptive 14
Comparaisons des données 15
VanillaJS : Tutorial **16**
Qu'est-ce que VanillaJS et pourquoi l'utiliser ? 16
Comparaisons des performances et analyses 17
WebPageTest 17
Alpha 18
Responsive 19
Adaptive 20
PageSpeed Insights 21
Alpha 21
Responsive 21
Adaptive 22
Comparaison des données 22
Comparaison des performances entre AngularJS et VanillaJS 23
Ionic 2 : Tutorial **24**
Qu'est-ce que IONIC ? 24
Configuration et installation d'IONIC2 25
Commencer avec Ionic 25
Arborescence d'un projet Ionic 26
Développement de l'application 26
Le projet FoodFactory 26
Le stockage de la liste de courses 27
Résultat de l'application 27
Différence entre "platforms" 27

Les services sous IONIC 2 28
Appel vers une API extérieure 28
Accès aux capteurs 29
Utilisation d'un plugin 29
Comment gérer l'adaptation aux dispositifs sous Ionic 2 ? 30
Utilisation d'un plugin cordova en fonction des platforms 30
Déploiement sur cibles réelles 30
Déploiement sur différents appareils 30
Test de performance 31
Conclusion sur Ionic 32

Xamarin : Tutorial 33
Qu'est-ce que Xamarin? 33
Configuration et installation Xamarin 33
Développement de l'application 36
Les services sous Xamarin 38
Accès aux capteurs 39
Comment gérer l'adaptation aux dispositifs sous Xamarin ? 41
Déploiement sur cibles réelles 41

Introduction

Dans ce rapport nous ferons l'étude de de quatre technologies différentes sur un seul et même problème: FoodFactory.

Nous pouvons nous demander pourquoi nous sommes partis vers ce thème. En effet, de plus en plus de consommateurs veulent savoir ce qu'ils achètent et surtout ce qu'ils mangent. Ainsi notre projet porte sur la lecture de produits alimentaires et des valeurs nutritionnelles. De plus nous nous sommes renseignés sur l'utilisation des applications mobiles et des sites web par les consommateurs. Il s'en est révélé que si le projet aboutissait sur l'utilisation de données sensibles (adresse, cartes bancaires) il faudrait une application mobile, alors que s'il n'y avait pas ces données et que notre projet servait uniquement à donner des informations sur les produits alimentaires, les consommateurs préféraient un site web utilisable depuis un ordinateur ou un mobile. De plus, il existe beaucoup de systèmes d'exploitations, ainsi le développement cross-plateforme et d'un site web adaptable sur mobile et ordinateur permet de toucher le plus grand nombre de personnes.

De ce fait, notre projet a pour but de montrer les limites et bénéfices des applications cross-plateformes (pour Ionic et Xamarin) et la différence des performances entre un framework (AngularJS) et le développement d'un site web de même fonctionnalité sans framework (VanillaJS).

Pour la partie Cross-plateforme, nous voulons mettre en avant les différents outils et techniques pour comprendre et mettre en place l'accès aux périphériques d'un smartphone (ici l'appareil photo pour scanner un code barre). Nous voulons mettre en avant les limites dans l'usage des technologies (Xamarin.Forms) par rapport aux technologies exclusives natives (iOS, Android et Windows phone dans leur langage de développement).

En ce qui concerne la partie la partie web, nous allons détailler les spécificités de développement de notre site web avec le framework AngularJS et sans framework. De plus nous allons expliquer comment nous avons amélioré les performances de notre site web. Enfin nous conclurons sur l'adaptation de notre site d'un ordinateur vers un smartphone.

Comment gérer l'adaptation aux dispositifs ?

1. Développement Responsive Web Design

Notre premier objectif était de développer la version responsive de notre site web. Celle-ci permet d'adapter ce projet sur n'importe quel type d'appareils sans avoir de modifications de versions à faire. Autrefois, nous n'avions qu'un ou deux navigateurs et les tailles d'écran variaient peu. Désormais, nous avons une multitude de navigateurs et un nombre important de résolutions (allant du 240x320 au 2515x1886, voire plus). Mais les supports également sont divers : nous pouvons accéder à Internet depuis nos classiques ordinateurs de bureau, mais aussi via les smartphones, tablettes, etc. Or nous savons par rapport à ces derniers que leur manière d'afficher un site Web diffère de celui d'un ordinateur de bureau.

Figure 1: Différences des tailles d'écrans

Enfin nous avons utilisé le responsive design pour ses trois notions principales:

- une grille d'affichage qui soit flexible :
  - autrement dit, un gabarit qui ne dépende pas d'une résolution minimale, maximale ;
- des médias flexibles :
  - à savoir, faire en sorte que nos images, vidéos, ne débordent pas du cadre de notre grille d'affichage/gabarit ;
- un ensemble de règles CSS basé sur Media Queries :
  - le principe: mettre des conditions sur l'affichage afin d'afficher/masquer, voire changer le rendu de notre application Web.

Pour ce faire, nous avons construit une arborescence simple et organisée en trois dossiers (html, scripts et styles) afin de rendre notre code maintenable, lisible et surtout adaptable pour la suite de notre projet.

Nous avons commencé par chercher à partir de quelle taille d'écran, une page devait s'afficher sous le format mobile. Voici le rendu que nous voulions afficher:

Figure 2: Page d'accueil du site web version Desktop

Figure 3: Page d'accueil du site web responsive version Mobile

Dans l'exemple de cette page, nous pouvons voir le changement de design entre la version desktop et la version mobile. Nous avons fait disparaître la bannière FoodFactory. La barre de navigation étant trop grande pour s'afficher la la version mobile, elle est cachée et transformée en un menu déroulant. De plus, pour un meilleur confort de l'utilisateur, les onglets pour chacune de nos pages web sont devenus des boutons cliquables faciles à identifier.

Pour mettre en place cette adaptation, nous avons dû utiliser du CSS et du Javascript. Au niveau du CSS, nous avons l'utilisation de media queries pour adapter l'affichage en fonction de la taille d'écran de notre appareil.

```
/* SmartPhone */
@media screen and (max-width: 480px) {
  .topnav a:not(:first-child) {
      display: none;
  }
  .topnav a.icon {
    float: right;
    display: block;
  }
  .topnav.responsive {
      position: relative;
  }
  .topnav.responsive a.icon {
    position: absolute;
    right: 0;
    top: 0;
  }
  .topnav.responsive a {
    float: none;
    display: block;
    text-align: left;
  }
```

Figure 4: Exemple de Media queries

Dans cette media queries nous avons indiqué que la taille de tout smartphone était inférieure à 480 pixels, ainsi le code CSS n'est pris en compte qu'à partir de cette taille d'écran. De plus nous pouvons aussi voir l'adaptation de la barre de navigation; quand elle est réduite, elle devient un menu déroulant vers le bas.

Enfin la dernière partie responsive est dans les deux versions, mobile ou desktop, la taille de toutes les "div" s'adapte à la place disponible. En effet, même si nous sommes sur la version desktop, la taille d'écran n'est pas universelle, il faut donc que la taille des composants s'adapte sur des écrans de différentes tailles; il en va de même pour toutes les tailles de téléphone mobile.

Figure 5: Responsive Design sur les “div” desktop et mobile

Ainsi le Responsive Design est une technologie très importante pour l’adaptation des pages à la taille des écrans mais elle a aussi de grosses limites: nous les verrons ci-dessous dans les liées à la performance.

2. Développement Adaptive Design

Dans un second temps, nous nous sommes plus axés sur les performances, or un site web responsive design pose de nombreux problèmes. En effet lorsqu’on charge une page sur desktop, nous devons aussi charger les éléments de la partie mobile et vice versa. Nous pouvons aujourd’hui argumenter cette réflexion en nous basant sur les géants du web tels que youtube, facebook, twitter et bien d’autres. Ces derniers ne disposent pas de site web responsive design. Ces géants du web proposent des sites web avec un nombre d’éléments colossal à afficher, donc la comparaison sur un site de cette ampleur serait nettement mieux représentée. Il est donc indispensable pour des applications de cette envergure de diviser le tout en deux parties distinctes, une partie mobile et une partie desktop.

Nous avons donc développé deux autres sites web, une version pour mobile et une autre pour desktop. Ces deux versions sont régies par un script PHP qui nous redirige automatiquement vers l’un ou l’autre par détection du device.

```php
<?php
$iphone = strpos($_SERVER['HTTP_USER_AGENT'],"iPhone");
$android = strpos($_SERVER['HTTP_USER_AGENT'],"Android");
$palmpre = strpos($_SERVER['HTTP_USER_AGENT'],"webOS");
$berry = strpos($_SERVER['HTTP_USER_AGENT'],"BlackBerry");
$ipod = strpos($_SERVER['HTTP_USER_AGENT'],"iPod");

if ($iphone || $android || $palmpre || $ipod || $berry == true)
{
    header('Location: ./indexM.php');
}
?>
```

Figure 6: Exemple de redirection PHP

Si le device détecté est de type mobile (iphone, android, blackberry, …) nous redirigeons vers notre page mobile, mais si le device détecté est de type desktop, dans ce cas nous restons sur la page desktop.

Ainsi en version desktop nous sommes redirigés vers :

Figure 7: Exemple de lien desktop

Alors qu'en version mobile nous sommes redirigés vers :

Figure 8: Exemple de lien mobile

La tâche a été différente et plus ou moins compliquée entre VanillaJS et angularJS. En effet grâce à angular, les contrôleurs et les vues sont restés identiques grâce au data-binding, il nous a simplement fallu retirer les media queries et garder seulement le code dédié à chaque plateforme. En ce qui concerne VanillaJS, la modification du HTML par le javascript n'étant pas gérée, il nous a fallu réécrire la plupart des fonctions afin de pouvoir renvoyer le code html correct adapté au device. On remarque ainsi la puissance d'un framework en comparaison a du VanillaJS pour la réadaptation de code en version mobile ou desktop pur.

AngularJS : Tutorial

Qu’est-ce que AngularJS et pourquoi l’utiliser ?

Angular est un framework, mais qu'est-ce qu’un framework ?
“c’est un ensemble d’outils et de composants logiciels à la base d’une application, il en compose son squelette”

AngularJS est donc un framework développé par google et qui a pour vocation de rendre lisible du code javascript html et css. Angular a donc été développé par google et pour google. Il permet donc de simplifier grandement l'écriture du javascript et permet d'intégrer des nouveaux types au html. Le code est ainsi plus clair, plus lisible et donc bien plus facilement maintenable.

Pourquoi l’utiliser ?

Premièrement, lorsque l’on a les bases du web, la prise en main d’angular est assez simple. De plus sa particularité principale permet de développer des pages web dynamiques de manière largement simplifiée par rapport a du vanillaJS. En effet angular a pour principale caractéristique le data-binding, ou la synchronisation en temps réel entre le modèle et la vue (entre $scope dans le controlleur et le page html). Ainsi quand notre scope change, notre vue change également.
De plus, angular nous permet d’utiliser des directives, qui sont des éléments d’interfaces graphiques (préfixés par ng-) nous permettant de lier les éléments du scope par l'intermédiaire du contrôleur.

En résumé, angular permet de rendre lisible et maintenable un projet, de grande ampleur ou non. Comme tout framework, angular a ses avantages mais aussi ses défauts, il permet la réutilisabilité et la facilité de développement, nous pouvons donc réutiliser nos contrôleurs ainsi que nos scopes dans nos vues, cependant comme tout framework, l’un de ses plus gros défauts est la perte de performances. Ainsi en comparaison a du vanilla JS, angular sera plus facile à adapter et à réutiliser mais du point de vue des performances, angularJS sera nettement moins bon que VanillaJS (Nous en reparlerons plus tard dans le tutoriel).

Installation et configuration d’AngularJS

Dans le cadre de ce projet nous avons utilisé Yeoman pour mettre en place notre squelette. Notre projet sera implémenté simplement avec Angular, aucune bibliothèque (ex: bootstrap) ou module ne seront ajoutés.
Afin d'installer angular, il faut que node.js soit installé sur notre machine.

Il nous suffit ensuite d'ouvrir une invite de commandes et d'entrer les commandes suivantes :

- npm install -g bower
- npm install -g yo
- npm install -g generator-angular

Puis de générer le projet avec :

- yo angular

A ce stade, Yeoman nous laisse le choix des modules à installer, nous avons donc le choix d'inclure ou non bootstrap ainsi que certaines dépendances. Nous avons fait le choix de n'inclure que le strict minimum, c'est à dire Angular et rien d'autre.

Ainsi notre projet se décompose de la façon suivante :

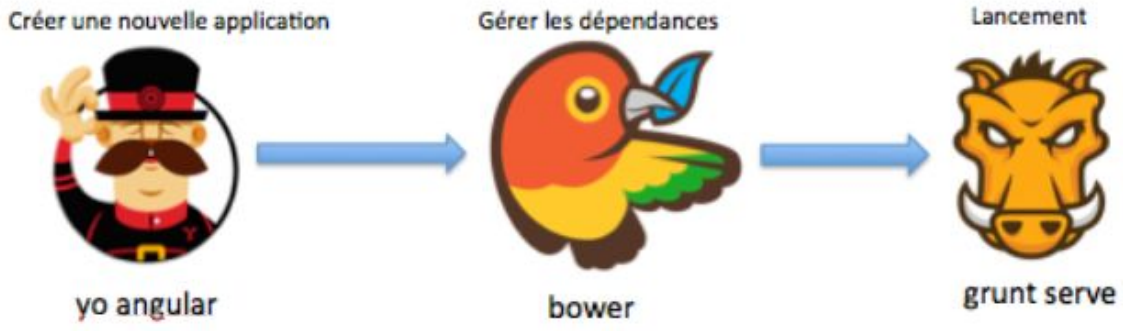


Figure 9 : Récapitulatif de la création du projet

(Bower n'est autre qu'un outil proche de NPM mais utilisé lors du développement front-end)

Une fois notre squelette, généré par yeoman, nous pouvons voir que l'architecture de ce dernier est plutôt claire : nous avons notre app dans laquelle se trouve un dossier view où les pages html se situent, un dossier scripts qui contient nos controllers et de même pour les feuilles de styles.

Un ensemble tout à fait clair qui nous permet de le maintenir de façon optimale.

Comparaisons des performances et analyses

1. WebPageTest

Nota : Nos tests ont été effectués en mode 3G Slow afin de tester le pire scénario utilisateur.

a. Responsive

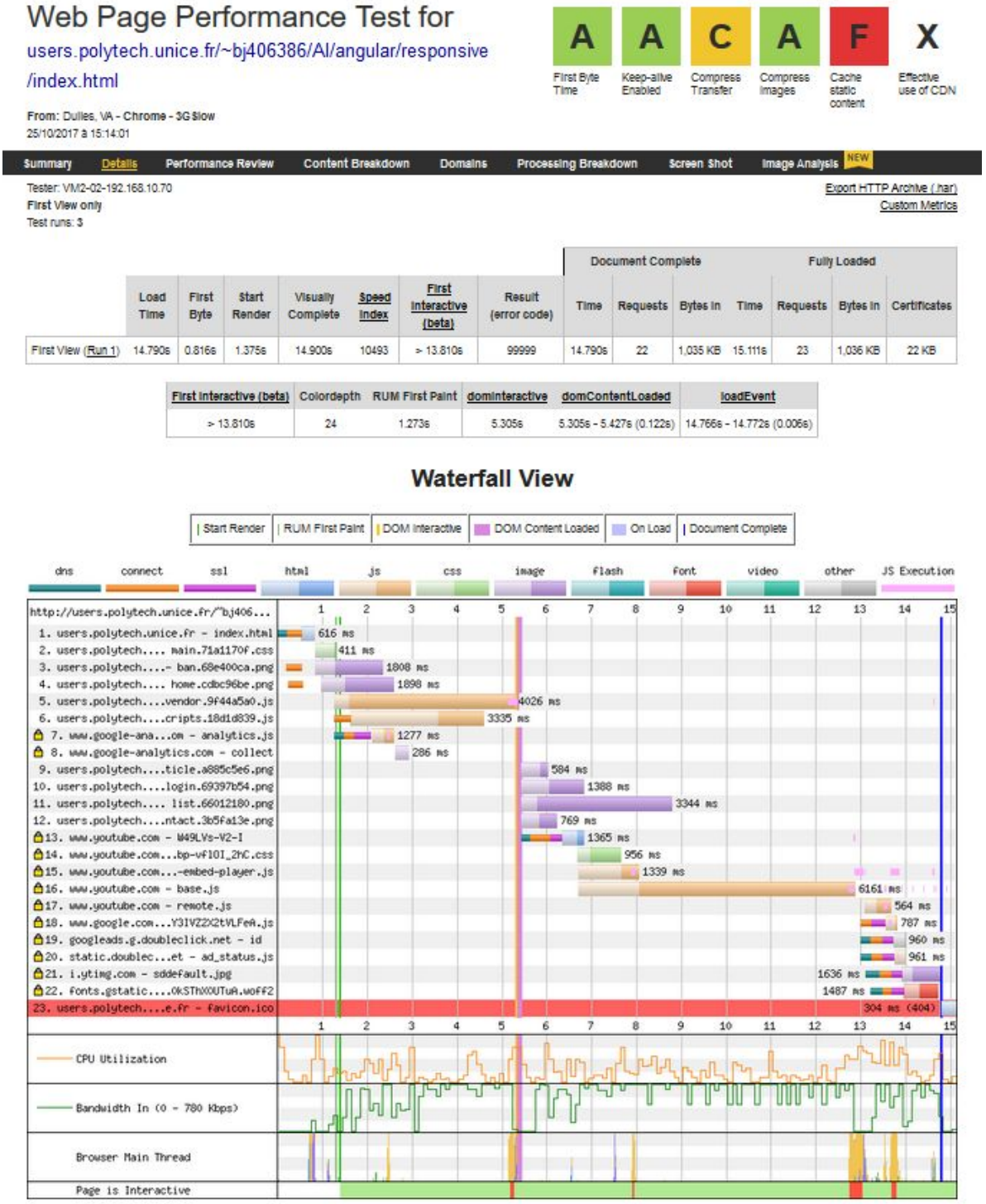


Figure 10: Test de la version responsive WPT

D'après une étude réalisée par google, 53% des utilisateurs quittent une page web si cette dernière met plus de trois secondes à s'ouvrir. De manière plus technique cela revient à tester le "critical render path" qui se doit d'être le plus petit possible mais surtout inférieur à trois secondes.

Ayant conscience de ces précieuses informations, nous avons donc développé de telle sorte que les chargements de css bloquant soient au maximum minimisés, nous avons optimisé la compression des images ainsi que l'affichage prioritaire du contenu visible.

Nous arrivons donc à des résultats plutôt satisfaisants en ce qui concerne le "start render" ou "critical render" qui est de 1.375 secondes. Cependant les performances "fully load" (chargement complète de la page) sont plutôt décevants : 15.11 secondes.

b. Adaptive

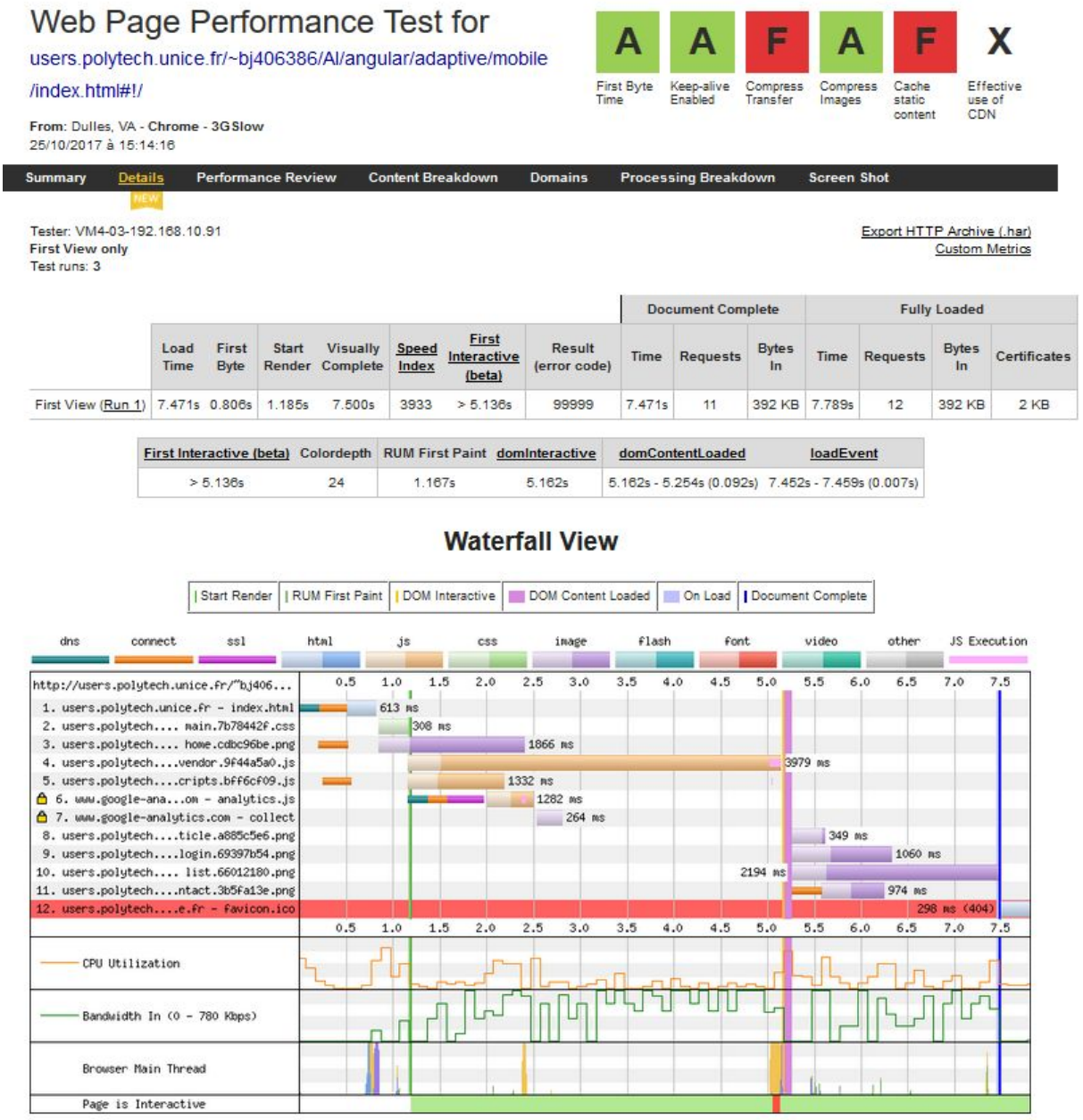


Figure 11: Test de la version adaptive WPT

Pour la version adaptive nous avons des résultats plutôt similaires en terme de "start render": 1.186 secondes. Par contre le "fully load" n'est dans cette version plus qu'à 7.789 secondes.

Ceci s'explique par le fait que nous n'avons pas besoin de charger des éléments inutiles; en effet sur le site mobile nous n'avons pas besoin de charger la partie desktop et vice et versa. Le CSS est donc réduit et de ce fait le CRP est un peu amélioré.

2. PageSpeed Insights
    a. Responsive

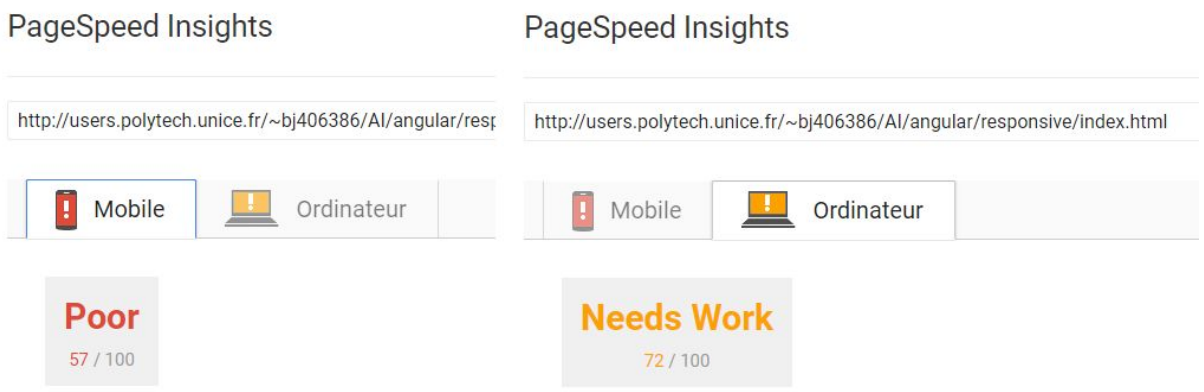


Figure 12: Analyse de la version responsive

PageSpeed Insights a pour but d'évaluer l'optimisation de notre page web. On remarque ici qu'en version responsive notre page web est plus optimisée pour ordinateur que pour appareil mobile, cela est dû au framework que nous utilisons. En réalité AngularJS rajoute une surcouche au moment du build nous permettant d'utiliser toutes les fonctionnalitées de ce dernier mais nous faisant perdre beaucoup en terme d'optimisation. Ainsi une device mobile aura plus de mal à afficher une page que notre Desktop.
En conclusion, aucunes de ces deux données n'est satisfaisante, bien au contraire.

    b. Adaptive

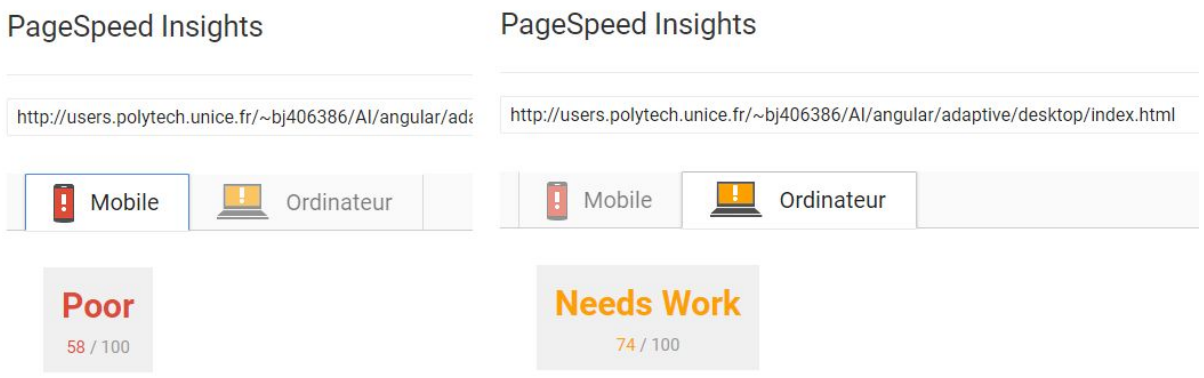


Figure 13: Analyse de la version adaptive

Afin d'optimiser les résultats précédemment obtenus, nous avons donc développé le même site web en adaptive design. Nous avons donc effectué les mêmes tests que précédemment pour en arriver à la conclusion suivante: nous ne gagnons quasiment rien de plus, au point de vue optimisation, à développer une version mobile et une version desktop avec angular. En effet, on voit clairement ici que du point de vue optimisation, c'est le framework qui nous bloque et que nous sommes dans les limites de ce dernier.

3. Comparaisons des données

Au vu des résultats obtenus dans les 2 précédents tests, nous pouvons conclure sur notre hypothèse de départ : déporter le site web en une version mobile et une version desktop plutôt qu'en une version responsive permet de gagner en performances. En effet le "critical render" est similaire car le css est un peu allégé. En revanche le "fully load" est quasiment divisé par deux, car en version responsive design, peu importe le device, nous devons charger les deux possibilités (mobile et desktop) alors qu'en version seulement mobile, seules les données du device en question sont chargées.
Cependant, du point de vue optimisation, nous ne gagnons rien de plus à développer une page mobile et une page desktop car nous poussons l'optimisation plus loin que le framework est capable de nous délivrer. En effet, une fois le build effectué, les scripts et les styles sont concaténés et compressés afin de gagner en taille, cependant le temps d'affichage global est réduit.

VanillaJS : Tutorial

Qu'est-ce que VanillaJS et pourquoi l'utiliser ?

VanillaJS est le framework le plus simple et le plus utilisé du monde car en réalité il n'y a pas de framework. VanillaJS est très léger car il n'y a pas besoin de rajouter une seule ligne de code pour le faire fonctionner. C'est clairement du javascript sans aucun ajout et donc le plus simple possible.

VanillaJS est tellement puissant que c'est un incontournable dans le monde du développement Javascript actuel. Facebook, Twitter, Amazon ou même Google, tous les plus gros utilisent ce framework pour sa puissance et sa vitesse incomparable avec les autres frameworks du marché. En effet, il est même à la base des jQuery, qui sont fait en Vanillajs. Mais nous pouvons nous demander s'il est vraiment si rapide que ce que nous prétendons ?

Voici quelques graphiques afin de voir plus explicitement ce que nous avançons.

	100 ops *Vanilla JS*	Code
Vanilla JS	100	document.getElementsByTagName("div");
Prototype JS	25	Prototype.Selector.select("div", document);
jQuery	21	$("div");
Dojo	3	dojo.query("div");
MooTools	2	Slick.search(document, "div", new Elements);

Figure 14:Retrouver 10 éléments du DOM par tag name

	100 ops *Vanilla JS*	Code
Vanilla JS	100	document.getElementById("vanilla");
Dojo	92	dojo.byId("dojo");
Prototype JS	57	$("prototype");
jQuery	42	$("#jquery");
MooTools	24	document.id("mootools");

Figure 15: Retrouver un élément du DOM par ID

Ces deux graphiques nous permettent de voir que les requêtes pour retrouver un élément sont plus rapides en vanillaJS. Ainsi nous pouvons en déduire que si VanillaJS est déjà beaucoup plus rapide que ses concurrents sur des opérations simples, alors dans les fichiers javascripts bien plus complexes, la différence de rapidité sera encore plus notable.

Name	angular-v1.6.3-keyed	angular-v4.1.2-keyed	ember-v2.13.0	react-lite-v0.15.30	react-v15.5.4-keyed	react-v15.5.4-mobX-v3.1.9	react-v15.5.4-redux-v3.6.0	vanillajs-keyed
create rows Duration for creating 1000 rows after the page loaded.	251.88.0 (1.8)	193.17.9 (1.4)	344.613.8 (2.5)	170.09.6 (1.2)	188.910.9 (1.4)	243.99.4 (1.8)	212.214.2 (1.5)	138.55.8 (1.0)
replace all rows Duration for updating all 1000 rows of the table (with 5 warmup iterations).	278.316.7 (1.9)	197.45.3 (1.3)	292.712.1 (2.0)	235.46.8 (1.6)	201.06.4 (1.4)	229.212.2 (1.5)	206.77.3 (1.4)	148.04.5 (1.0)
swap rows Time to swap 2 rows on a 1K table. (with 5 warmup iterations).	14.71.5 (1.0)	13.41.0 (1.0)	16.41.5 (1.0)	30.01.8 (1.9)	14.70.9 (1.0)	18.01.2 (1.1)	17.11.3 (1.1)	11.41.1 (1.0)
create many rows Duration to create 10,000 rows	3108.72162.2 (2.3)	1946.041.8 (1.5)	2569.356.3 (1.9)	2300.951.4 (1.7)	1852.429.0 (1.4)	2217.371.5 (1.7)	1931.735.6 (1.5)	1331.122.2 (1.0)
append rows to large table Duration for adding 1000 rows on a table of 10,000 rows.	454.842.1 (1.5)	324.610.1 (1.1)	524.223.6 (1.8)	2087.565.2 (7.1)	345.610.4 (1.2)	459.847.2 (1.6)	366.410.9 (1.2)	295.312.8 (1.0)
clear rows Duration to clear the table filled with 10.000 rows.	817.637.2 (4.7)	379.911.3 (2.2)	303.974.7 (1.7)	344.126.4 (2.0)	398.48.2 (2.3)	495.128.8 (2.8)	410.99.8 (2.4)	174.84.2 (1.0)
startup time Time for loading, parsing and starting up	118.15.1 (2.9)	84.32.6 (2.1)	245.25.6 (6.0)	44.92.6 (1.1)	70.02.9 (1.7)	87.64.3 (2.2)	93.86.9 (2.3)	40.59.5 (1.0)
slowdown geometric mean	**2.08**	**1.45**	**2.10**	**1.92**	**1.42**	**1.74**	**1.56**	**1.00**

Figure 16: Retrouver un élément du DOM par ID

Nous pouvons faire la même analyse mais maintenant avec des framework concurrents. Nous pouvons encore une fois voir que VanillaJS se démarque de tous les autres framework mais surtout de Angular.

En outre, un énorme avantage est qu'il est compatible avec tous les navigateurs déployés. Cependant nous n'avons ici présenté que les avantages de notre outil. N'y a-t-il vraiment que des avantages ? Bien entendu que non. Le fait de ne pas utiliser de framework a pour gros inconvénient de ne pas avoir les avantages de ses concurrents. En effet, il faut une très bonne organisation pour utiliser vanillaJS car nous perdons, comparé à AngularJS, l'organisation simple des contrôleurs et la maintenabilité qui en découle.

Comparaisons des performances et analyses

1. WebPageTest

Nota: Nous tenons à préciser que tous nos tests ont été effectués en 3G Slow afin de prendre l'un des pires scénarios possibles pour l'utilisateur.

a. Alpha

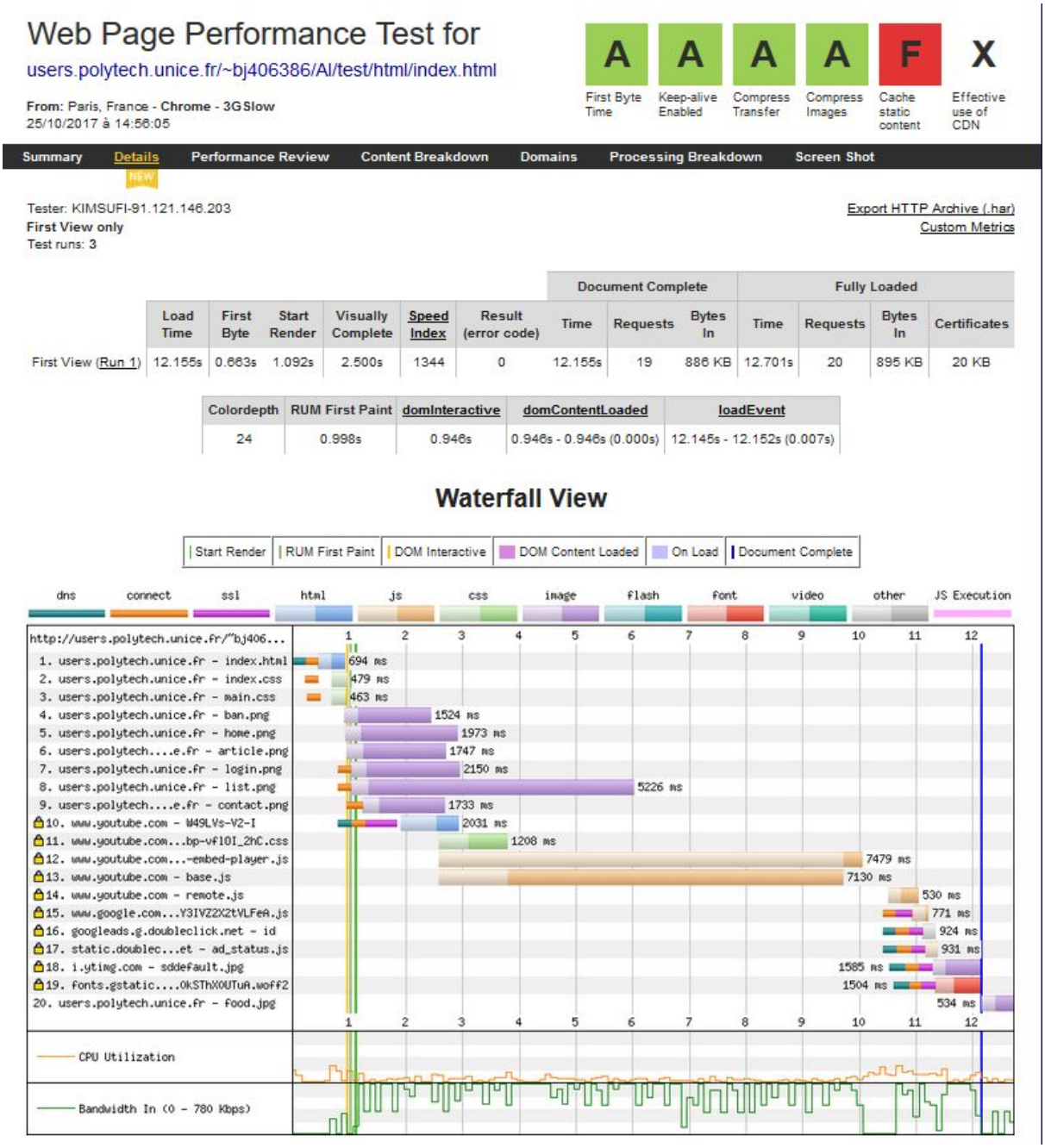


Figure 17: Test de la version alpha WPT / Critical Render Path

D'après une étude de Google, 53% des personnes abandonnent le chargement d'une page si celui-ci dure plus de trois secondes. De ce fait le paramètre le plus important lors de notre développement était de rester en dessous de ce seuil pour le critical render path (CRP).

Ce graphique illustre notre première version de tests pour le site web. Nous pouvons voir que le CRP est respecté car il est à environ une seconde et donc bien en dessous de la valeur fatidique des trois secondes.Cependant ces valeurs ne sont pas valables pour les

comparer avec le projet final car le développement et l'ajout de fonctionnalités n'étaient pas encore finis (première démonstration).

b. Responsive

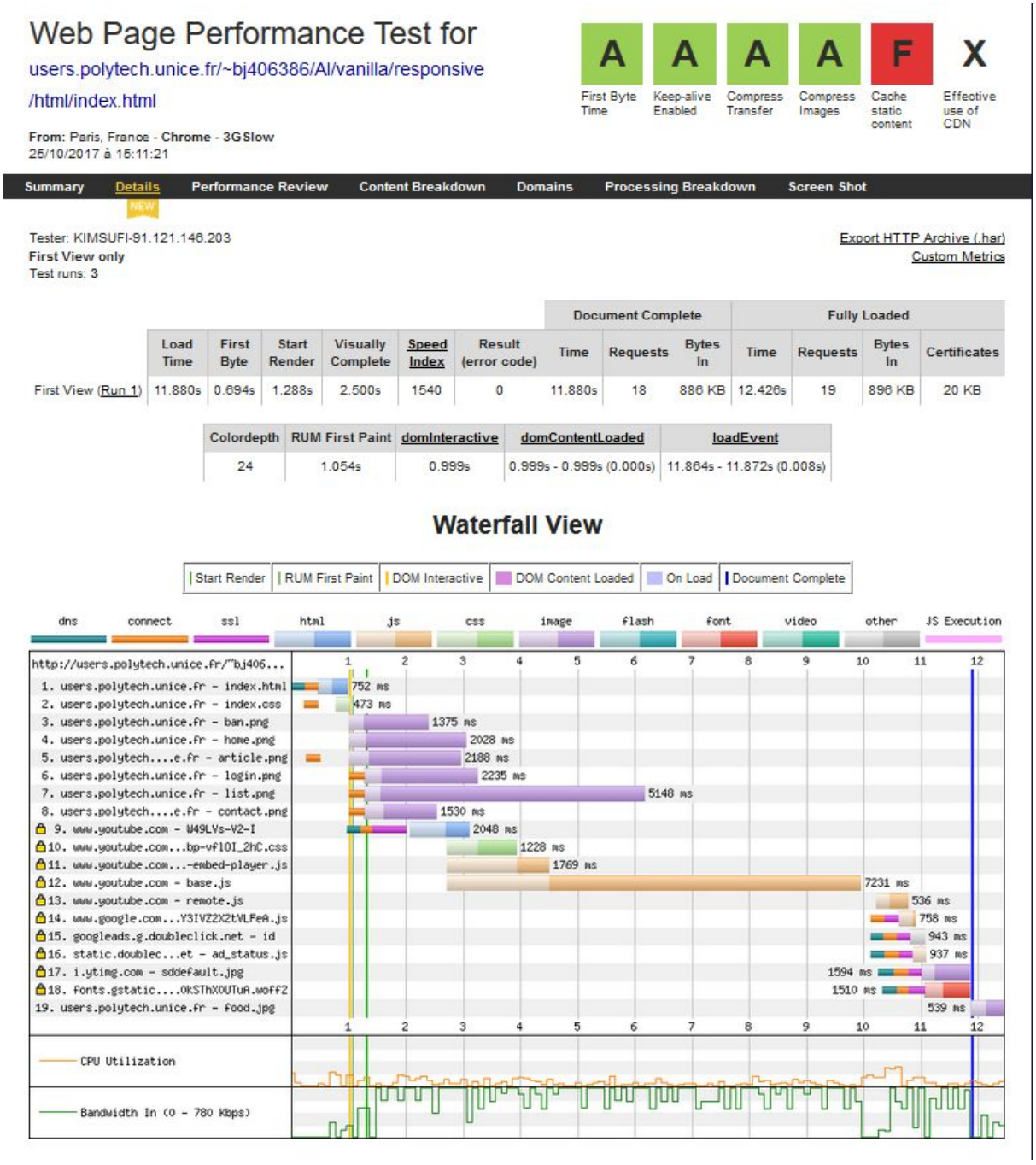


Figure 18: Test de la version responsive WPT / Critical Render Path

Ce graphique représente les tests de notre version finale au niveau du responsive design. Nous pouvons encore une fois voir que le CRP est encore respecté. Ceci est dû au fait que nous y avons ajouté quelques optimisations. En effet, le CSS a été épuré: il n'y plus de CSS bloquant et le CSS minimal est présent dans le html. De plus nous avons compressé les images mais en gardant suffisamment de détails pour qu'elles puissent

s'afficher du mobile jusqu'au desktop. Le Fully Loaded est à 12 secondes: ceci est bien trop grand car nos images ne s'affichent pas même si le site reste fonctionnel.

c. Adaptive

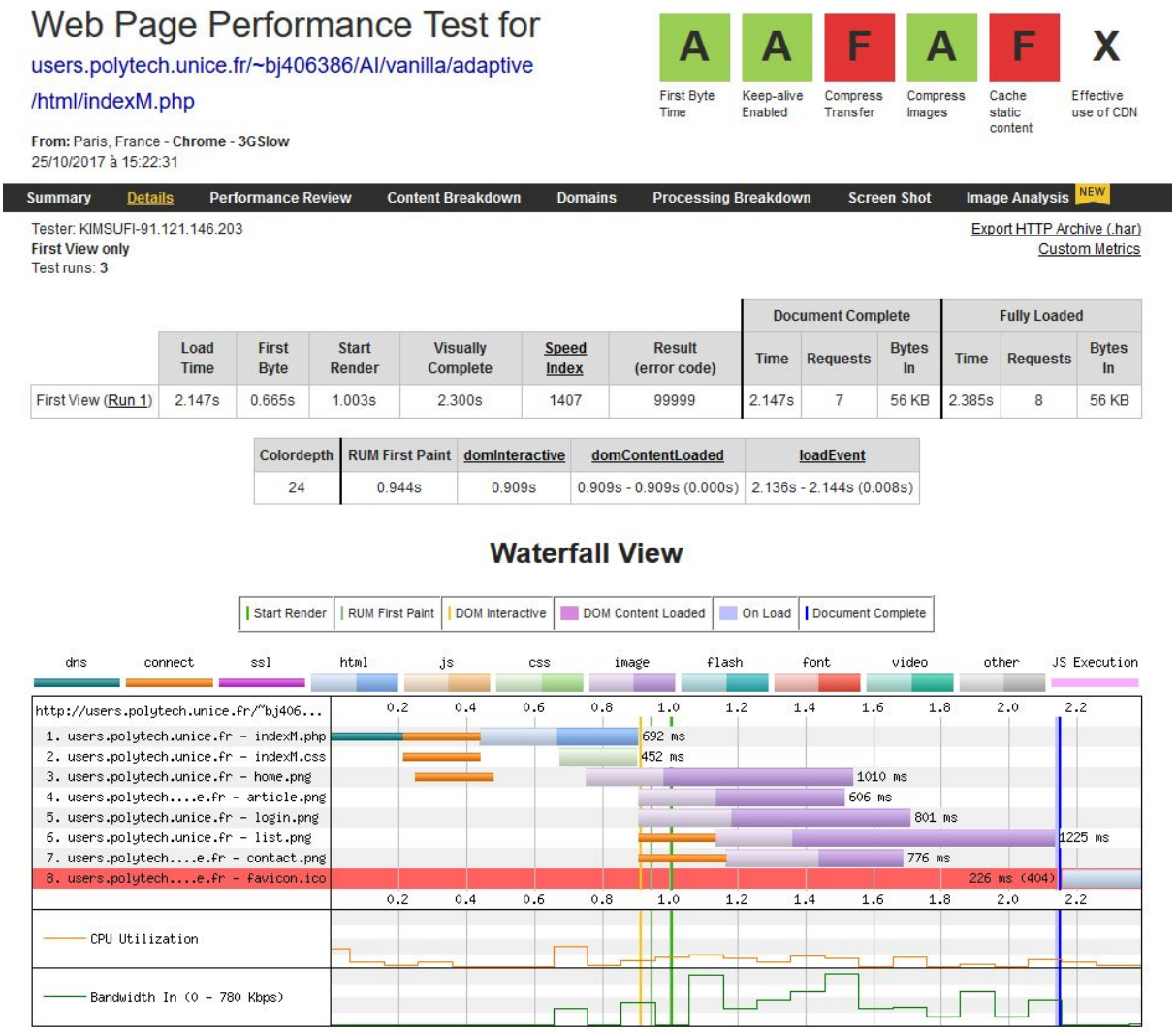


Figure 19: Test de la version adaptive WPT / Critical Render Path

Ce graphique représente les tests faits sur la version finale et adaptive de notre site web pour la version mobile. Dans cette version, le CRP est vraiment au plus bas car le code minimal est optimisé au maximum. En effet, il n'y a plus de CSS bloquant et les images sont compressées au maximum. De plus, vu que nous sommes sur une version mobile et qu'elle est dissociée de la version desktop, il n'y a plus tous les éléments du desktop qui sont chargés. Ainsi la page complète est chargée en moins de 2.4 secondes; plus rapidement que la valeur limite de trois secondes du CRP.

2. PageSpeed Insights

a. Alpha

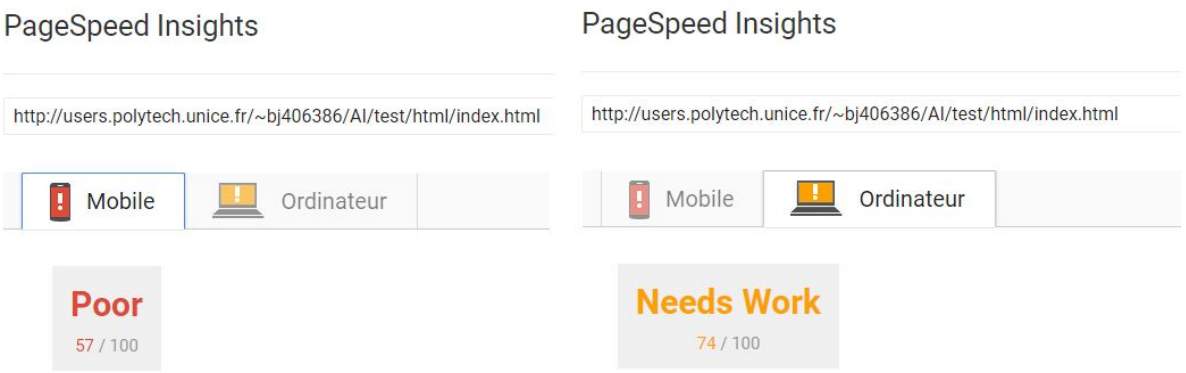


Figure 20: Test de la première version

Dans cette figure 20 nous pouvons voir la qualité de notre code au niveau de la première présentation. Il n'y avait absolument aucune optimisation. Seul le code était organisé, le CSS séparé du HTML. De ce fait nous avons un point de départ sur nos futures améliorations.

b. Responsive

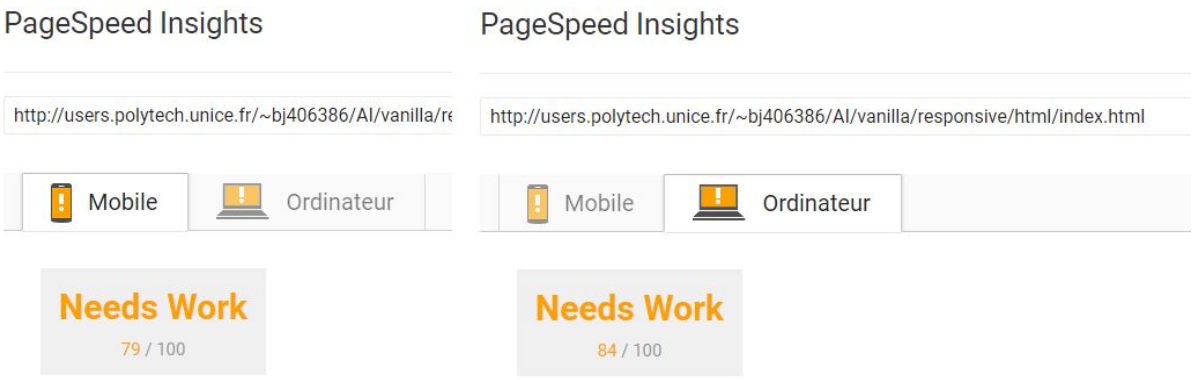


Figure 21: Test de la version responsive

Ici nous pouvons voir les résultats des améliorations par rapport à la figure 20. Dans cette version responsive, nous avons changé la disposition du CSS: la partie minimale du CSS à utiliser pour afficher le fond de la page est intégré dans le HTML afin de présenter un squelette de la page lors d'un chargement potentiellement long. De plus le javascript a été optimisé dans cette version afin de gagner en rapidité. Enfin les images ont été compressées pour réduire leur poids mais elles sont tout de même toutes chargées au chargement de la page, les images pour la version mobile ainsi que celles pour la version desktop.

c. Adaptive

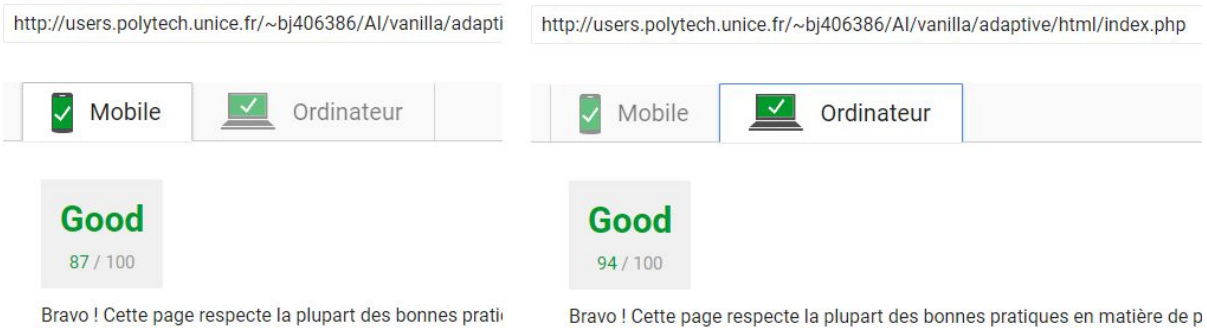


Figure 22: Test de la version Adaptive

Cette figure 22 montre les résultats de notre meilleure version, tout le projet confondu. Dans cette version, il n'y a pas de framework utilisé car nous sommes en vanillaJS. De plus nous avons poussé beaucoup plus loin la compression des images, jusqu'à même en changer certaines afin de gagner en espace. Enfin, seules les images de la version de l'appareil de l'utilisateur sont chargées au début de la page. De plus, ici le javascript a été encore une fois remanié afin de n'utiliser que les fonctions utiles à chacune des versions.

3. Comparaison des données

Pour faire la comparaison de ces trois données, nous pouvons déjà tout de suite mettre à part la version alpha. En effet, cette version nous a permis de voir que le développement du notre site était dans la bonne voie car le CRP était déjà respecté.

Ensuite sur les versions responsive et adaptive design nous pouvons voir que le CRP est bien respecté à chaque fois: environ 1.1 seconde, même si celui de la version adaptive reste un peu plus petit. Cependant lorsque nous regardons le chargement total de la page, nous voyons un changement colossal. En effet, la version responsive doit charger toutes les données même si elles ne sont pas forcément affichées. Cela ralentit considérablement le chargement complet de la page; par exemple, la vidéo youtube est chargée dans la version mobile alors qu'elle n'est pas affichée au vu de la taille de l'appareil. Dans la version Adaptive, nous avons choisi de faire une site mobile dédié pour ne pas avoir ce type de soucis. En effet, lorsqu'un utilisateur va sur une page du site web, il est redirigé vers le bon type de page en fonction de l'appareil sur lequel il ouvre le site.

De plus, la redirection avec le PHP nous a permis d'alléger notre code, surtout au niveau du CSS et des images qui sont notamment compressées dans un format très léger. Le javascript a été réduit à son minimum

Enfin le javascript a été continuellement amélioré tout au long du développement du site web. En effet, dans la version responsive, nous l'avons amélioré et épuré alors que dans la version adaptive le javascript a été scindé en deux parties: celle pour le mobile et celle pour le desktop.

Ainsi au vu de ces résultats, nous pouvons conclure que le fait de faire une version mobile et une version desktop semble être la meilleure des solutions. Le code est allégé et

facilement maintenable. Enfin les performances sont considérablement accrues au vu des résultats que nous avons pu avoir.

Comparaison des performances entre AngularJS et VanillaJS

Venons en à comparer les performances entre angularJS et vanillaJS; si nous nous basons sur le critical render path (CRP), les résultats sont plutôt bons. En effet nous sommes en moyenne à 1.2 secondes toutes versions confondues. Ainsi le contrat des moins de trois secondes de CRP est largement respecté. Cela est dû à l'optimisation de notre code et de l'ordre d'affichage des éléments de notre page. Pour ce qui est du chargement total, les écarts se creusent nettement entre nos deux versions. On constate que l'Angular responsive est presque 2 fois plus lent que le Vanilla responsive. Cela est dû au framework qui pollue notre code dans sa conception de la version à exporter. En effet ceci est imputé au fait qu'il concatène tous les styles et les scripts; de même, il apporte du javascript propre à son fonctionnement et à la lecture du code généré. La nette différence se joue au niveau de la partie mobile adaptative, car dans le cas d'angular, nous devons toujours charger notre framework, alors que le Vanilla ne charge que ce dont il a besoin: en mobile, le strict minimum. On arrive donc à des performances imbattables pour du Vanilla adaptive où la barre des moins de trois secondes est atteinte en chargement total de la page.

En ce qui concerne les optimisations, encore une fois la marge de manoeuvre est bien plus importante du côté de vanillaJS. En effet, le fait de compresser les photos, les styles et les scripts manuellement donne un résultat bien meilleur que la manière automatique d'angularJS. Cependant vanillaJS n'a pas que des avantages, comme le fait d'organiser le développement du site web; il faut une bonne gestion afin de rester dans le droit chemin alors que chez angular, la séparation du code permet une maintenabilité très facile du code.

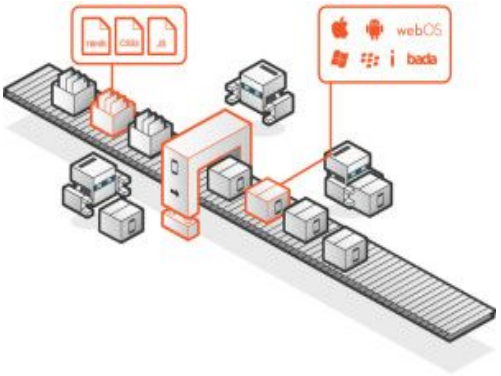
Enfin le fait de faire un site web dynamique entre parfaitement en adéquation avec le framework angular car il est conçu pour cela et les gère automatiquement. Bien entendu dans vanillaJS tout est possible mais le problème c'est qu'il n'y a aucune aide pour réaliser ses projets.

"*La simplicité peut être plus difficile à atteindre que la complexité : il faut travailler dur pour bien penser et faire simple. Mais cela vaut le coup à la fin parce qu'une fois que tu y arrives, tu peux déplacer des montagnes.*" **Steve JOBS**

Ionic 2 : Tutorial

Qu'est-ce que IONIC ?

Ionic est un framework de développement mobile cross platform qui permet de réaliser une application ayant pour destination les plateformes Android, IOS, Windows. Ce framework permet de réaliser des applications par le biais d'une WebView qui est proposé de façon native sur les appareils. Ce framework est un mélange entre une partie AngularJS pour la partie développement web et une partie Cordova pour le développement d'applications natives, le tout formera donc des applications hybrides.



La présentation d'AngularJS est faite dans la partie "AngularJs" du document.
Cordova est un framework développé par Apache Fondation, il permet d'accéder à certaines fonctionnalités natives du smartphone. Le tableau ci-dessous présente les différents capteurs et fonctionnalités natives que l'on va pouvoir accéder via ce framework.

	amazon-fireos	android	blackberry10	Firefox OS	ios	Ubuntu	wp8 (Windows Phone 8)	windows (8.0, 8.1, Phone 8.1)	tizen
Accéléromètre	✓	✓	✓	✓	✓	✓	✓	✓	✓
Niveau de la batterie	✓	✓	✓	✓	✓	✗	✓	✓ *Windows Phone 8.1 seulement*	✓
Appareil photo	✓	✓	✓	✓	✓	✓	✓	✓	✓
Capture Audio/Video	✓	✓	✓	✗	✓	✓	✓	✓	✗
Boussole	✓	✓	✓	✗	✓ (3GS+)	✓	✓	✓	✓
État de la connexion	✓	✓	✓	✗	✓	✓	✓	✓	✓
Répertoire de contacts	✓	✓	✓	✓	✓	✓	✓	partiellement	✗
Dispositif	✓	✓	✓	✓	✓	✓	✓	✓	✓
Évenements	✓	✓	✓	✗	✓	✓	✓	✓	✓
Fichier	✓	✓	✓	✗	✓	✓	✓	✓	✗
Transfert de fichier	✓	✓	✓ Pas supporté	✗	✓	✗	✓ *Pas supporté*	✓ Pas supporté	✗
Géolocalisation	✓	✓	✓	✓	✓	✓	✓	✓	✓
Internationalisation	✓	✓	✓	✗	✓	✓	✓	✓	✗
Navigateur Internet intégré	✓	✓	✓	✗	✓	✓	✓	utilise les iframe	✗
Lecteur multimédia	✓	✓	✓	✗	✓	✓	✓	✓	✓
Notifications	✓	✓	✓	✗	✓	✓	✓	✓	✓
Écrans de lancement	✓	✓	✓	✗	✓	✓	✓	✓	✗
Barre de statut	✗	✓	✗	✗	✓	✗	✓	✓ Windows Phone 8.1 seulement	✗
Stockage interne	✓	✓	✓	✗	✓	✓	✓ localStorage & indexedDB	✓ localStorage & indexedDB	✓
Vibration	✓	✓	✓	✓	✓	✗	✓	✓ * Windows Phone 8.1 seulement	✗

Figure 23: Tableau capteurs supportés par platform

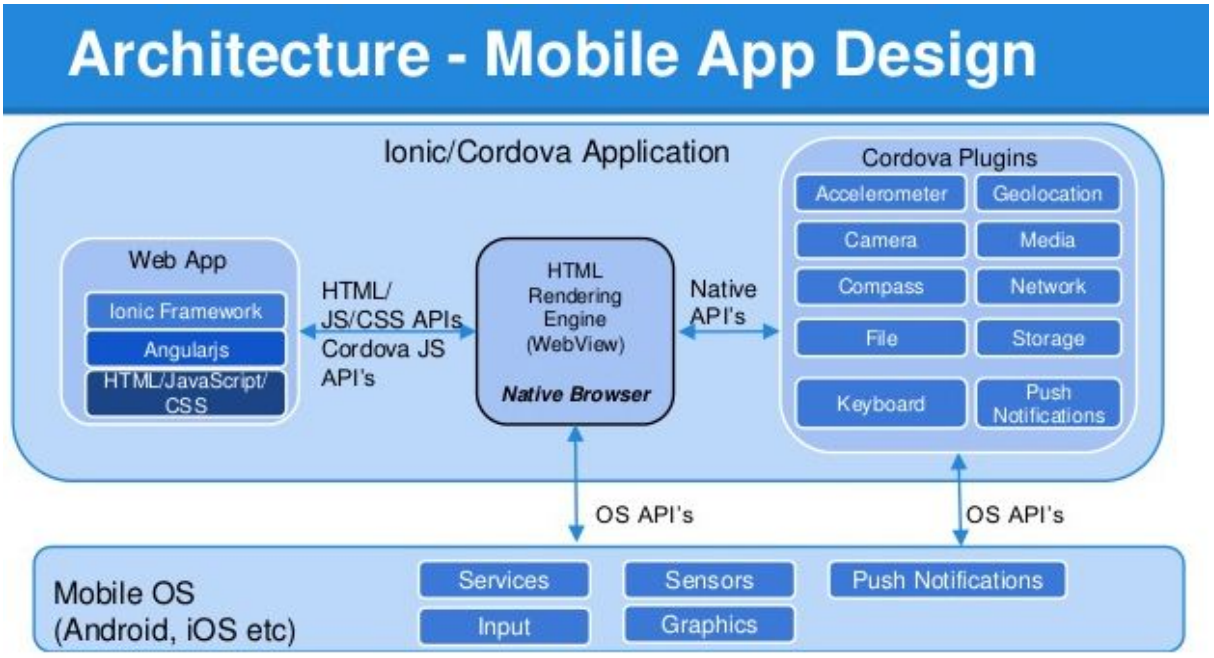


Figure 24: Système du framework Ionic

Ce schéma présente le système du framework Ionic: on peut voir à gauche la partie web de l'application et à droite la partie Cordova qui permet l'accès au capteur du smartphone. Le centre correspond à la webView qui permet de réaliser les applications sur les smartphones de différentes "platform".

Configuration et installation d'IONIC2

1. Commencer avec Ionic

Afin de pouvoir réaliser une projet avec le framework Ionic, l'utilisateur doit avoir installé au préalable Node.js afin de gérer l'installation de package NPM. Dans un second temps il faut installer ionic cordova afin de pouvoir créer un projet ionic. Ionic propose des types de projets pour démarrer notre projet, voir ci dessous.

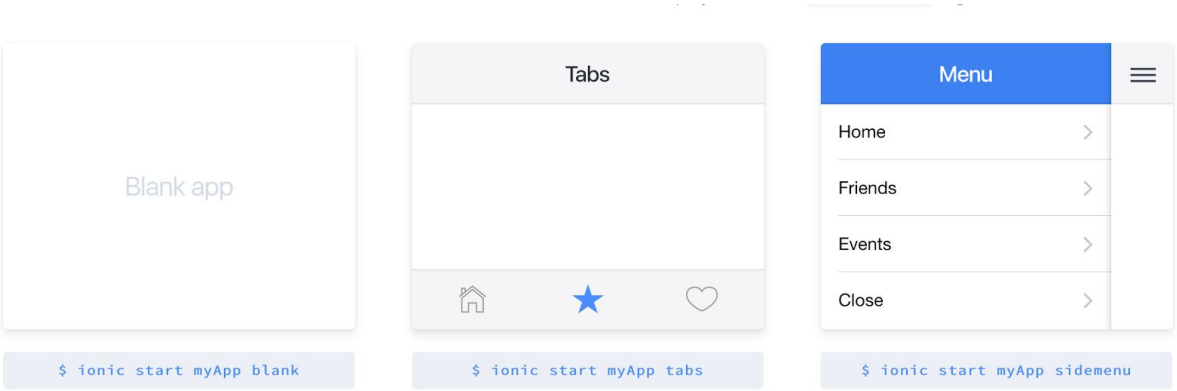
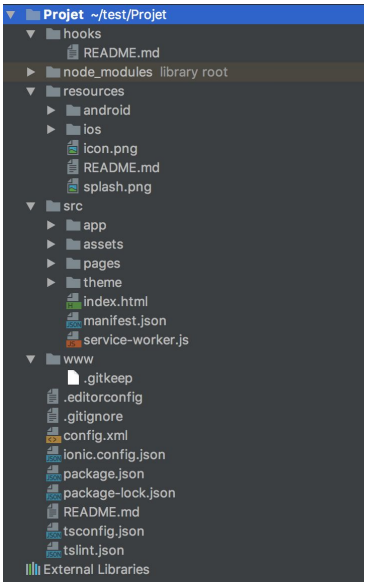


Figure 25: Types de projets proposés par Ionic

Un projet “blank” permet de réaliser le projet le plus simple disponible avec une arborescence de projet minimal, voir l’arborescence ci-dessous. On peut réaliser un projet “tabs”, c’est celui que nous avons utilisé pour réaliser l’application dans la mesure où cela correspondait parfaitement à ce que l’on souhaitait implémenter. Enfin une application avec “sidemenu”, elle comprendra un menu comme présenté ci-dessus, ce type de projet permet de réaliser une application avec des sous-menus, par exemple pour une application de shopping.
Le projet peut être créé à présent via la commande “`$ ionic start name`“ maintenant il suffit de rentrer dans le projet et de rentrer la commande “`$ ionic serve`” afin de lancer le projet sur le navigateur web.

2. Arborescence d’un projet Ionic

Lors de la création du projet, ionic réalise une génération et une arborescence de fichiers.



Dans le projet on peut remarquer plusieurs dossiers, les principaux sont les sont dans le dossier “src” on travaillera principalement dans le dossier app, page etc.

Le projet pourra être déployé sur un émulateur de devices, enfin le projet peut être compilé et déployé sur une plateforme android, cependant pour déployer le projet sur un IOS, il faut que l’utilisateur possède un Macbook avec un certificat et Xcode afin de pouvoir téléverser le projet sur ce dernier.

Figure 26: Architecture projet Ionic

Développement de l’application

1. Le projet FoodFactory

Pour le développement de l’application “FoodFactory”, nous avons créé le projet en utilisant un type de projet proposé par le framework, qui est le projet “tabs”, dans la mesure où ce dernier nous permet d’avoir un visuel direct sur les informations sans avoir besoin de passer par un menu, il correspond donc parfaitement à nos attentes pour le type d’application que nous souhaitions réaliser. Dans Ionic pour la création de nouvelles pages il faut le faire en ligne de commande via “`$ ionic generate [type] [name]`“, il est nécessaire de le faire de cette façon dans la mesure où des liens sont ajoutés et permettent une reconnaissance de la nouvelle page.

2. Le stockage de la liste de courses

Pour le développement de l'application dans un dossier provider, nous retrouvons notre liste d'articles, ils seront stockés avec leur code barre et le nom de l'article. Le code barre sera utilisé à des fins de recherches d'informations nutritionnelles.

3. Résultat de l'application

Dans le provider une classe a été implémentée pour permettre l'appel à un service extérieur pour la recherche de produit via le code barre de ce dernier.
Pour la partie adaptabilité de l'application elle est sur le scanner du code barre du produit via la caméra du smartphone, il n'y a pas de réelle différence entre l'utilisation de l'application en mode portrait ou en mode paysage.

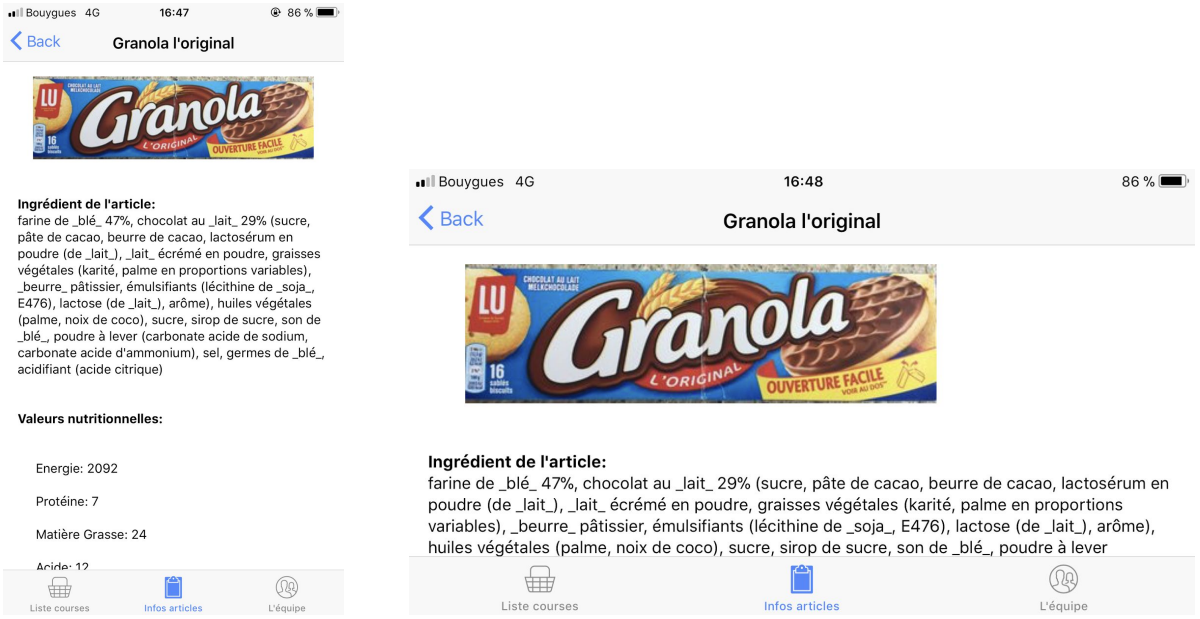


Figure 26: Présentation application paysage / portrait

4. Différence entre "platforms"

La différence se fait en revanche via l'accès au capteur du device qui peut se faire différemment; par exemple pour le lecteur d'empreintes il faut plusieurs plugins, un premier pour IOS, un pour samsung et un dernier pour les android par exemple. Cela peut être problématique dans la mesure où ionic est justement là pour faire un code similaire, il existe certaines choses propres à chaque plateforme ou à une marque particulière.

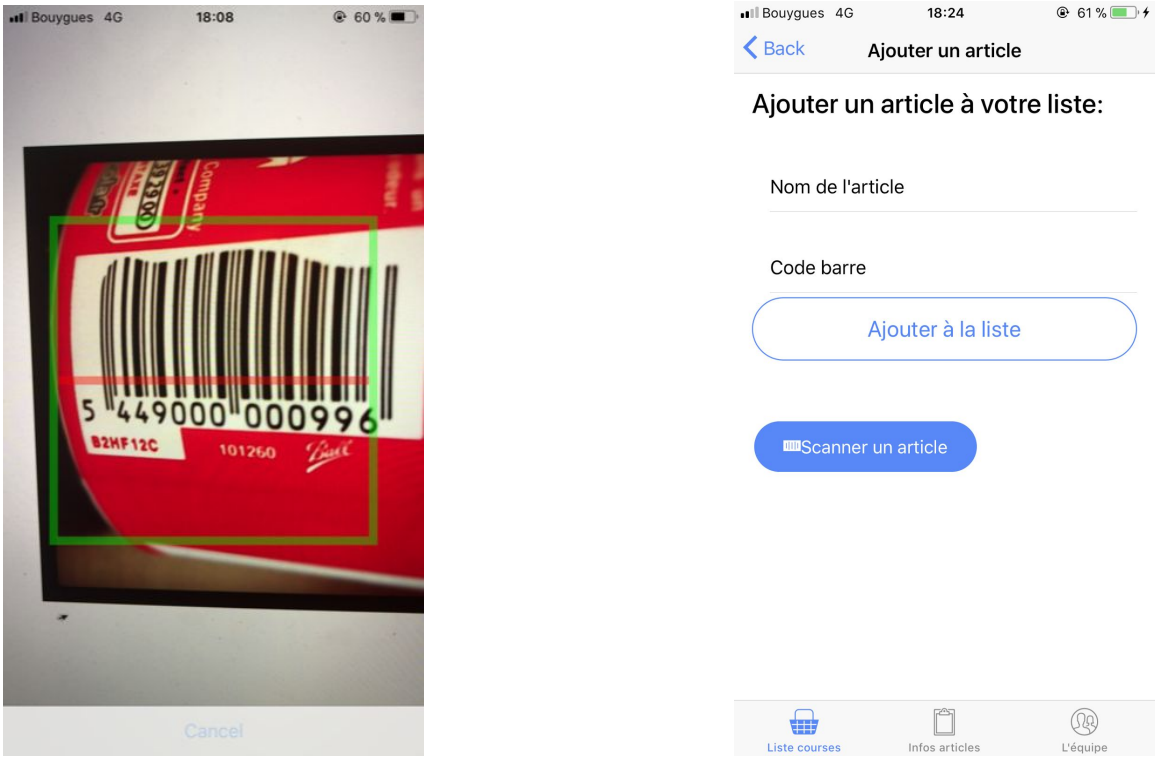


Figure 27: Fonctionnalités d'ajout d'un article à la liste

Pour l'ajout d'un article, il y a deux solutions, soit passer par le scanner, cela implique que l'utilisateur doit entrer le code barre à la main et le nom de l'article, soit il scanne le produit et tout est automatiquement pris en compte. Cependant l'ajout se fait cette fois- ci via le même plugin que nous présenterons dans la suite du rapport.

Les services sous IONIC 2

1. Appel vers une API extérieure

Dans notre application nous utilisons un service extérieur qui nous permet par le biais de la caméra de recueillir des informations. Nous utilisons l'API Open Food Facts, elle permet, via une requête GET avec le code barre, de récupérer un Json contenant toutes les informations relatives à un produit donné et donc de pouvoir l'ajouter à notre liste de courses. Afin de communiquer avec cette API, il faut réaliser une class de service qui permet d'interroger l'API lors de l'appel de la fonction de recherche.

```
@Injectable()
export class ArticleServiceProvider {

  constructor(public http: Http) {
  }

  searchArticle(codeBarre) {
    var url = 'http://fr.openfoodfacts.org/api/v0/produit/' + codeBarre + '.json';
    var response = this.http.get(url).map(res => res.json());
    return response;
  }
}
```

Figure 28: Class ArticleServiceProvider

Pour réaliser la recherche du produit on a simplement besoin du code barre, on a récupéré le lien fourni par l'API, on concatène avec le code barre et on sait que le format retourné est un Json. Il nous suffit donc tout simplement de récupérer les informations dont nous avons besoin pour l'utilisateur.

Accès aux capteurs

1. Utilisation d'un plugin

Dans notre application nous utilisons le capteur de la caméra afin de scanner un code barre, pour cela un plugin existe, c'est "Barcode Scanner"; ce plugin permet de lancer la caméra et de scanner un code barre pour en récupérer les numéros. Afin d'ajouter ce plugin à notre projet il faut réaliser deux étapes:

- la première consiste à télécharger le plugin cordova correspondant au lecteur de code barre, puis de le sauvegarder.
- la seconde étape consiste à l'ajouter à notre application afin de planifier le lancement du lecteur dans l'application.

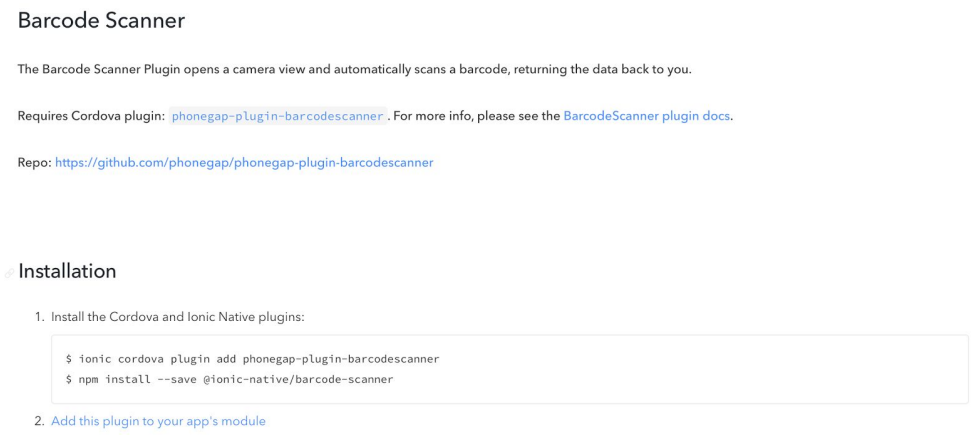


Figure 29: Plugin Barcode Scanner avec son installation dans le projet

De plus un point positif à ce plugin est qu'il fonctionne sur tous les dispositifs que ce soit:

- Android
- BlackBerry 10
- Browser
- IOS
- Windows

Ce plugin est donc parfait pour les différents appareils et ne nécessite pas d'adaptation pour le faire fonctionner en fonction des plateformes.

Barcode Scanner: https://ionicframework.com/docs/native/barcode-scanner/

Comment gérer l'adaptation aux dispositifs sous Ionic 2 ?

1. Utilisation d'un plugin cordova en fonction des platforms

L'implémentation est la même pour les dispositifs concernant la partie webApp (voir le schéma Ionic) étant donné que c'est du code HTML, JS, etc; cependant l'accès au capteur des dispositifs peut se faire via différents plugins en fonction de la compatibilité de certains.
Précédemment nous avons vu que le plugin "Barcode Scanner" était compatible avec toutes les plateformes, cependant si l'on veut avoir un lecteur d'empreintes dans notre application, on va trouver différents plugin: le premier s'appelle "Touch Id", cependant l'on constate que ce plugin est supporté seulement par les smartphone IOS. Le plugin qui correspond à la même action mais pour Android se nomme "Android Fingerprint Auth" celui-là est supporté simplement par les platform Android et non pas IOS, on se retrouve donc avec 2 plugins pour 2 "platform" différentes.

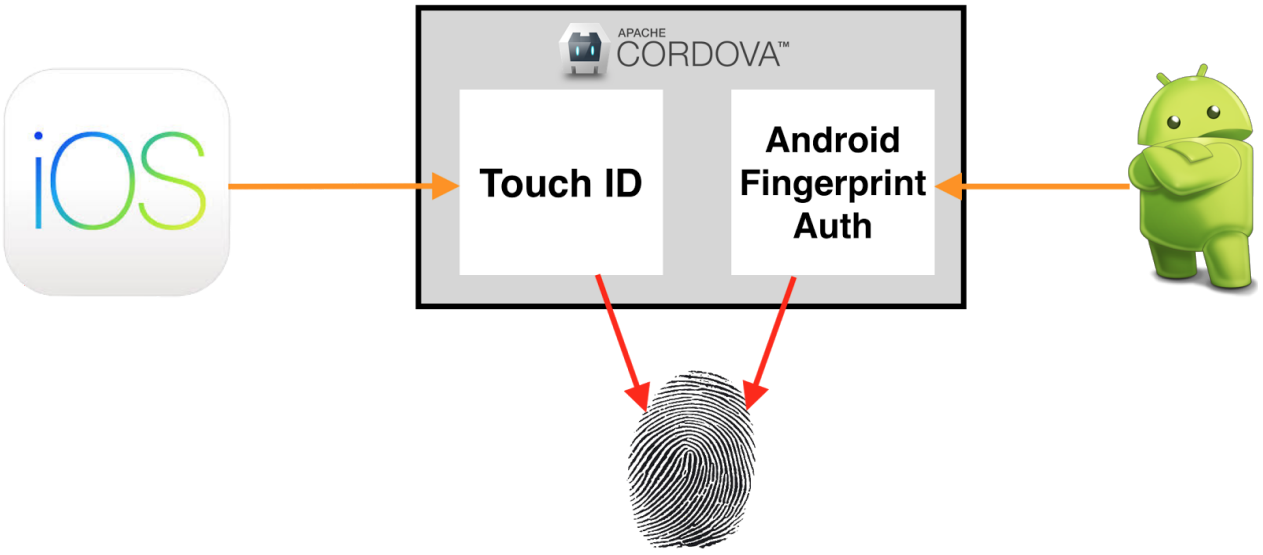


Figure 30: Schéma pour ajouter un lecteur d'empreinte

Cela n'est donc pas optimal dans la mesure où il faut réaliser une vérification du device en fonction pour le bon plugin. De plus pour le rendu visuel, des tests doivent être réalisés afin d'observer les différences et corriger si une erreur bloquante est visible (ex: emplacement bouton problématique ou autre).

Déploiement sur cibles réelles

1. Déploiement sur différents appareils

L'application a été testé sur un samsung A5, un iPhone 6s Plus, un iPhone 7 plus, un iPad et enfin sur une Galaxy Note 10.1. Les résultats sont concluants, les seules latences

que l'on peut rencontrer sont au niveau de l'appel à l'API qui retourne un JSON, dans les autres cas rien n'est à signaler, un petit test a été effectué afin de visualiser les performances sur un iPhone 7 plus.
Ci-dessous la page d'accueil sur un appareil IOS et sur un appareil Android.

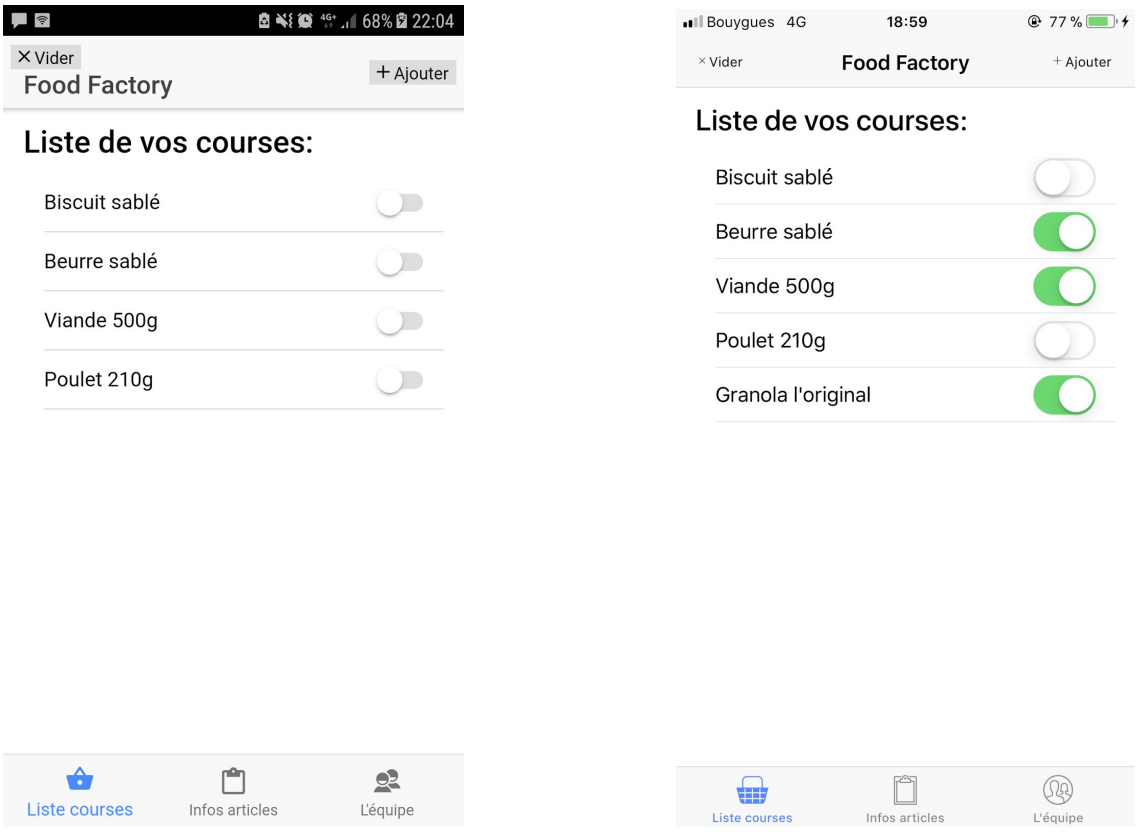


Figure 31: Application Android vs IOS

Nous pouvons apercevoir que la page d'accueil est relativement similaire entre les deux platform, cependant on peut distinguer une différence plus importante au niveau du bouton "vider", nous avons jugé que cette différence n'était pas dérangeante pour l'utilisation de la fonctionnalité. Cette différence est liée au fait que chaque platform a des "normes", ici le titre est à gauche sur Android et au centre pour IOS, auparavant le tab du bas était positionné en haut sur un appareil Android, les adaptations graphiques sont automatiquement réalisées, cependant, il faut vérifier que ce ne soit pas dérangeant pour l'utilisation de l'application.

2. Test de performance

Un petit test a été réalisé afin de visualiser les performances au repos et en activité de l'application:

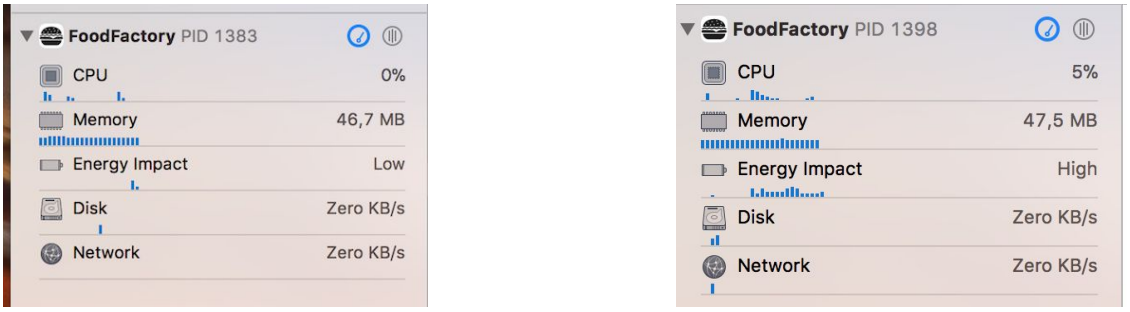


Figure 32: Performance de l'application repos / requête

Nous proposons deux images, celle de gauche correspond à une consommation après le lancement de l'application (ex: sur la page d'accueil), l'image de droite correspond elle à une consommation lors de la demande d'informations concernant un produit.
On remarque que la consommation du CPU est très basse pour les deux avec une hausse de 5% qui est normal dans la mesure où une requête GET est réalisée, la consommation de mémoire est relativement grande, une optimisation à ce niveau- là pourrait être réalisée afin de moins consommer encore. On remarque que l'énergie de la batterie est basse lorsque l'application est au repos et forte lors de la demande d'information, cela est normal car l'utilisation d'internet est un élément qui consomme le plus sur un appareil. Enfin on peut remarquer que le Network a eu un léger pic lors de la requête à l'API.

Conclusion sur Ionic

En conclusion, Ionic est un excellent moyen de réaliser des applications cross platforms; cependant si l'on souhaite accéder aux capteurs de l'appareil, il faut vérifier que les plugins d'utilisation soient supportés par toutes les platforms ou s'il est nécessaire d'en trouver plusieurs afin de couvrir la plus grande majorité des utilisateurs (IOS, Android). Quelques optimisations peuvent être réalisées; de plus beaucoup de plugins existent pour les différentes plateformes afin de pouvoir accéder à un maximum de capteur de chaque appareil.

Xamarin : Tutorial

Qu'est-ce que Xamarin?

Xamarin est un framework de développement mobile cross-plateforme. Il permet tout simplement de développer son application native sur IOS, Windows Phone et Android en un seul projet en n'utilisant qu'un seul langage de programmation, le C#. C'est une technologie rachetée par Microsoft Corporation en 2016. Cette technologie est réputée pour égaler les performances des applications mobiles natives (comme java pour Android, ou Objectif C/Swift pour iOS). Chaque méthode d'un SDK (Android, iOS, Windows Phone) est encapsulée à l'identique en C#. Sur chaque plateforme, toutes les fonctionnalités propres à la plateforme sont conservées et permettent aux développeurs android, iOS et Windows Phone de ne pas se perdre dans l'appel de fonctions ou de classes. Il est principalement utilisé par les entreprises dans le but de réaliser des applications complexes. Ce framework peut être utilisé au sein de Visual Studio, mais Xamarin possède aussi son propre IDE Xamarin Studio basé sur Mono Develop. Xamarin est un framework payant à moins de posséder une licence mensuelle ou annuelle.

Configuration et installation Xamarin

Pour la mise en oeuvre de l'application de FoodFactory à l'aide de Xamarin, nous devons télécharger l'IDE Visual studio (sur Windows) et Xamarin Studio (sur Mac). Une fois cela fait nous devons créer un projet cross-plateforme.

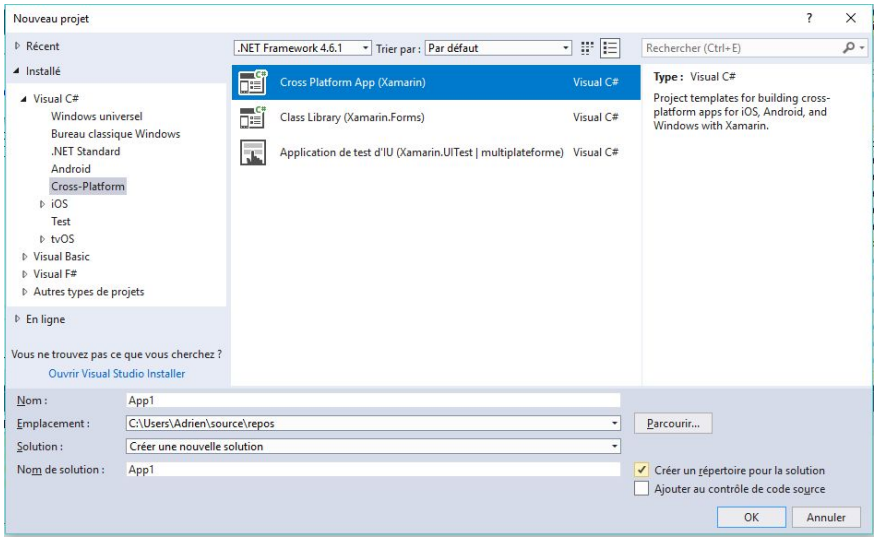


Figure 33 : Choix de projet Visual studio

Nous choisissons une application cross-plateforme en C#.

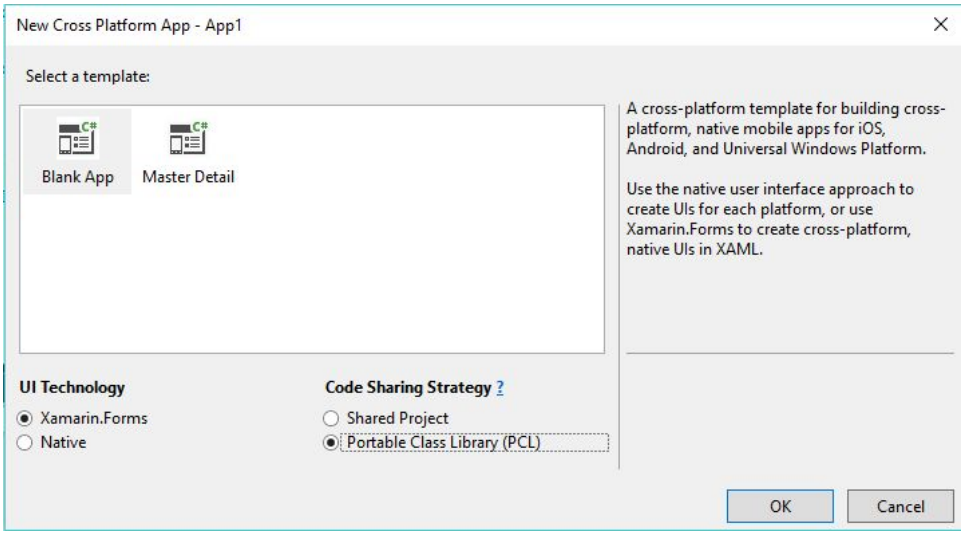


Figure 34 : choix de type de projet Visual studio pour Xamarin

Xamarin est composé de 2 types de technologie d'interface et de 2 stratégies de partage de code de projet. Nous allons utiliser une technologie d'interface utilisateur "Xamarin.Forms" et non native car nous voulons une application qui soit cross-plateforme. En effet, une application "Xamarin.Forms" est destinée à une application qui met en avant le partage de code plutôt qu'une interface utilisateur spécifique pour chaque plateforme. Il permet d'utiliser davantage le langage Xaml qui est utilisé dans les vues de l'application (avec le "code-behind"). De plus, ce langage permet d'avoir du code maintenable ce qui permet de modifier et d'intégrer des composants facilement (l'ajout d'une autre vue sur un ensemble de vues par exemple). Les applications natives permettent d'être au plus près de la plateforme cible (comportement spécifique pour chaque plateforme), et elles sollicitent les applications où les spécificités de design priment sur le partage de code.

Nous devons choisir dans cette même fenêtre la stratégie du partage de code. Un "projet partagé" est pour un projet où nous voulons faire du code spécifique pour chaque plateforme (dans un projet partagé) et où nous devons faire de la compilation conditionnelle pour chaque plateforme. Cette stratégie utilise un projet partagé et compliqué à mettre en oeuvre car les implémentations pour les 3 plateformes sont dans les mêmes fichiers et sont dupliquées.

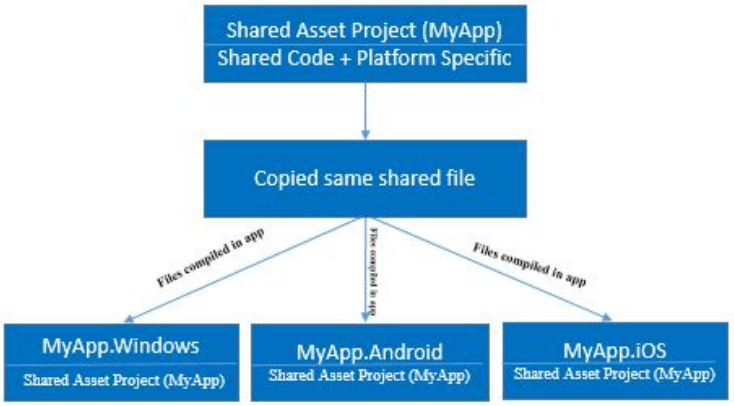


Figure 35 : Architecture d'un projet "Shared Project"

Nous avons également une stratégie plus intéressante pour le code à partager, le PCL (Portable Class Library) qui permet d'avoir un dossier commun et qui instaure les injections de dépendances lorsque nous devons accéder à des fonctionnalités spécifiques à chaque plateforme. Chaque plateforme a un dossier spécifique avec ses initialisations de composants et widgets, mais nous avons un dossier commun qui implémente lui-même dans les plateformes cibles. Une application en PCL est à préférer lorsque nous voulons faire une application plus complexe et que le code que nous intégrons est par conséquent cross-plateforme et donc ne nécessite pas de réécriture pour chaque OS.

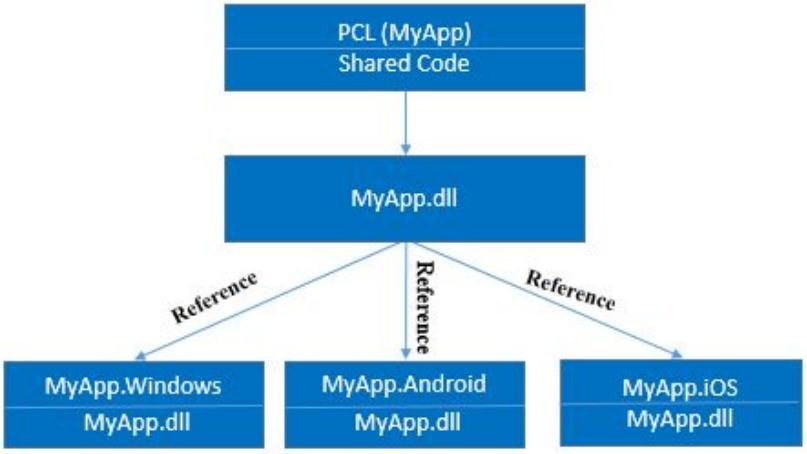


Figure 36 : Architecture d'un projet "PCL"

Selon la communauté Xamarin, les projets "shared projects" sont limités en terme de fonctionnalité et demandent souvent de la réécriture entre les plateformes, nous allons donc nous préoccuper des projets PCL en Xamarin.Forms pour FoodFactory.

Développement de l'application

Pour une adaptation des interfaces en fonction d'un contexte, d'un environnement ou d'un utilisateur, il est nécessaire de bien appréhender les patrons de conception et de programmation pour mettre en oeuvre une adaptation. Nous devons organiser un projet selon l'approche MVVM(Model/View/View-Model). Ce pattern permet de dissocier les objets métiers d'une vue et favorise la maintenabilité de l'application.

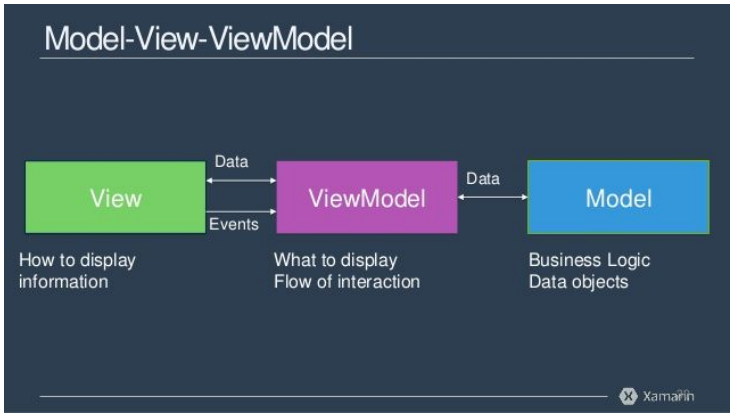
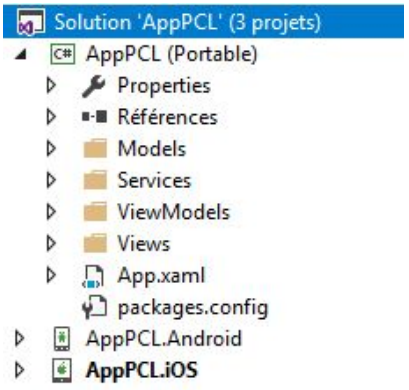


Figure 37 : pattern MVVM

Ce pattern est utile pour :

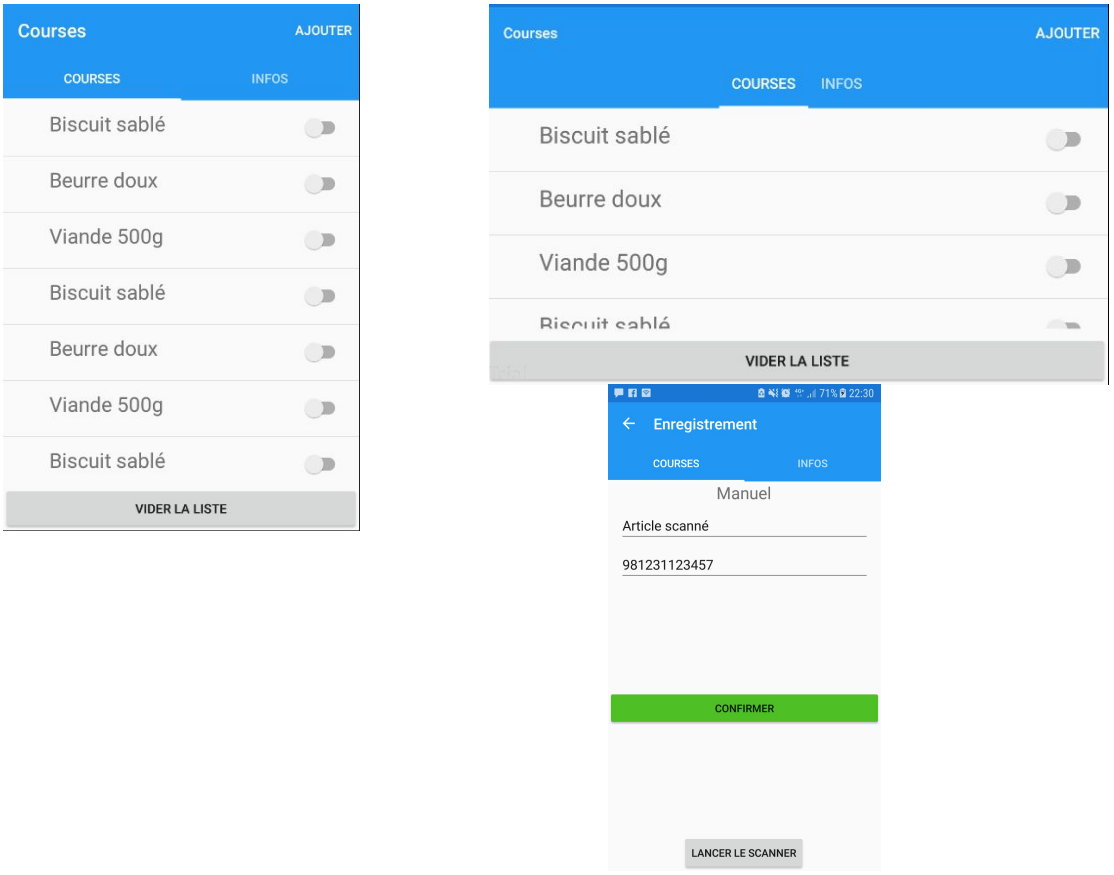
- Une maintenabilité de l'application,
- Pas besoin de toucher à la partie métier si nous devons refaire la partie UI,
- La partie Vue et Modèle sont distinctes ce qui permet de travailler en parallèle sans que l'une des deux parties impacte son code sur l'autre.



Une fois cette architecture mise en place, nous pouvons développer l'application.

Pour développer l'application selon des normes Android, iOS et Windows Phone, Xamarin gère la plateforme comme si elle était native ce qui nous facilite l'usage des ressources et des dépendances (comme sur Android studio pour Android et Xcode avec iOS). Nous pouvons ainsi nous focaliser sur l'adaptation des interfaces en fonction de l'environnement cible.

Android :



Figures 38 : captures d'écran de l'application sous Android

iOS :

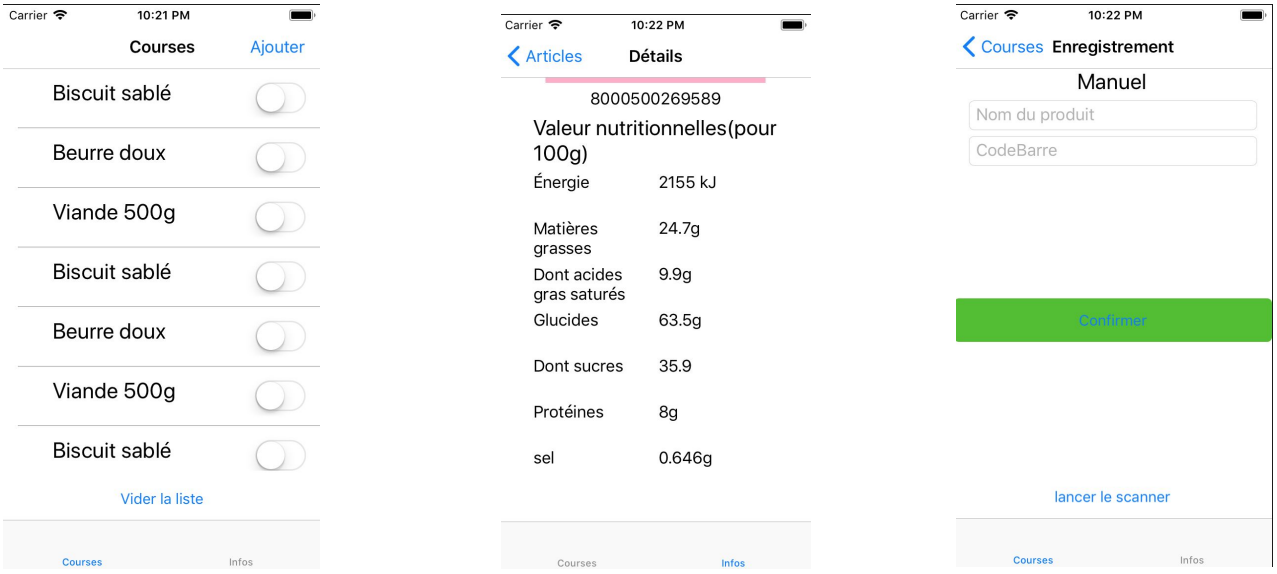


Figure 39 : Capture d'écran de l'application sous iOS

Nous avons adapté l'expérience utilisateur tout d'abord en fonction de l'orientation du smartphone et ainsi ce procédé nous garantit une adaptation pour les appareils de grand écran (pour les tablettes sous Android).

Figures 40 : capture d'écran de l'application de valeurs nutritionnelles

Les informations sont conservées et nous pouvons voir que qu'elles sont centrées en fonction de la largeur de l'écran. Nous constatons que d'après le rendu visuel et la documentation Xamarin, le projet crée bien des composants et widgets d'origine Android natif ce qui nous conforte dans l'idée que Xamarin utilise bien des composants propres aux OS et que sa puissance réside aussi dans le fait de téléverser un code commun en 3 OS distincts.

Cependant, du fait de proposer des implémentations communes, certaines fonctionnalités sont omises pour des raisons de compatibilité. Par exemple le composant "CheckBox" présent sur les OS n'est pas implémenté par Xamarin de base. Nous ne pouvons nous fier qu'au "Switch" qui permet de remplir la même fonction logique mais pas graphique.

Nous voulons également solliciter l'utilisation de la caméra ce qui suppose de connaître les permissions et les autorisations des OS cibles (voir partie "Accès aux capteurs").

Les services sous Xamarin

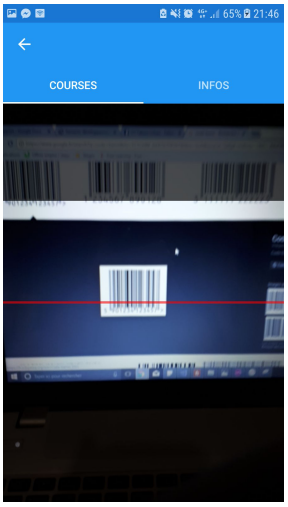
Les services que nous avons mis en place est l'appel à l'api Open Food Facts (https://fr.openfoodfacts.org/data) qui permet de récupérer les données de l'article ajouté par une saisie ou par scan code barre. Comme pour les 3 autres technologies précitées, nous faisons une requête GET pour récupérer un fichier JSON pour ensuite parser le fichier.

```
HttpClient client;
client = new HttpClient();
client.MaxResponseContentBufferSize = 256000;
var uri = new Uri("https://fr.openfoodfacts.org/api/v0/produit/"+barCode+".json");
var response = await client.GetAsync(uri);
if (response.IsSuccessStatusCode)
{
    var content = await response.Content.ReadAsStringAsync();
    //Items = JsonConvert.DeserializeObject<List<TodoItem>>(content);
}
```

Figure 41 : code pour l'appel à l'API Open Food Facts

Accès aux capteurs

L'adaptation de notre interface pour la consultation d'un article alimentaire réside également dans l'utilisation d'un périphérique propre à un appareil mobile : la caméra. Nous pouvons émettre une conjecture sur l'utilisation en fonction de l'environnement avec la caméra. En effet, si nous sommes sur un ordinateur il est plus difficile de solliciter une reconnaissance de code-barre avec une webcam (ordinateur) qu'avec une caméra intégrée au smartphone. Cela permet de nous rendre compte de l'adaptation en fonction de l'appareil car nous avons également l'utilisation alternative qui est la saisie d'un article avec un clavier.



Nous pouvons ainsi ajouter un article scanné dans la liste de courses grâce à l'action de la caméra qui détecte un code barre.

Cet appel n'a pu se faire qu'avec l'importation d'un plugin nommé Zxing.Net.Mobile

Figure 42 : plugins à installer pour l'utilisation de la caméra pour le scanner code barre

Nous devons faire une implémentation pour chaque plateforme :

iOS :

```
ZXing.Net.Mobile.Forms.iOS.Platform.Init();
```

Android :

```
    ZXing.Net.Mobile.Forms.Android.Platform.Init();
}
public override void OnRequestPermissionsResult(int requestCode, string[] permissions, Permission[] grantResults)
{
    global::ZXing.Net.Mobile.Forms.Android.PermissionsHandler.OnRequestPermissionsResult(requestCode, permissions, grantResults);
}
```

Et nous devons demander la permission de l'utilisation de la caméra pour l'usage de ce plugin.

```
<uses-permission android:name="android.permission.CAMERA" />
```

Une fois ceci fait, nous devons implémenter l'activation et l'utilisation de la caméra pour le scanner code barre :

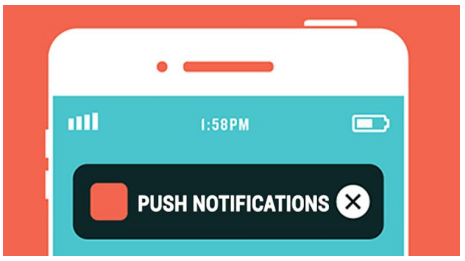
```
private async void Button_Clicked(object sender, EventArgs e)
{
    var scanPage = new ZXingScannerPage();
    ProductBarCode.Text = "";
    ProductName.Text = "";
    String resultat = "";
    scanPage.OnScanResult += (result) => {
        // Stop scanning
        scanPage.IsScanning = false;

        // Pop the page and show the result
        Device.BeginInvokeOnMainThread(() => {
            Navigation.PopAsync();
            resultat = ProductBarCode.Text = result.Text;
            DisplayAlert("Scanned Barcode", result.Text, "OK");
            ProductName.Text = MockData.SearchProductName(resultat);
            System.Diagnostics.Debug.WriteLine(resultat);
        });


    };
```

Figure 43 : code pour la gestion du scan

Ainsi nous créons une page de scan que nous intégrons au thread main.

Nous pouvons l'utiliser, une fois ceci fait le scanner code barre mis à la disposition par Redth (https://github.com/Redth/ZXing.Net.Mobile). Malheureusement, l'appel à l'API échoue.

Comment gérer l'adaptation aux dispositifs sous Xamarin ?



Comme pour Ionic nous pouvons intégrer plus ou moins facilement des fonctionnalités qui permettent de montrer une adaptation du dispositif. Par exemple, utiliser le système de notification (je vais au lit sans finir le taff) peut s'avérer utile pour la partie mobile. Nous pouvons en quelques lignes de code implémenter un système de notification de différentes façons avec différents plugins disponibles pour Xamarin.Forms (comme par exemple PushSharp). Nous devons implémenter dans du code spécifique (notamment pour la partie iOS où la restriction d'Apple nous contraint d'ajouter une couche supplémentaire).

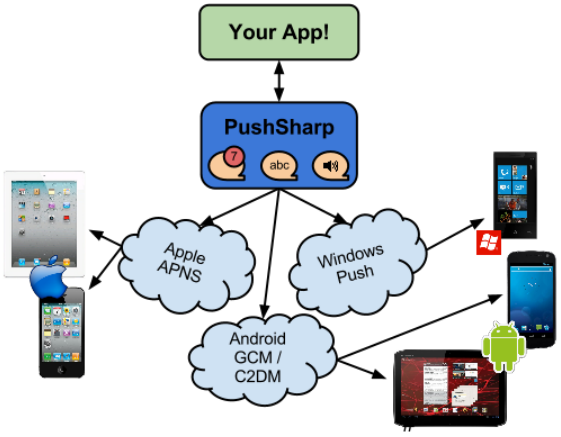


Figure 44 : architecture du fonctionnement du plugin PushSharp

Déploiement sur cibles réelles

Nous avons utilisé cette application mobile en debug et nous l'avons déployé par USB sur un Galaxy A5 (2017), un Galaxy J5 (2016) et sur une tablette galaxy tab 10.1 et un iPhone 7 plus et nous constatons les mêmes résultats :

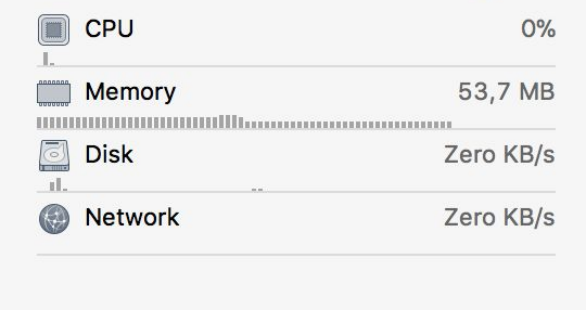


Figure 45 : performance de l'application sous iOS

Contre toute attente l'utilisation de la mémoire est due à l'utilisation des ressources en natif et de la caméra qui est dans cette situation plus énergivore que l'application Ionic. C'est une petite application, ce qui peut laisser penser que Xamarin utilise sa propre interprétation pour chaque plateforme et le téléverse selon des normes qui provoquent des défauts de performances. Nous constatons également que les performances sont nettement meilleures sur iOS que sur Android donc nous pouvons laisser penser que Xamarin adapte mieux le code sur iOS que sur Android en temps d'exécution et de performance.

Conclusion

En conclusion nous pouvons dire que l'utilisation d'un framework nuit gravement aux performances du site web. Le fait de travailler sans framework augmente effectivement les performances en permettant une liberté totale au niveau des optimisations. Cependant il faut une grande rigueur et organisation de développement pour le faire correctement et garder le code maintenable.

Nous pouvons constater entre les deux technologies cross-plateform que certaines fonctionnalités sont susceptibles de demander des efforts dans la communication entre l'application cliente et l'API. Les deux technologies se basent sur des frameworks qui ont trouvé une place dans le domaine du mobile dans l'histoire de l'informatique (.Net et Cordova/AngularJs nouvelle tendance du moment). Nous constatons que ces deux technologies permettent de développer des applications mobiles natives et sont équipées de plugins et dépendances qui permettent d'enrichir l'application. Nous voyons aussi que l'approche côté web (pour Ionic) est plus optimisée pour les appels à des API et en terme de temps de réponse, la différence de performance est nette. Xamarin, est plus lent, lourd et instable.

Cependant le développement cross-platform ne correspond pas obligatoirement à un code unique pour toutes les platforms, il peut arriver qu'un développement supplémentaire doive être réalisé en ajoutant des vérifications ou encore en utilisant des plugins différents afin de parvenir au même résultat.

Références

- http://vanilla-js.com/
- http://www.stefankrause.net/wp/
- https://openclassrooms.com/courses/developpez-vos-applications-web-avec-angularjs/installez-votre-environnement
- http://bradfrost.com/blog/post/this-is-the-web/
- http://www.supinfo.com/articles/single/509-xamarin-developpement-mobile-multiplateforme
- https://xamarin.com/
- https://3.bp.blogspot.com/-DDks3qQCIO8/V3UepJ01yJI/AAAAAAAACt4/HAb25Jk31cQ0eRt7NTNMT4AFdjJNFJb3wCLcB/s1600/SAP.PNG
- https://2.bp.blogspot.com/-L0FnL8N9yos/V3UeZbEwEeI/AAAAAAAACtw/QzUhSHfTkWw4nB43DlROnPb63z_qm56GwCKgB/s1600/PCL.PNG
- http://users.polytech.unice.fr/~bj406386/AI/
- https://www.webpagetest.org/
- https://developers.google.com/speed/pagespeed/insights/
- https://ionicframework.com/docs/
- https://www.disko.fr/reflexions/technique/apache-cordova-ou-comment-creer-des-applications-mobiles-avec-les-langages-du-web/
- http://flaven.fr/2016/04/ionicframework-angular-cordova-decouvrir-ionicframework-pour-creer-une-application-ios/
- https://developer.xamarin.com/guides/xamarin-forms/cloud-services/push-notifications/
- https://github.com/Redth/PushSharp
- https://raw.githubusercontent.com/Redth/PushSharp/master/Resources/PushSharp-Diagram.png