h e g

Haute école de gestion
Genève

Quels Frameworks et technologies pour le développement d'application mobile ?

Travail de Bachelor réalisé en vue de l'obtention du Bachelor HES

par :

Daniel AVELINO

Conseiller au travail de Bachelor :

Rolf HAURI, Maître d'enseignement HES

Genève, le 31 août 2023

Haute École de Gestion de Genève (HEG-GE)

Filière IG

Hes·so GENÈVE
Haute Ecole Spécialisée
de Suisse occidentale

Déclaration

Ce travail de Bachelor est réalisé dans le cadre de l'examen final de la Haute école de gestion de Genève, en vue de l'obtention du titre Bachelor of Sciences HES-SO en Informatique de gestion.

L'étudiant a envoyé ce document par email à l'adresse remise par son conseiller au travail de Bachelor pour analyse par le logiciel de détection de plagiat URKUND, selon la procédure détaillée à l'URL suivante : https://www.urkund.com .

L'étudiant accepte, le cas échéant, la clause de confidentialité. L'utilisation des conclusions et recommandations formulées dans le travail de Bachelor, sans préjuger de leur valeur, n'engage ni la responsabilité de l'auteur, ni celle du conseiller au travail de Bachelor, du juré et de la HEG.

« J'atteste avoir réalisé seul le présent travail, sans avoir utilisé des sources autres que celles citées dans la bibliographie. »

Fait à Genève le 31 août 2023

Daniel Avelino

Remerciements

Je souhaite remercier M. HAURI Rolf, qui m'a suggéré le thème de ce travail. Sa supervision, ses conseils et sa disponibilité ont grandement contribué à la réalisation de ce travail.

De plus, je souhaite aussi remercier mes proches pour leur soutien constant. Ils ont consacré du temps et des efforts pour lire et corriger mon travail.

Résumé

Ce travail de Bachelor vise à explorer le domaine du développement d'applications mobiles en se concentrant sur trois types d'applications : native, cross-plateforme et Progressive Web App (PWA).

La première partie du travail parlera de chaque catégorie d'applications mobiles. Chacune de ces catégories aura un chapitre dédié à son fonctionnement, ainsi que les avantages et inconvénients respectifs.

Le chapitre suivant présentera les environnements de développement pour chaque type d'application. Xcode pour les applications natives, React Native pour les applications cross-plateforme, et ReactJS pour les PWA.

Ensuite, les langages de programmation les plus couramment utilisés pour le développement mobile, à savoir JavaScript, Dart, Swift et Kotlin seront présentés. Une évaluation des avantages et des désavantages de chaque langage sera faite.

En outre, un chapitre dédié détaille le processus de développement de prototypes pour chaque type d'application, chacun offrant les mêmes fonctionnalités de base. Pour chaque prototype, le choix de l'environnement de développement, l'implémentation du prototype et une évaluation de la facilité de développement sont présentés. Chaque section se termine par une discussion sur les avantages et inconvénients de l'environnement de développement choisi.

Pour conclure, le travail partage des impressions personnelles sur les performances des différents prototypes avant de terminer avec une conclusion.

Table des matières

Liste des tableaux

Liste des figures

1. Introduction

Les smartphones et les tablettes ont révolutionné notre façon de vivre et de travailler. En effet, les applications mobiles sont devenues un élément indispensable de la vie quotidienne pour de nombreuses personnes. Nous utilisons ces appareils pour communiquer avec nos proches, naviguer sur Internet, effectuer des achats, et même pour travailler. Les entreprises semblent également avoir pris conscience de l'importance de développer des applications mobiles pour rester compétitives sur le marché.

Les applications mobiles offrent une meilleure expérience utilisateur, une portabilité accrue, et une accessibilité à tout moment et partout. Les développeurs doivent donc être en mesure de concevoir des applications performantes, conviviales et fonctionnelles pour les plateformes mobiles.

Le développement d'applications mobiles est un domaine en constante évolution qui nécessite une connaissance approfondie des différents environnements de développement et des technologies associées. Les développeurs doivent être en mesure de choisir les environnements de développement les plus appropriés pour leur projet, afin de garantir un développement efficace et une expérience utilisateur optimale.

La création d'applications mobiles a traditionnellement été réalisée en utilisant la méthode native, c'est-à-dire en développant deux versions distinctes pour chaque système d'exploitation qui sont Android et iOS. Cependant, cette méthode peut être fastidieuse et coûteuse pour les développeurs. Heureusement, les technologies cross-plateformes ont émergé pour répondre à cette problématique. En outre, les Progressive Web App (PWA) sont une autre alternative aux applications mobiles traditionnelles. En effet, ces dernières offrent la possibilité de développer des applications web qui ressemblent à des applications mobiles et qui peuvent être installées sur un smartphone.

Finalement, l'objectif de ce travail de Bachelor est de fournir une analyse des différents environnements de développement mobile et des types d'applications mobiles, pour aider les développeurs à prendre des décisions éclairées lorsqu'ils choisissent un framework et une technologie pour leur projet.

2. Types d'application mobile

Les applications mobiles peuvent être classées en trois catégories principales : les applications natives, les applications cross-plateforme et les PWA.

2.1 Application native

Les applications natives sont développées spécifiquement pour une plateforme mobile, telles qu'iOS ou Android. Apple fournit en conséquence aux développeurs un environnement de développement nommé Xcode (iOS). Google fournit comme Apple son environnement de développement nommé Android Studio (Android). Les applications sont écrites dans un langage natif, Objective-C ou Swift pour iOS, Java ou Kotlin pour Android.

2.1.1 Fonctionnement

Le fonctionnement des applications natives est relativement simple. Le code est développé pour une plateforme spécifique et il faut utiliser les environnements de développement mis à disposition par Apple et Android pour créer l'application. L'application est ensuite compilée pour fonctionner sur cette plateforme spécifique et est disponible pour les utilisateurs à travers des canaux de distribution tels que Google Play ou l'App Store.

2.1.2 Avantages

Du fait que ce type d'application est écrite dans un langage natif, cela implique que l'application sera spécifiquement conçue pour fonctionner sur un système d'exploitation mobile particulier. Par conséquent, cela lui confère une interaction directe avec le système d'exploitation et donc des performances élevées[1].

Les applications natives bénéficient de toutes les fonctionnalités du système d'exploitation. De plus, elles peuvent également accéder aux API spécifiques à la plateforme pour une meilleure intégration.

2.1.3 Inconvénients

En revanche, développer des applications natives peut se révéler coûteux. Cela s'explique par le fait que si l'on souhaite que l'application soit compatible pour iOS et Android, il est nécessaire de développer deux applications qui auront un code source différent qui correspondra aux deux types de plateformes mobiles. Par ce fait, cela implique aussi un temps de développement plus long.

[1] https://medium.com/neoxia/pwa-vs-flutter-vs-react-native-vs-native-dc06a17ebf1a

Ensuite, en ce qui concerne les mises à jour, il faudra donc les développer séparément pour chaque plateforme. En outre, ce type d'application nécessite des mises à jour régulières pour maintenir leur compatibilité avec les dernières versions des systèmes d'exploitation mobiles.

2.2 Application cross-plateforme

Une application mobile cross-plateforme est une application qui fonctionne sur plusieurs plateformes mobiles, telles qu'iOS et Android, en utilisant un seul code source.

Les frameworks les plus populaires pour le développement d'applications cross-plateforme sont, par exemple, React Native, Flutter et Xamarin.

2.2.1 Applications cross-plateforme - Première Génération

Ce type d'application encapsule une vue Web dans une application native. Elles sont appelées « application WebView ». En d'autres termes, cela signifie qu'elles utilisent le moteur de rendu Web du système d'exploitation pour afficher du contenu HTML, CSS et JavaScript. Cela ressemble à une page web dans une application native. Les applications sont écrites en JavaScript et utilisent des frameworks, tels que Cordova ou PhoneGap.

L'un des avantages des applications cross-plateforme de première génération est la facilité de développement. Étant donné que du HTML, CSS et du JavaScript est utilisé, la majorité des développeurs connaissent déjà ces langages. De ce fait, il n'est pas nécessaire d'apprendre un nouveau langage, par exemple. De plus, il n'est pas nécessaire d'avoir un code source pour chaque plateforme.

L'inconvénient majeur est que les applications sont souvent plus lentes et moins performantes que les applications natives du fait qu'elles utilisent une WebView. De plus, le design n'est pas similaire à des applications natives, ce qui peut être déroutant pour les utilisateurs.

Cette première génération d'applications cross-plateforme a permis d'ouvrir la voie à React Native, Flutter et Xamarin qui ont repris l'aspect de la réutilisation du code et apporté les performances manquantes.

2.2.2 Applications cross-plateforme - Deuxième Génération

La deuxième génération d'applications cross-plateforme a eu pour but d'améliorer les performances et l'accès aux fonctionnalités natives des plateformes. React Native, Xamarin et Flutter font partie de cette catégorie.

2.2.2.1 Fonctionnement

Pour qu'une application cross-plateforme puisse fonctionner, il y a plusieurs éléments clé qui la composent : moteur de rendu, bibliothèques et frameworks, couche d'abstraction et le code source commun. De plus, pour que le code source puisse être interprété par chacune des différentes plateformes, il est nécessaire d'avoir un pont. Cette partie sera expliquée plus en détail au chapitre « React Native ».

2.2.2.2 Moteur de rendu

Il permet d'afficher les éléments graphiques et la gestion des interactions utilisateur. Chaque framework utilise un moteur de rendu qui lui est spécifique. Il permet d'optimiser la performance et l'apparence de la plateforme.

Par exemple, React Native utilise un moteur de rendu natif pour chaque plateforme (iOS et Android), c'est pour cela que les applications développées avec ce framework ont l'apparence d'applications natives. Flutter lui utilise Skia comme moteur de rendu.

2.2.2.3 Bibliothèques et frameworks

Chaque bibliothèque ou framework fournit aux développeurs des fonctionnalités et des composants pour développer une application. React Native par exemple fournit un composant nommé « View ». De son côté Flutter fournit aussi le même type de composant, mais sous le nom de « Container ».

2.2.2.4 Couche d'abstraction

La couche d'abstraction facilite l'intégration des fonctionnalités natives tout en n'ayant besoin que d'un code unique pour toutes les plateformes. Cette couche dans le cas de React Native est représentée par des modules et composants natifs.

2.2.2.5 Code source commun

Le code source est par la suite interprété, afin de pouvoir fonctionner sur chaque plateforme. React Native utilise JavaScript, quant à Flutter, il utilise Dart, et Xamarin utilise C#.

2.2.3 Avantages

Étant donné que les applications cross-plateforme fonctionnent sur iOS et Android, cela permet de réduire les coûts et les délais de développement.

Une application cross-plateforme peut permettre de toucher un plus grand public en offrant l'application sur plusieurs systèmes d'exploitation mobiles.

Ensuite, comme il n'y a besoin que d'un code source, les mises à jour et les maintenances sont plus faciles et moins coûteuses.

2.2.4 Inconvénients

En revanche, comme les applications cross-plateforme peuvent être utilisées sur iOS et Android, il est possible d'avoir des problèmes de compatibilité avec certaines fonctionnalités spécifiques à chaque plateforme mobile.

De plus, comme expliqué dans la partie 2.2.2.1, les applications cross-plateforme utilisent des bridges, par exemple, pour convertir leur code en code natif. De ce fait, elles peuvent être moins performantes[2] que les applications natives.

Ensuite, comme il faut utiliser des frameworks spécifiques, il se peut que certaines fonctionnalités existantes sur des applications natives ne soient pas disponibles.

2.3 Application Progressive Web App (PWA)

Les Progressive Web App (PWA) sont des applications web. Elles sont disponibles via un navigateur web ou bien elles peuvent directement être installées sur notre téléphone. Elles offrent une expérience utilisateur proche de celle d'une application native. Les PWA sont développées grâce à du HTML, CSS et du JavaScript.

2.3.1 Fonctionnement

Pour mettre en place une PWA, il a deux éléments principaux à retenir : Service worker et manifest.

Un manifest est un fichier JSON de configuration qui est utilisé par le navigateur web. Il fournit des informations, telles que le nom de l'application, l'icône, l'URL de départ, la couleur du thème et d'autres métadonnées.

[2] https://medium.com/neoxia/pwa-vs-flutter-vs-react-native-vs-native-dc06a17ebf1a

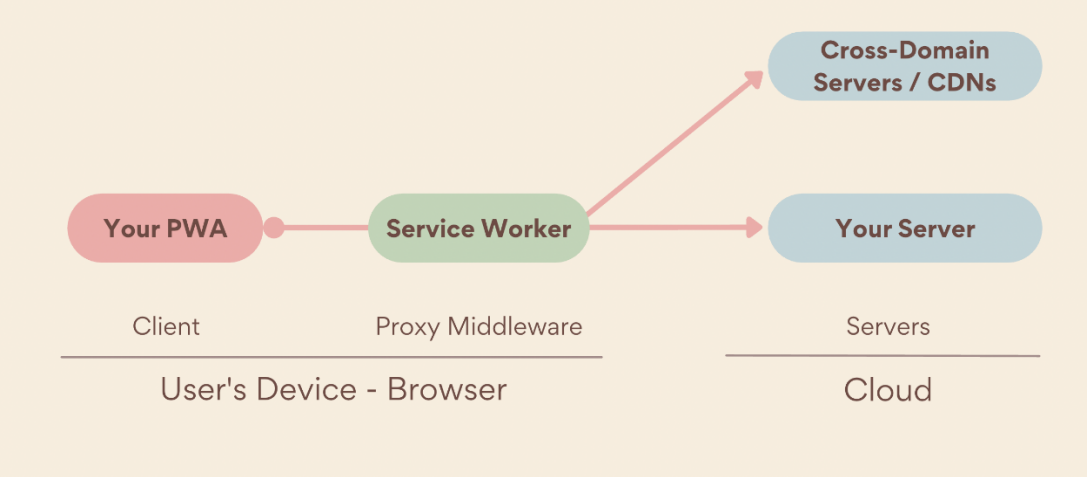
Figure 1 : Exemple fichier manifest.json

```
{
  "short_name": "React App",
  "name": "Create React App Sample",
  "icons": [
    {
      "src": "favicon.ico",
      "sizes": "64x64 32x32 24x24 16x16",
      "type": "image/x-icon"
    },
    {
      "src": "logo192.png",
      "type": "image/png",
      "sizes": "192x192"
    },
    {
      "src": "logo512.png",
      "type": "image/png",
      "sizes": "512x512"
    }
  ],
  "start_url": ".",
  "display": "standalone",
  "theme_color": "#000000",
  "background_color": "#ffffff"
}
```

(Capture écran personnelle)

Le Service Worker est un proxy virtuel qui se place entre la PWA et le serveur utilisé par la PWA. Il permet la mise en cache de certaines données et par la suite de les utiliser pour les afficher à l'utilisateur lorsqu'il est hors ligne ou bien pour charger la PWA plus rapidement. Le Service Worker permet également de faire de la synchronisation en arrière-plan. Prenons le cas de Twitter, un utilisateur est en train d'écrire son tweet et d'un coup, il perd sa connexion Internet, cela ne pose pas de problème aux PWA, car grâce à la synchronisation en arrière-plan, lorsqu'il se connectera à nouveau à Internet, son tweet sera automatiquement posté.

Figure 2 : Service Worker



(https://web.dev/learn/pwa/service-workers/)

2.3.2 Avantages

Les PWA peuvent être exécutées sur plusieurs types d'appareils. De plus, elles peuvent même être installées directement sur son téléphone sans passer par le store officiel du téléphone. Cela veut dire que l'application n'a pas besoin de recevoir l'approbation d'un quelconque store pour être installée. Ensuite, comme elle n'est pas publiée sur un store officiel, il n'y a pas non plus besoin d'attendre un certain temps pour qu'elle soit accessible au grand public.

Ce type d'application permet aussi le fonctionnement en mode hors ligne et aussi des performances améliorées grâce au Service Worker comme vu au point 2.3.1.

Contrairement aux autres types d'application mobile, pour la mettre à jour, il faut que l'utilisateur le fasse manuellement. Comme les PWA sont des applications web, il suffit de modifier le code source et la mise à jour se fait automatiquement pour tous les utilisateurs. Cela garantit que les utilisateurs utilisent toujours la version la plus récente de l'application.

2.3.3 Inconvénients

Certaines fonctionnalités sont limitées[3] sur iOS comme la synchronisation en arrière-plan et les notifications.

[3] https://firt.dev/notes/pwa-ios/

2.3.4 Résumé des avantages et inconvénients

Tableau 1 : Avantages et inconvénients de chaque type d'applications mobiles

Type d'applications	Avantages	Désavantages
Native	Performances élevées	Coûts de développement plus élevés
	Accès à toutes les fonctionnalités du système d'exploitation	Nécessité de développer séparément pour iOS et Android
	Accès aux API spécifiques à la plateforme	Temps de développement plus long
		Mises à jour régulières nécessaires pour maintenir la compatibilité
Cross-plateforme	Coûts et délais de développement réduits	Problèmes de compatibilités possibles avec certaines fonctionnalités spécifiques à chaque plateforme
	Touche un plus grand public	Moins performantes que les applications natives
	Mises à jour et maintenances plus faciles et moins coûteuses	Certaines fonctionnalités natives peuvent ne pas être disponibles
PWA	Fonctionne sur plusieurs types d'appareils	Certaines fonctionnalités non disponibles sur iOS
	Installation sans passer par un store officiel	
	Fonctionnement en mode hors-ligne et performances améliorées	
	Mise à jour automatique pour tous les utilisateurs	

3. Environnements de développement pour chaque type d'application

Étant donné que chaque type d'application mobile comprend plusieurs environnements de développement, je me suis focalisé uniquement sur un environnement par type d'application.

3.1 Native

3.1.1 Xcode

3.1.1.1 Qu'est-ce que Xcode

Xcode est un environnement de développement intégré (IDE) créé par Apple pour faciliter le développement d'applications pour iOS, macOS, watchOS et tvOS. Il est conçu spécifiquement pour les développeurs qui utilisent les langages de programmation Swift et Objective-C. Xcode offre de nombreuses fonctionnalités pour faciliter le développement, le débogage et le déploiement d'applications.

Xcode demeure l'environnement de développement officiel d'Apple pour le développement d'applications d'appareils Apple.

3.1.1.2 SwiftUI

3.1.1.2.1 Approche déclarative

Contrairement à UIKit qui est aussi un framework qui permet de développer des applications natives comme SwifUI. Les deux frameworks ont choisi des stratégies différentes. UIKit utilise une approche impérative, alors que SwiftUI utilise une approche déclarative.

La différence réside dans le fait que l'approche déclarative, utilisée par SwiftUI, permet de décrire le résultat attendu sans spécifier explicitement toutes les étapes nécessaires pour y parvenir. À l'inverse, UIKit, en adoptant une approche impérative, requiert une description détaillée de chaque étape pour arriver à l'état final. L'approche se concentre sur le "comment" plutôt que le résultat souhaité "quoi".

Pour illustrer la différence entre ces deux approches, prenons un exemple en JavaScript. Imaginons que nous souhaitons créer une liste de nombres et doubler chaque élément.

Tableau 2 : Code expliquant la différence entre approche itérative et déclarative

Approche itérative	Approche déclarative

`const nombres = [1, 2, 3, 4];`
`let doubles = [];`
`for (let i = 0; i < nombres.length; i++) {`
`··doubles[i] = nombres[i] * 2;`
`}`
	`const nombres = [1, 2, 3, 4];`
	`let doubles = nombres.map(`
	`··(nombre) => nombre * 2);`

Pour l'approche itérative, chaque étape est détaillée, offrant un contrôle total sur le code et permettant une observation minutieuse de ses actions. En revanche, dans l'approche déclarative, les fonctionnalités fournies par JavaScript, par exemple "map" qui génère un nouveau tableau, sont utilisées. Cela diffère de l'approche itérative où il est nécessaire de déclarer explicitement un tableau, puis d'y insérer les éléments.

3.1.1.2.2 Mise à jour de l'état

Pour que SwiftUI puisse savoir quand mettre à jour l'interface utilisateur, il faut utiliser la propriété Oberservable par exemple sur une classe ou une variable. En quelques mots, cela permet de dire à SwiftUI d'observer les modifications et lorsqu'une ou plusieurs modifications se produisent, SwiftUI pourra mettre à jour l'interface en question.

De plus, SwiftUI utilise un procédé assez similaire à ReactJS. Le diffing permet de faire la comparaison de la nouvelle version de l'interface avec l'ancien. Lorsqu'une différence est détectée, seules les parties concernées seront mises à jour.

Pour illustrer ceci, prenons l'exemple d'une liste d'éléments dans laquelle une modification sera apportée à un élément. Grâce à la propriété Observable, SwiftUI surveille les changements. Lorsqu'une modification sera détectée, la liste affichée à l'utilisateur sera mise à jour. Cela permet au développeur de se concentrer sur la logique métier, tandis que SwiftUI gère la synchronisation de l'état avec l'interface utilisateur[4].

[4] https://medium.com/just-tech-it-now/comprendre-le-rendu-dans-swiftui-8353c3a04624

3.2 Cross-plateforme

3.2.1 React Native

React Native est un framework open source développée par Facebook. Sa première version a été publiée le 26 mars 2015[5]. Ce framework est basé sur la bibliothèque ReactJS et utilise le langage JavaScript.

3.2.1.1 Javascript VM

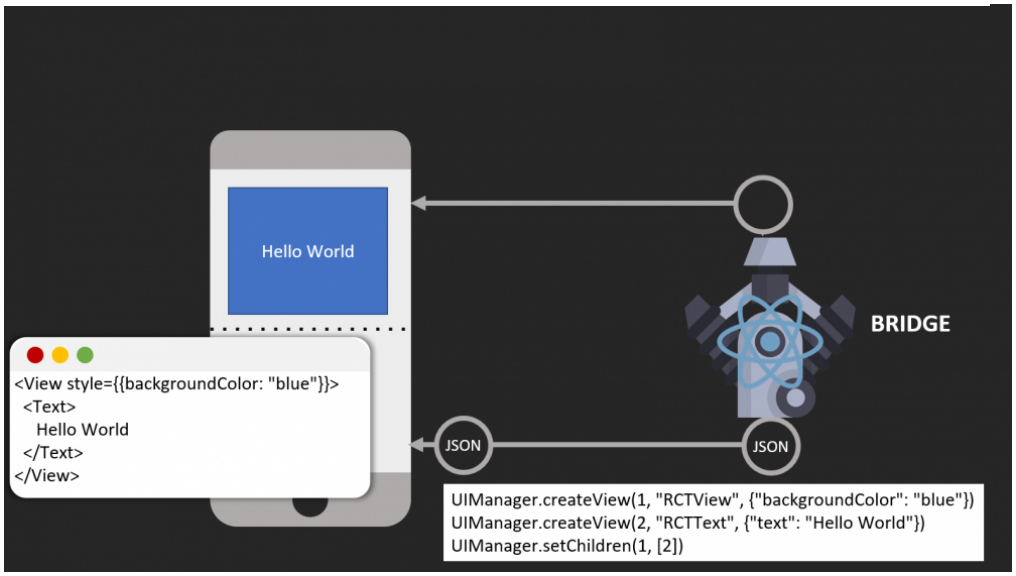
React Native utilise la machine virtuelle JavaScript qui exécute le code de l'application. Le nom de la machine virtuelle utilisé est JavaScriptCore, c'est le moteur JavaScript de Safari. JavaScriptCore est un framework OpenSource qui est un WebKit.

3.2.1.2 Bridge

Il assure la communication entre la machine virtuelle et le code natif de l'appareil. En d'autres termes, le bridge permet que le code JavaScript puisse être traduit en code natif. Le bridge est développé grâce aux langages Java et C++.

Ce bridge est déclenché lors de l'utilisation de l'une de ces deux commandes. Pour Android, il faudra utiliser cette commande : `react-native run-android.` En ce qui concerne iOS, il faudra à la place utiliser cette commande : `react-native run-ios`

Figure 3 : Fonctionement du bridge



(https://thecodingmachine.com/developpement-mobile-les-applis-react-native/)

[5] https://fr.wikipedia.org/wiki/React_Native

3.2.1.3 Native Modules

Il est possible que les composants prédéfinis par React Native ne conviennent pas aux besoins spécifiques de l'application en cours de développement. A ce moment, les modules natifs[6] font leur apparition. Cela permet aux développeurs de mettre à disposition ou d'exposer certaines fonctionnalités spécifiques des systèmes d'exploitation iOS ou Android directement dans une application créée avec React Native.

En revanche, pour développer des modules natifs, il faut obligatoirement être familier avec Swift ou Objectif-C pour IOS, et Java ou Kotlin pour Android.

3.3 PWA

3.3.1 ReactJS

ReactJS est une bibliothèque open source développée par Facebook. Sa première version a été publiée le 29 mai 2013. Cette librairie utilise le langage JavaScript et son but est de faciliter le développement d'application single page et de créer des interfaces utilisateurs[7].

3.3.1.1 DOM virtuel

3.3.1.1.1 Explication

Le DOM virtuel (VDOM) est un concept de programmation dans lequel une représentation idéale, ou « virtuelle », d'une interface utilisateur (UI) est conservée en mémoire et synchronisée avec le DOM « réel » par une bibliothèque telle que ReactDOM[8].

3.3.1.1.2 Fonctionnement

React utilise le concept de state et de props dans les composants. Chaque state possède un « setter ». Il va permettre de mettre à jour le state en fonction des cliques qu'effectue un utilisateur[9]. React va comprendre que le state a été mis à jour et que l'affichage doit être mis à jour. De ce fait, un nouveau DOM virtuel va être créé. Suite à ça, React va comparer le DOM virtuel avec le DOM réel et si besoin le mettre à jour. Ce

[6] https://reactnative.dev/docs/native-modules-intro#:~:text=React%20Native%20does%20not%20expose,in%20your%20React%20Native%20application

[7] https://react.dev/

[8] https://fr.legacy.reactjs.org/docs/faq-internals.html#:~:text=Le%20DOM%20virtuel%20(VDOM)%20est,une%20biblioth%C3%A8que%20telle%20que%20ReactDOM.

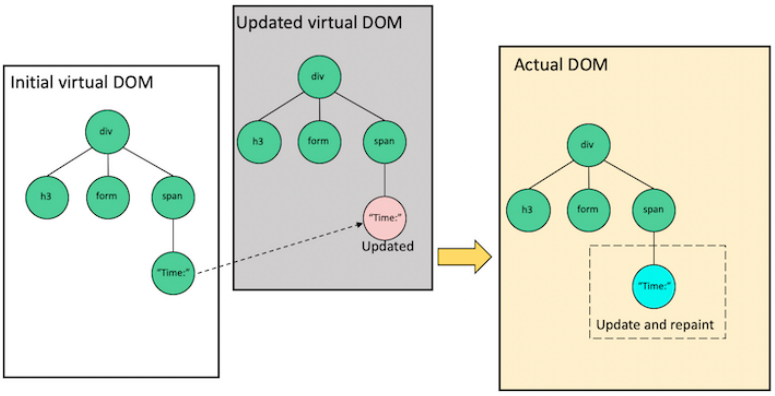
[9] https://payalpaul2436.medium.com/10-main-core-concept-you-need-to-know-about-react-303e986e1763#:~:text=React%20allows%20us%20to%20pass,%2Dto%2Dchild%20components).

processus s'appelle la « réconciliation ». Il va être abordé plus en profondeur dans le chapitre suivant.

3.3.1.1.3 L'algorithme de réconciliation

L'algorithme de réconciliation permet de déterminer comment mettre à jour le DOM réel de la manière la plus optimale. En d'autres mots, il permet de comparer le DOM virtuel précédent au nouveau DOM virtuel. Cela s'appelle le « diffing ». Si des mises à jour sont détectées, le DOM réel sera mis à jour, mais uniquement aux endroits où les différences ont été constatées.[10]

Figure 4 : Fonctionnement du DOM virtuel



(https://blog.logrocket.com/virtual-dom-react/)

Il est important de savoir que pour détecter des changements dans une liste, React a besoin d'avoir une « key » pour chacun des éléments. Cela permet d'éviter d'avoir à mettre la liste entière à jour dans le DOM réel. Grâce à la « key », uniquement l'élément de la liste concerné par le changement sera mis à jour dans le DOM réel[11].

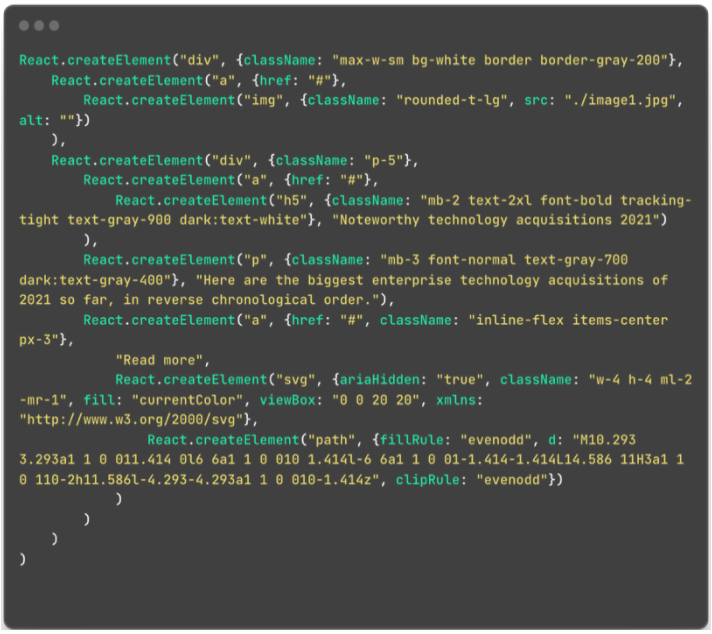
[10, 8] https://blog.logrocket.com/virtual-dom-react/

3.3.2 JSX

JSX est une extension *syntaxique pour JavaScript qui vous permet d'écrire des balises de type HTML à l'intérieur d'un fichier JavaScript.*[12]

Voici un code n'utilisant pas du JSX :

Figure 5 : Code n'utilisant pas de JSX



```
React.createElement("div", {className: "max-w-sm bg-white border border-gray-200"},
    React.createElement("a", {href: "#"},
        React.createElement("img", {className: "rounded-t-lg", src: "./image1.jpg",
alt: ""})
    ),
    React.createElement("div", {className: "p-5"},
        React.createElement("a", {href: "#"},
            React.createElement("h5", {className: "mb-2 text-2xl font-bold tracking-
tight text-gray-900 dark:text-white"}, "Noteworthy technology acquisitions 2021")
        ),
        React.createElement("p", {className: "mb-3 font-normal text-gray-700
dark:text-gray-400"}, "Here are the biggest enterprise technology acquisitions of
2021 so far, in reverse chronological order."),
        React.createElement("a", {href: "#", className: "inline-flex items-center
px-3"},
            "Read more",
            React.createElement("svg", {ariaHidden: "true", className: "w-4 h-4 ml-2
-mr-1", fill: "currentColor", viewBox: "0 0 20 20", xmlns:
"http://www.w3.org/2000/svg"},
                React.createElement("path", {fillRule: "evenodd", d: "M10.293
3.293a1 1 0 011.414 0l6 6a1 1 0 010 1.414l-6 6a1 1 0 01-1.414-1.414L14.586 11H3a1 1
0 110-2h11.586l-4.293-4.293a1 1 0 010-1.414z", clipRule: "evenodd"})
            )
        )
    )
)
```

(Capture écran personnelle)

[12] https://fr.legacy.reactjs.org/docs/introducing-jsx.html

Le JSX facilite la compréhension et la maintenance grâce à sa syntaxe simplifiée. Il évite la nécessité d'écrire constamment "React.createElement". À la place, des balises HTML peuvent être utilisées.

Figure 6 : Code utilisant du JSX

```
<div class="max-w-sm bg-white border border-gray-200">
  <a href="#">
    <img class="rounded-t-lg" src="./image1.jpg" alt="" />
  </a>
  <div class="p-5">
    <a href="#">
      <h5 class="mb-2 text-2xl font-bold tracking-tight text-gray-900 dark:text-
white">
        Noteworthy technology acquisitions 2021
      </h5>
    </a>
    <p class="mb-3 font-normal text-gray-700 dark:text-gray-400">
      Here are the biggest enterprise technology acquisitions of 2021 so far, in
      reverse chronological order.
    </p>
    <a
      href="#"
      class="inline-flex items-center px-3"
    >
      Read more
      <svg
        aria-hidden="true"
        class="w-4 h-4 ml-2 -mr-1"
        fill="currentColor"
        viewBox="0 0 20 20"
        xmlns="http://www.w3.org/2000/svg"
      >
        <path
          fill-rule="evenodd"
          d="M10.293 3.293a1 1 0 011.414 0l6 6a1 1 0 010 1.414l-6 6a1 1 0 01-1.414-
1.414L14.586 11H3a1 1 0 110-2h11.586l-4.293-4.293a1 1 0 010-1.414z"
          clip-rule="evenodd"
        ></path>
      </svg>
    </a>
  </div>
</div>;
```

(Capture écran personnelle)

Il est important de noter que lors de l'utilisation du JSX et de l'exécution de la commande "npm run dev", cela déclenche une chaîne d'outils de développement comprenant notamment Babel. Babel est un transpileur qui convertit le JSX en JavaScript. De plus, le JSX permet d'intégrer du JavaScript directement dans le code grâce à cette notation « {} », rendant ainsi possible la création d'interfaces utilisateurs dynamiques. C'est grâce à cette intégration que ce qui est affiché à l'utilisateur peut changer en fonction des interactions ou des données reçues.

Figure 7 : Utilisation de JavaScript avec du JSX

```
const name = "John Doe";

const nameElement = (
  <div>
    <h1>Hello {name}</h1>
  </div>
)
```

(Capture écran personnelle)

4. Langages de programmation pour le développement mobile

4.1 JavaScript

JavaScript est un langage de programmation populaire pour le développement web et mobile. Il est utilisé pour créer des applications qui fonctionnent sur plusieurs plates-formes, notamment iOS et Android, en utilisant des frameworks tels que React Native et Ionic.

4.1.1 Avantages et désavantages

Tableau 3 : Résumé avantages et désavantages JavaScript

Avantages	Désavantages
Très populaire et largement utilisé que ce soit dans le développement web et mobile	Performances inférieures par rapport aux langages natifs comme Swift et Kotlin[13]
Permets le développement d'application cross-plateforme	L'absence de typage statique peut conduire à des erreurs à l'exécution
Grande communauté de développeurs pour le support et l'apprentissage	

4.2 Dart

Dart est un langage de programmation open source développé par Google. Il est utilisé pour créer des applications mobiles pour Android et iOS. Dart est utilisé avec le framework Flutter qui permet de développer des applications cross-plateformes.

4.2.1 Avantages et désavantages

Tableau 4 : Résumé avantages et désavantages Dart

Avantages	Désavantages
Permets le développement d'applications mobiles performantes avec Flutter	Performances légèrement inférieures par rapport aux langages natifs comme Swift et Kotlin[14]
Fort soutien de Google	La courbe d'apprentissage peut être plus raide pour les débutants.
Grande communauté de développeurs pour le support et l'apprentissage	

[13] https://medium.com/swlh/flutter-vs-native-vs-react-native-examining-performance-31338f081980

[14] https://medium.com/swlh/flutter-vs-native-vs-react-native-examining-performance-31338f081980

4.3 Swift

Swift est un langage de programmation développé par Apple pour créer des applications mobiles pour iOS et macOS. Il a été introduit en 2014 pour remplacer Objective-C. Swift est devenu un choix populaire pour le développement mobile iOS, dépassant Objective-C[15].

4.3.1 Avantages et désavantages

Tableau 5 : Résumé avantages et désavantages Swift

Avantages	Désavantages
Langage natif pour le développement d'applications iOS et macOS, offrant de meilleures performances	Limité à l'écosystème Apple
Soutien massif de la part d'Apple et de la communauté de développeurs	Documentation complète, mais mauvaise structuration. Difficulté à trouver l'information cherchée

4.4 Kotlin

Kotlin est un langage de programmation open source développé par JetBrains. Il est utilisé pour créer des applications mobiles pour Android. Kotlin est devenu un choix populaire pour le développement mobile en raison de sa simplicité comparé à Java. De plus, Kotlin apporte une meilleure lisibilité du code que Java.

4.4.1 Avantages et désavantages

Tableau 6 : Résumé avantages et désavantages Kotlin

Avantages	Désavantages
Fortement soutenu par Google pour le développement Android	Limité à l'écosystème Android
Offre une compatibilité totale avec Java et permets de convertir[16] le code Java en Kotlin	Courbe d'apprentissage qui peut être lente suivant l'expérience du développeur
Code plus lisible[17]	
Permet d'éviter les erreurs de type null value[18]	

[15] https://pypl.github.io/PYPL.html
[16] https://androidtutos.com/convertir-un-code-java-en-kotlin-sous-android-studio/
[17] https://alignminds.com/advantages-kotlin-over-java/
[18] https://alignminds.com/advantages-kotlin-over-java/

5. Développement du prototype

Étant donné que je dois concevoir trois prototypes, un pour chacun des environnements de développement, chacune des applications implémentera uniquement les fonctionnalités que j'estime les plus importantes pour un prototype de site d'e-commerce.

5.1 Fonctionnalités de l'application

5.1.1 Afficher une page avec des articles

Lorsque l'utilisateur ouvre l'application, une page avec des produits s'affiche. Il pourra consulter les différents produits. Les différents produits seront récupérés grâce à une requête à une API REST[19].

5.1.2 Panier

Un onglet pour consulter son panier sera mis en place. Il permettra de voir les différents articles qui s'y trouvent. L'utilisateur pourra supprimer les produits qu'il ne désire pas garder. Il pourra aussi mettre à jour la quantité d'un produit. Ensuite, le prix total du panier sera affiché.

5.1.3 Filtrer les produits par nom

Une barre de recherche permettra à l'utilisateur de filtrer les produits par leur nom. Cela lui permettra d'afficher uniquement le produit voulu sans devoir le chercher dans le catalogue entier.

[19] https://fakestoreapi.com/

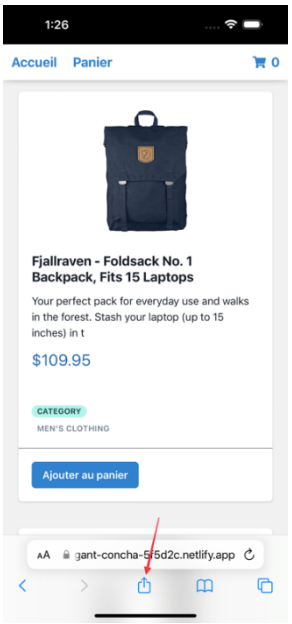
5.2 Prototype PWA

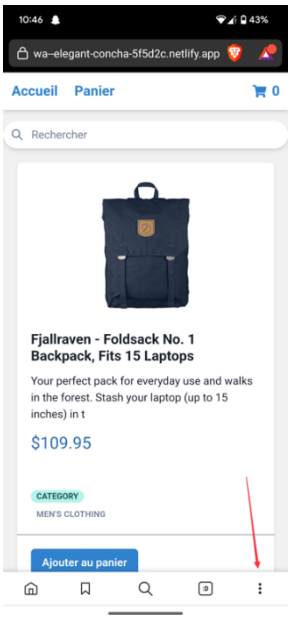
5.2.1 Choix de l'environnement de développement

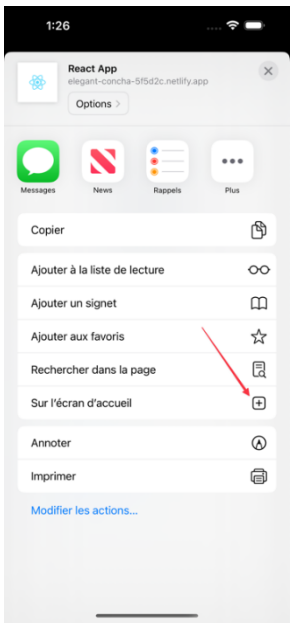
J'ai décidé de développer le prototype de la PWA grâce à l'environnement de développement ReactJS.

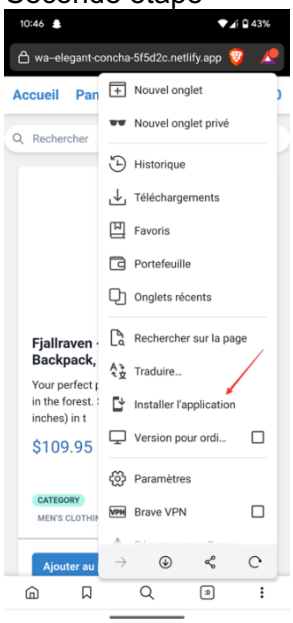
5.2.2 Installer la PWA

Tableau 7 : Mode d'emploi pour installer la PWA

<table>
<tr><th>iOS</th><th>Android</th></tr>
<tr><td>Première étape
 

(Capture écran personnelle)</td><td>Première étape
 

(Capture écran personnelle)</td></tr>
<tr><td>Seconde étape
 

(Capture écran personnelle)</td><td>Seconde étape
 

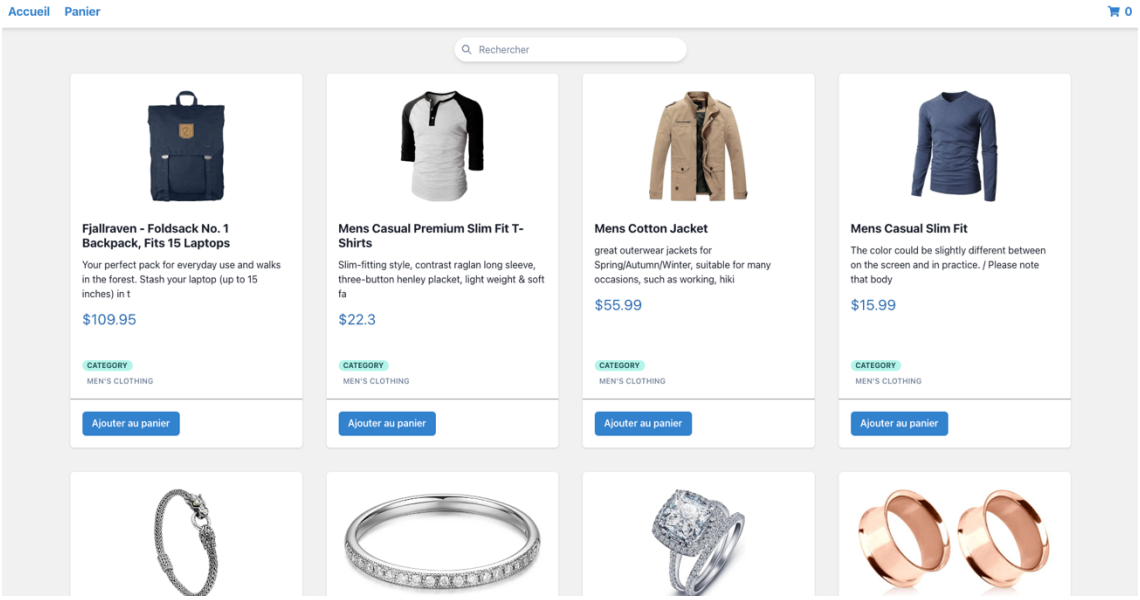
(Capture écran personnelle)</td></tr>
</table>

5.2.3 Implémentation du prototype

Le code du prototype se trouve sur GitHub[20] dans la branche « PWA ». Il est aussi possible d'essayer ce prototype en scannant le QR code[21].

5.2.3.1 Page d'accueil

Figure 8 : Visuel de la page d'accueil



(Capture écran personnelle)

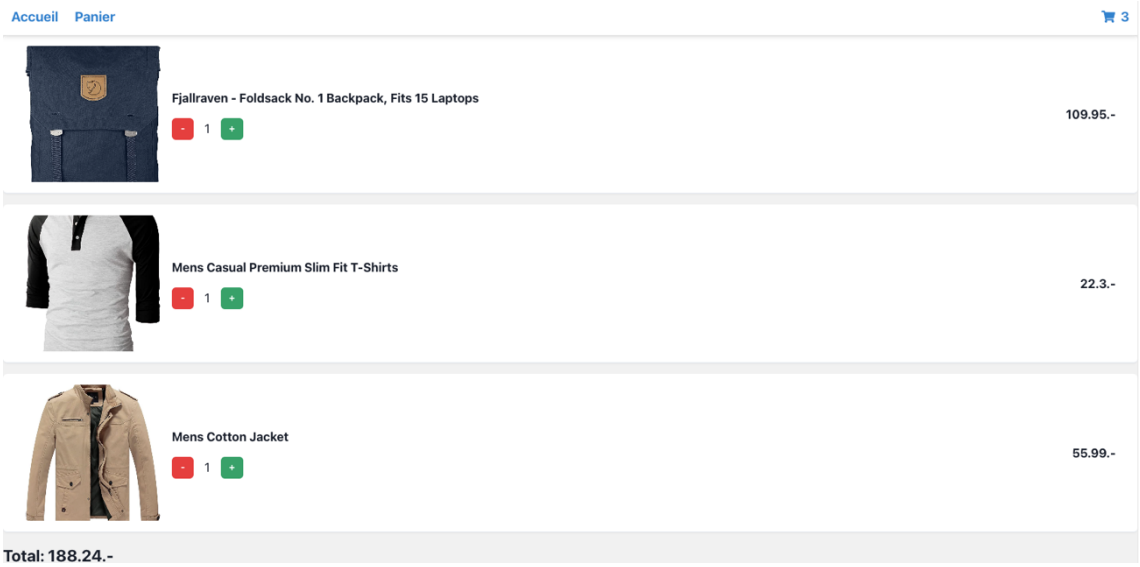
Lorsque l'utilisateur ouvrira le site web, cette page s'affichera. Tout en haut de la page se trouve la barre de navigation. Elle permet de naviguer sur les onglets que le site contient. Ensuite juste en dessous, une barre de recherche permet de rechercher spécifiquement un article. Pour finir, chaque article peut être ajouté au panier en cliquant sur le bouton « Ajouter au panier ». Une fois l'article ajouté au panier, le chiffre à côté de l'icône du chariot s'incrémente et affiche le nombre d'articles qui se trouve dans le panier.

[20] https://github.com/danielAvelino26/Bachelor_application
[21] https://pwa--elegant-concha-5f5d2c.netlify.app/

Pour consulter son panier, il existe deux possibilités. La première, cliquer directement sur l'onglet « Panier ». La seconde, il faut cliquer sur l'icône du chariot. Ensuite, un résumé du panier est affiché avec un bouton « Voir le panier », il suffit de cliquer dessus pour être dirigé vers le panier.

5.2.3.2 Panier

Figure 9 : Visuel du panier



(Capture écran personnelle)

Les divers articles contenus dans le panier sont visibles. Il est possible d'augmenter ou de diminuer la quantité d'un article en cliquant sur le bouton rouge ou vert. Tout à droite de l'image du produit est affiché le prix de l'article (prix * quantité). Pour finir, tout en bas de l'écran, cela correspond au prix total du panier.

5.2.3.3 Explication du code

5.2.3.3.1 Composant Home.js

Figure 10 : Code qui permet de fetch les articles

```
const fetchProducts = async () => {
  const productsUrl = "https://fakestoreapi.com/products";

  if (navigator.onLine) {
    try {
      const response = await fetch(productsUrl);
      if (!response.ok) {
        throw new Error("Erreur lors de la récupération des produits");
      }
      const data = await response.json();

      // Mise en cache des données récupérées
      const cache = await caches.open("products");
      cache.put(productsUrl, new Response(JSON.stringify(data)));

      return data;
    } catch (error) {
      console.error("Erreur lors de la récupération des produits:", error);
    }
  }

  // L'utilisateur est hors ligne, essayons de récupérer les données mises en cache
  const cache = await caches.open("products");
  const cachedResponse = await cache.match(productsUrl);
  if (cachedResponse) {
    const cachedData = await cachedResponse.json();
    return cachedData;
  } else {
    return []; // Renvoie un tableau vide en cas d'erreur et si aucune donnée n'est
mise en cache
  }
};
```

(Capteur d'écran personnel)

Le composant Home.js correspond à la page d'accueil. Le défi dans ce composant a été de mettre en place dans la fonction un « if » qui regarde si l'utilisateur est connecté à Internet ou non. Comme lors de la première visite, il faut être connecté à Internet, les données obtenues via la requête fetch sont mises en cache. De ce fait, lorsque la connexion sera perdue, les données qui se trouvent dans le cache seront utilisées.

5.2.3.3.2 CartContext.js

Figure 11 : Code qui permet de gérer le panier

```
import React, { createContext, useReducer } from "react";

// Context for cart state
export const CartContext = createContext();

const initialState = {
  cart: [],
};

const CartReducer = (state, action) => {
  switch (action.type) {
    case "ADD_TO_CART":
      const existingProductIndex = state.cart.findIndex(
        (item) => item.id === action.payload.id
      );

      if (existingProductIndex !== -1) {
        const updatedCart = [...state.cart];
        updatedCart[existingProductIndex].quantity += 1;
        return { ...state, cart: updatedCart };
      } else {
        return {
          ...state,
          cart: [...state.cart, { ...action.payload, quantity: 1 }],
        };
      }
    case "REMOVE_FROM_CART":
      const productIndex = state.cart.findIndex(
        (item) => item.id === action.payload
      );

      if (productIndex !== -1) {
        const updatedCart = [...state.cart];
        updatedCart[productIndex].quantity -= 1;

        if (updatedCart[productIndex].quantity === 0) {
          updatedCart.splice(productIndex, 1);
        }

        return { ...state, cart: updatedCart };
      } else {
        return state;
      }
    case "CLEAR_CART":
      return {
        ...state,
        cart: [],
      };
    default:
      return state;
  }
};

export const CartProvider = ({ children }) => {
  const [state, dispatch] = useReducer(CartReducer, initialState);

  // Pass the state and dispatch to the value of CartContext.Provider
  return (
    <CartContext.Provider value={{ state, dispatch }}>
      {children}
    </CartContext.Provider>
  );
};
```

(Capture d'écran personnelle)

Il fallait que je puisse avoir accès au nombre d'articles qui se trouvent dans le panier dans mon composant *NavBar.js* et le détail des articles pour les afficher dans le composant *Cart.js*. De ce fait, j'ai décidé de mettre en place un *Context*, qui me permet de centraliser à un endroit mon panier et de l'utiliser dans plusieurs composants.

Dans ce *Context*, j'ai dû mettre dans la fonction *CartReducer* une action qui permet d'ajouter un produit au panier. Lorsque le produit est à nouveau ajouté au panier, il faut vérifier si l'article est déjà ou non dans le panier. S'il se trouve dedans, il faut incrémenter la quantité et non le rajouter à nouveau. La même réflexion a été mise en place pour supprimer un article du panier. Il faut juste vérifier que si la quantité de l'article est à zéro, il faut le supprimer de la liste.

5.2.3.3.3 Service-worker.js

Figure 12 : Code qui permet la mise en cache des données

```
registerRoute(
  ({ url }) => url.href.startsWith("https://fakestoreapi.com/products"),
  new StaleWhileRevalidate({
    cacheName: "products",
    plugins: [
      new ExpirationPlugin({
        maxEntries: 50,
        maxAgeSeconds: 60 * 60 * 24, // 24 heures
      }),
    ],
  })
);
```

(Capteur d'écran personnel)

La mise en cache des données de l'API est gérée simplement grâce à ce code. Les données stockées dans le cache restent pour une durée de 24h.

5.2.4 Évaluation de la facilité de développement

Du fait que ReactJS est basé sur le langage JavaScript et que j'utilise dans mes projets personnels React Native et également ReactJS, la prise en main de cette bibliothèque a été plutôt facile. J'ai seulement dû me rappeler comment était géré le routing, car je ne m'en rappelais plus. Le fait de pouvoir décomposer l'application en plusieurs composants permet aussi d'avoir un code beaucoup plus facile à lire et à maintenir du fait qu'un composant fait une tâche. Ensuite, il suffit d'imbriquer les différents composants développés pour que l'application prenne forme.

Au premier abord, j'avais une crainte quant à la complexité de mettre en place un service worker. Finalement, en fouillant la documentation de ReactJS, j'ai vu qu'un template[22] existe pour les PWA. Ce template contient un service worker et il suffit ensuite d'y ajouter les différentes fonctions que l'on souhaite. Dans mon cas, j'ai dû mettre en place une fonction qui permet de mettre en cache les données qui me sont envoyées par l'API que j'utilise. Ensuite, au démarrage de l'application, il suffit de vérifier si l'utilisateur est connecté ou non à Internet pour choisir d'utiliser les données mises en cache ou bien les données envoyées par l'API.

[22] https://create-react-app.dev/docs/making-a-progressive-web-app/

J'ai également apprécié la flexibilité offerte par ReactJS en matière de gestion des états. L'utilisation d'un *Context* a facilité la gestion de l'état global de l'application sans avoir recours à des solutions plus lourdes comme Redux.

En ce qui concerne l'UI de l'application, tout a été mis en place très rapidement, car j'ai décidé d'utiliser la librairie ChakraUI[23]. Cette librairie fournit divers composants avec des styles pré-faits. Cela m'a permis finalement d'avoir un design moderne et rapide de mise en place.

5.2.5 Avantages et inconvénients de l'environnement de développement choisi

5.2.5.1 Avantages

Grâce au développement de ce prototype, je vois clairement deux avantages techniques. Le premier est le service worker qui permet la mise en cache des données et par ce fait de faire fonctionner l'application en mode hors-ligne. Dans le cas de mon site web d'e-commerce, cela peut être confortable pour un utilisateur qui perd la connexion de toujours pouvoir consulter le catalogue de produits.

Le deuxième avantage réside dans la possibilité d'avoir un site web pouvant être utilisé à tout moment de la même manière qu'une application téléchargeable depuis l'App Store ou le Play Store. Cela permet aux utilisateurs d'avoir directement une icône de l'application sur le bureau et il n'est plus obligatoire d'aller sur son navigateur Internet et de taper l'URL du site web.

Un autre avantage notable est la réutilisabilité des composants dans ReactJS. Ceci réduit la duplication du code, facilite la maintenance, et accélère le processus de développement. De plus, le support de la communauté ReactJS est un atout précieux. L'écosystème est très actif, avec une multitude de ressources disponibles pour aider à résoudre les problèmes qui peuvent survenir.

5.2.5.2 Désavantages

Selon moi, et après mûre réflexion à la suite de la mise en place du prototype, je ne trouve que des avantages à cette architecture.

[23] https://github.com/chakra-ui/chakra-ui

5.3 Prototype cross-plateforme

5.3.1 Choix de l'environnement de développement

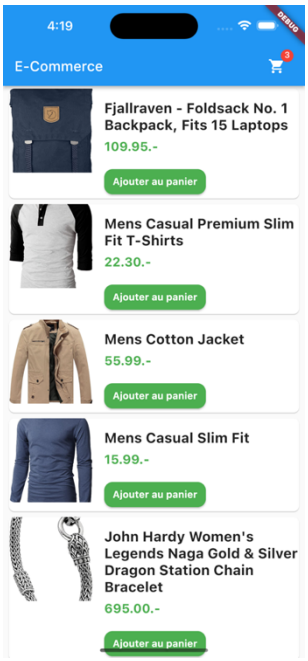
J'ai décidé de développer le prototype de l'application cross-plateforme grâce à l'environnement de développement Flutter.

5.3.2 Implémentation du prototype

Le code du prototype se trouve sur GitHub[24] dans la branche « crossplateforme ». Il est aussi possible d'essayer ce prototype en scannant le QR code[25].

5.3.2.1 Page d'accueil

Figure 13 : Visuel de la page d'accueil



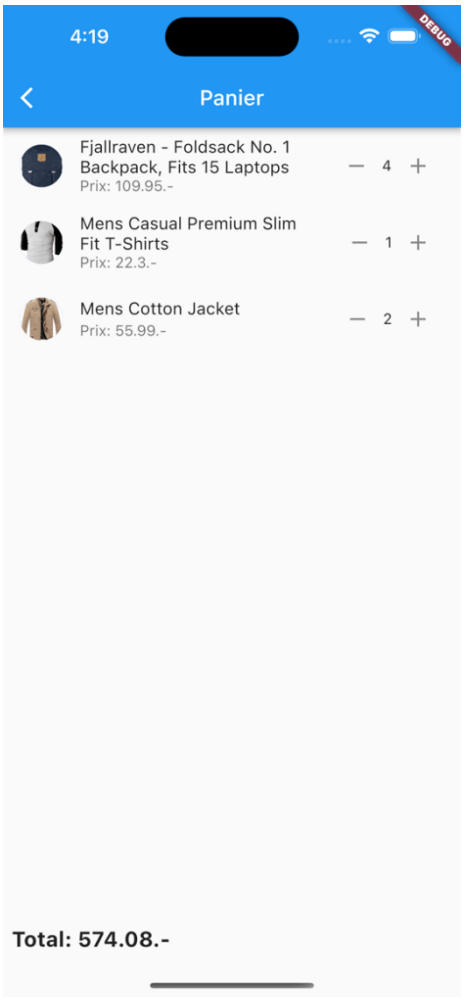
(Capture écran personnelle)

[24] https://github.com/danielAvelino26/Bachelor_application
[25] https://cross-plateforme--visionary-cocada-0a5a8b.netlify.app/

Au lancement de l'application, cette page sera affichée à l'utilisateur. Elle permet à l'utilisateur d'ajouter à son panier les articles qui l'intéressent en appuyant sur le bouton « Ajouter au panier ». Une fois les produits ajoutés au panier, il est possible de le consulter en cliquant sur l'icône qui représente un caddie.

5.3.2.2 Panier

Figure 14 : Visuel du panier



(Capture écran personnelle)

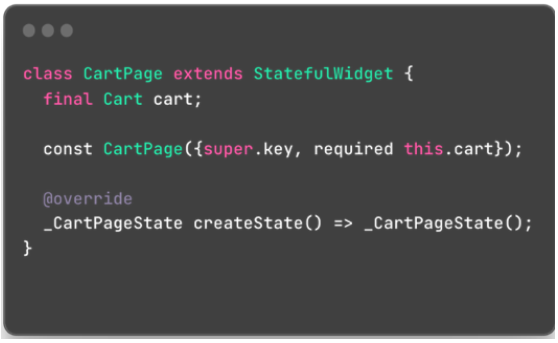
Cette page permet à l'utilisateur de visualiser ce qui se trouve dans son panier. De plus, il est possible d'augmenter la quantité de chaque article ou bien de diminuer la quantité. Le total est aussi affiché tout en bas de l'écran.

5.3.2.3 Explication du code

5.3.2.3.1 Cart_page.dart

La classe *CartPage* est définie comme un *StatefulWidget*. Cela signifie qu'elle possède un état qui peut évoluer dans le temps. Dans le cas présent, cet état est géré par la classe *_CartPageState*, qui est une sous-classe privée de *CartPage*. De plus, cette classe accepte une instance de *Cart* en tant que paramètre obligatoire.

Figure 15 : Classe CartPage



(Capture d'écran personnelle)

Figure 16 : Code qui permet de gérer le visuel du panier

```dart
class _CartPageState extends State<CartPage> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Panier')),
      body: ValueListenableBuilder<int>(
        valueListenable: widget.cart.itemCountNotifier,
        builder: (context, value, child) {
          return widget.cart.items.isNotEmpty
              ? ListView.builder(
                  itemCount: widget.cart.items.length,
                  itemBuilder: (context, index) {
                    var item = widget.cart.items[index];
                    return ListTile(
                      leading: CircleAvatar(
                        backgroundImage: NetworkImage(item.product.image),
                      ),
                      title: Text(item.product.title),
                      subtitle: Text('Prix: ${item.product.price}.-'),
                      trailing: Row(
                        mainAxisSize: MainAxisSize.min,
                        children: [
                          IconButton(
                            icon: const Icon(Icons.remove),
                            onPressed: () {
                              widget.cart.remove(item.product);
                            },
                          ),
                          Text('${item.quantity}'),
                          IconButton(
                            icon: const Icon(Icons.add),
                            onPressed: () {
                              widget.cart.add(item.product);
                            },
                          ),
                        ],
                      ),
                    );
                  },
                )
              : const Center(child: Text('Votre panier est vide.'));
        },
      ),
      bottomNavigationBar: widget.cart.items.isNotEmpty
          ? Padding(
              padding: const EdgeInsets.all(8.0),
              child: ValueListenableBuilder<int>(
                valueListenable: widget.cart.itemCountNotifier,
                builder: (context, value, child) {
                  return SafeArea(
                    child: Text(
                      'Total: ${widget.cart.getTotalPrice().toStringAsFixed(2)}.-',
                      style: const TextStyle(
                          fontSize: 20, fontWeight: FontWeight.bold),
                    ),
                  );
                },
              ),
            )
          : null,
    );
  }
}
```

(Capture écran personnelle)

L'interface utilisateur de la page de panier est construite en utilisant plusieurs widgets Flutter. Un widget notable est le *ValueListenableBuilder*, qui écoute les modifications de la valeur de *itemCountNotifier*. Lorsque cette valeur change, le *ValueListenableBuilder* reconstruit une partie de l'arbre des widgets, évitant ainsi de devoir reconstruire l'ensemble de l'arbre et améliorant ainsi les performances de l'application.

Le widget *ListView.builder* est utilisé pour afficher la liste des articles dans le panier. *ListView.builder* est une approche efficace pour afficher des listes d'éléments, car il ne construit que les éléments actuellement visibles à l'écran, ce qui est particulièrement utile pour les longues listes.

Les widgets *IconButton* sont utilisés pour fournir une interface utilisateur interactive. En appuyant sur ces boutons, les fonctions *add* et *remove* de l'objet *Cart* sont appelées.

Enfin, le widget *SafeArea* est utilisé pour garantir que le contenu est affiché dans une zone sécurisée. Cela prend en compte divers facteurs tels que les encoches de l'écran et les barres de système, assurant ainsi une expérience utilisateur optimale, quel que soit l'appareil utilisé.

5.3.2.3.2 Store.dart

La classe *Store* utilise le design pattern Singleton. En effet, elle utilise un constructeur privé et une instance statique pour garantir qu'une seule instance de *Store* est créée tout au long de la durée de vie de l'application. Cela est particulièrement utile pour gérer un état global qui doit être partagé entre plusieurs parties de l'application, comme c'est le cas ici avec la liste des produits.

Figure 17 : Classe Store

```dart
class Store {
  // Private Constructor and instance declarations
  Store._privateConstructor();
  static final Store _instance = Store._privateConstructor();
  factory Store() {
    return _instance;
  }
```

(Capture écran personnelle)

La classe *Store* utilise également un *StreamController* pour gérer un flux de données de type *Store*. Ce flux est utilisé pour informer les widgets à l'écoute des changements d'état du *Store*, tels que la mise à jour de la liste des produits.

La classe *Store* expose également une méthode *builder* qui renvoie un *StoreStreamBuilder*. Ce dernier est un widget personnalisé qui écoute les changements du Store et reconstruit ses enfants en conséquence.

La méthode *fetchProducts* de la classe *Store* utilise le package HTTP pour récupérer une liste de produits à partir d'une API distante. Elle décode la réponse JSON et crée une liste de produits à partir des données obtenues.

Figure 18 : Code qui permet de fetch les articles

```dart
Future<List<Product>> fetchProducts() async {
  final response =
      await http.get(Uri.parse('https://fakestoreapi.com/products/'));

  if (response.statusCode == 200) {
    // If the server did return a 200 OK response,
    // then parse the JSON.
    var products = (jsonDecode(response.body) as List)
        .map((e) => Product.fromJson(e))
        .toList();
    return products;
  } else {
    // If the server did not return a 200 OK response,
    // then throw an exception.
    throw Exception('Failed to load album');
  }
}
```

(Capture écran personnelle)

Enfin, la méthode *getProducts* appelle *fetchProducts* pour obtenir une liste de produits, met à jour le champ *products* du Store avec cette liste, et informe ensuite les « écouteurs » du changement d'état en appelant la méthode *update*.

5.3.2.3.3 Cart.dart

La classe *Cart* contient une liste items de *CartItem*, chaque *CartItem* représente un produit spécifique dans le panier avec sa quantité. La classe *Cart* expose également un *ValueNotifier*, *itemCountNotifier*, qui est utilisé pour informer les « écouteurs » des modifications du nombre total d'articles dans le panier.

La méthode *_updateItemCount* met à jour la valeur de *itemCountNotifier* en sommant les quantités de tous les *CartItem* dans le *Cart*.

Les méthodes *add* et *remove* permettent d'ajouter ou de supprimer un produit du panier. Elles recherchent d'abord l'article correspondant dans le panier en utilisant la méthode *firstWhereOrNull*, qui renvoie le premier élément qui satisfait la condition fournie ou null si aucun élément ne la satisfait. Si l'élément est trouvé, sa quantité est incrémentée ou décrémentée, sinon, un nouvel élément est ajouté au panier. Après chaque modification, *_updateItemCount* est appelée pour mettre à jour le compteur d'articles.

Figure 19 : Code qui permet d'ajouter ou supprimer un article du panier

```
void add(Product product) {
    var item =
        items.firstWhereOrNull((element) => element.product.id == product.id);

    if (item == null) {
      items.add(CartItem(product, 1));
    } else {
      item.quantity++;
    }
    _updateItemCount();
  }

  void remove(Product product) {
    var item =
        items.firstWhereOrNull((element) => element.product.id == product.id);

    if (item != null) {
      if (item.quantity == 1) {
        items.remove(item);
      } else {
        item.quantity--;
      }
    }
    _updateItemCount();
  }
```

(Capture écran personnelle)

Enfin, la méthode *getTotalPrice* calcule le coût total des articles dans le panier en multipliant le prix de chaque produit par sa quantité et en sommant les résultats.

5.3.3 Évaluation de la facilité de développement

Ayant par le passé voulu essayer Flutter, mais n'ayant jamais vraiment le temps pour, ce prototype m'a permis de m'essayer à Flutter. En revanche, tout ne sait pas passer comme je l'imaginais. La mise en place de ce prototype a été plus complexe que la PWA. Cela s'explique par le fait que je ne connaissais absolument pas le langage Dart, ni Flutter. En comparaison à ReactJS, cela a été plus laborieux. Au début, j'ai eu un peu de mal avec la syntaxe de Dart qui ressemble à du Java et du JavaScript mélangé. En revanche, au fur et à mesure que je passais du temps sur le projet, la syntaxe me dérangeait de moins en moins, bien que je ne sois toujours pas très à l'aise avec.

Dart est un langage de programmation basé sur le concept d'objets. Étant familier avec le langage Java, l'idée de classes ne m'a pas posé de souci. Cependant, pour ceux qui ne sont pas initiés à la programmation orientée objet, cela peut s'avérer déconcertant et nécessite un certain temps d'apprentissage. Par conséquent, de mon point de vue, pour le développement d'une application à partir de zéro sans aucune connaissance préalable, la courbe d'apprentissage serait plus longue pour Flutter par rapport à ReactJS.

Durant le développement de ce prototype, j'ai été confronté à un problème qui m'a pris plusieurs jours à le corriger. Cela était la gestion des états. J'ai, par exemple, rencontré le problème suivant : lorsque j'ajoutais un produit au panier, dans la console je voyais l'article dans le panier. En revanche, quand j'allais dans le panier, il n'y avait aucun article. En regardant la documentation, j'ai pu constater qu'il fallait rajouter une méthode setState() sur mon bouton pour rafraîchir l'interface utilisateur et ainsi voir le changement. Cette solution avait déclenché un nouveau problème. A chaque ajout d'un produit au panier, la page se rafraîchissait complètement. Pour contrer cela, j'ai dû enlever la méthode setState(). A la place, dans mon widget qui contient ma méthode pour ajouter un produit au panier, il a fallu utiliser un ValueListenableBuilder.

Il faut souligner que lorsque j'ai eu le moindre bug lors du développement, j'ai souvent trouvé les réponses à mes questions sur StackOverflow. En ce qui concerne la documentation officielle, elle est assez fournie. Je l'ai beaucoup utilisée pour les widgets.

5.3.4 Avantages et inconvénients de l'environnement de développement choisi

Dart étant un langage de programmation orienté objet et typé, cela offre certains avantages. Cela permet d'avoir un code réutilisable et modulable comme ReactJS. Le code est aussi, je trouve plus lisible qu'avec ReactJS, grâce aux classes. De plus, l'héritage, le polymorphisme et l'encapsulation offrent des mécanismes puissants pour structurer et organiser les applications de manière cohérente.

En outre, Dart étant un langage typé, ce qui signifie que les variables, les constantes et les paramètres de fonction ont des types spécifiques, tels que String, int ou double. Cette caractéristique présente plusieurs avantages. Tout d'abord, elle permet de détecter les erreurs, ce qui réduit les erreurs potentielles lors de l'exécution du programme. Ensuite, les types aident à améliorer la lisibilité et la compréhension du code, car ils fournissent des informations supplémentaires sur la nature des données manipulées.

En termes de performances, Flutter compile directement en code natif, ce qui offre de meilleures performances par rapport à React Native[26], qui nécessite un pont JavaScript pour interagir avec les composants natifs.

En ayant déjà développé plusieurs projets en React Native, si je devais choisir un environnement de développement pour des applications cross-plateforme, cela serait sans hésitation React Native si l'application n'effectue pas beaucoup de calculs. A l'inverse, si l'application effectue beaucoup de calculs, Flutter en termes de performance est loin devant React Native. Suite à ma propre expérience de mise en place du prototype avec Flutter, je peux affirmer que l'environnement de développement de Flutter est beaucoup plus compliqué à prendre en main que React Native et que les temps de développement sont aussi plus longs. De plus, l'utilisation de TypeScript avec React Native permet de passer d'un langage non typé comme JavaScript à un langage typé, ce qui offre certains avantages. En revanche, si l'application est conséquente Flutter peut être une meilleure solution, car la compréhension du code est facilitée grâce à l'orienté objet.

[26] https://medium.com/swlh/flutter-vs-native-vs-react-native-examining-performance-31338f081980

5.4 Prototype natif

5.4.1 Choix de l'environnement de développement

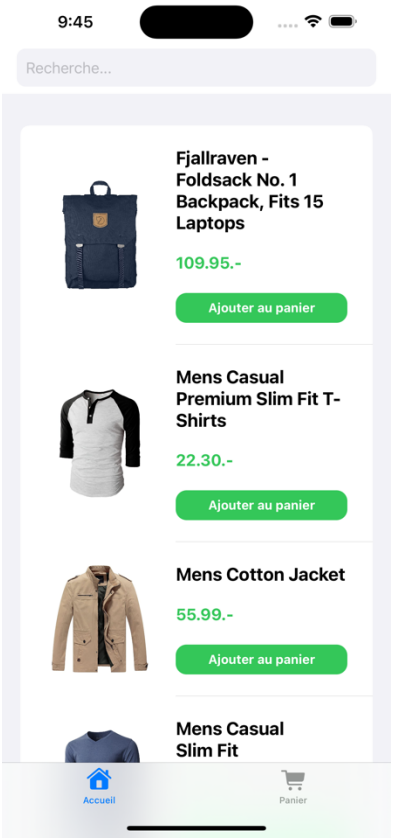
J'ai décidé de développer le prototype de l'application native grâce à l'environnement de développement SwiftUI.

5.4.2 Implémentation du prototype

Le code du prototype se trouve sur GitHub[27] dans la branche « native ».

5.4.2.1 Page d'accueil

Figure 20 : Visuel de la page d'accueil



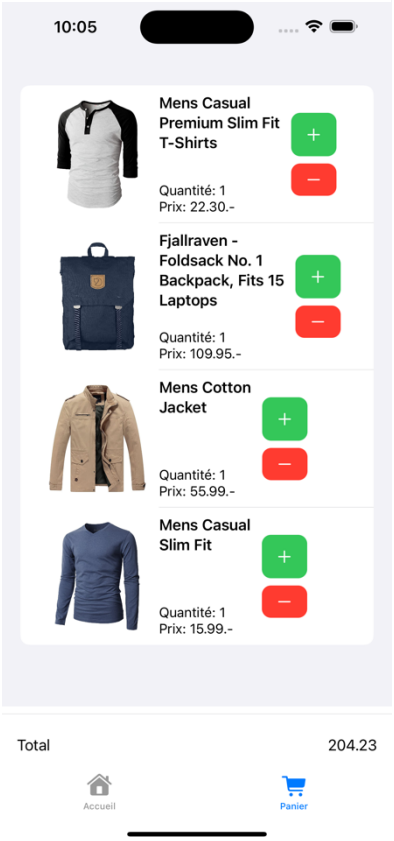
(Capture écran personnelle)

A l'ouverture de l'application, la page d'accueil sera affichée. Elle permet de consulter les différents articles et de les ajouter à son panier. De plus, tout en haut, se trouve une barre de recherche qui permet de filtrer les articles. Pour finir, tout en bas l'icône en forme de maison et celle en forme de panier permettent de naviguer dans l'application.

[27] https://github.com/danielAvelino26/Bachelor_application

5.4.2.2 Panier

Figure 21 : Visuel du panier



(Capture écran personnelle)

Grâce à cette page, il est possible de consulter les différents articles qui se trouvent dans le panier. De plus, si l'on désire augmenter la quantité ou la diminuer d'un article, il est possible grâce aux boutons « - » et « + ». Ensuite, en bas de l'écran, se trouve le montant total du panier.

5.4.2.3 Explication du code

5.4.2.3.1 CardProduct

Le composant *CardProduct* représente une carte de produit dans l'application. Il va afficher une image du produit, son titre, son prix et un bouton pour ajouter le produit au panier.

Il reçoit comme argument un *Product*. Ce produit est utilisé pour utiliser les différentes données citées au-dessus pour les afficher à l'utilisateur. Le composant a également accès à un objet *Cart* partagé, qui est injecté dans l'environnement par SwiftUI et qui permet d'ajouter des produits au panier.

Dans le corps de la vue, un *HStack* est utilisé pour organiser horizontalement l'image du produit et les détails du produit. L'image est chargée de manière asynchrone à l'aide d'*AsyncImage*, qui prend une URL et charge l'image en arrière-plan. L'image est ensuite redimensionnée pour s'adapter à son conteneur tout en conservant son aspect ratio, et elle est coupée pour avoir des coins arrondis.

Figure 22 : Conteneur qui affiche l'image du produit

```
var body: some View {
        HStack {
            AsyncImage(url: URL(string: product.image)) { image in
                image
                    .resizable()
                    .aspectRatio(contentMode: .fit)
                    .clipShape(RoundedRectangle(cornerRadius: 8, style: .continuous))
            } placeholder: {
                Color.gray
            }
            .frame(width: 120, height: 120)
        }
```

(Capture d'écran personnelle)

À droite de l'image, un *VStack* est utilisé pour organiser verticalement le titre du produit, son prix et le bouton d'ajout au panier. Le titre et le prix du produit sont affichés à l'aide de vues Text. Le bouton d'ajout au panier est une vue Button qui, lorsqu'on clique dessus, appelle la méthode addToCart de l'objet Cart avec le produit actuel.

Figure 23 : Conteneur qui affiche le titre, son prix et un bouton

```
var body: some View {
        HStack {
            VStack(alignment: .leading) {
                Text(product.title)
                    .font(.system(size: 20))
                    .fontWeight(.bold)
                    .padding(.top, 16)
                Text(String(format: "%.2f.-", product.price))
                    .font(.system(size: 18))
                    .fontWeight(.bold)
                    .foregroundColor(.green)
                    .padding(.top, 8)

                Spacer()

                Button(action: {
                    cart.addToCart(product: product)
                }) {
                    Text("Ajouter au panier")
                        .font(.system(size: 14))
                        .fontWeight(.bold)
                        .frame(maxWidth: .infinity)
                        .padding(.vertical, 8)
                        .padding(.horizontal, 12)
                        .background(Color.green)
                        .foregroundColor(.white)
                        .cornerRadius(12)
                }.buttonStyle(BorderlessButtonStyle())

                Spacer().frame(height: 16)
            }
        }
```

(Capture écran personnelle)

5.4.2.3.2 CardProductCart

Le composant *CardProductCart* représente une carte de produit spécifique pour le panier d'achats. Il utilise un produit de type *CartItem* comme argument. *CartItem* est une structure qui contient un produit et une quantité. Le composant a aussi accès à un objet partagé *Cart*, utilisé pour gérer les opérations sur le panier d'achats.

Le corps de la vue est organisé horizontalement avec un *HStack*. Il comporte l'image du produit, ses détails, et deux boutons « + » et « - » pour ajuster la quantité du produit dans le panier. L'image du produit est chargée de manière asynchrone, tout comme dans le composant *CardProduct*.

Ensuite, un *VStack* est utilisé pour afficher verticalement le titre du produit, la quantité et le prix. Les vues *Text* sont utilisées pour cela.

La dernière partie de *HStack* est un autre *VStack* qui contient deux boutons, « + » et « - », représentés par les composants *PlusButton* et *MinusButton*. Ces deux boutons sont des vues *Button* qui, en cas de clic, appellent les méthodes *addToCart* et *removeFromCart* de l'objet *Cart* avec le produit actuel.

Figure 24 : Conteneur qui permet d'afficher le prix et la quantité et aussi deux boutons

```
var body: some View {
        HStack {
            VStack(alignment: .leading) {
                Text(product.product.title)
                    .font(.headline)
                Spacer()
                Text("Quantité: \(product.quantity)")
                    .font(.subheadline)
                Text("Prix: \(product.product.price, specifier: "%.2f").-")
                    .font(.subheadline)
            }

            VStack {
                PlusButton(cart: cart, product: product)
                MinusButton(cart: cart, product: product)
            }
            .frame(width: 50)
        }
        .padding(5)
        .cornerRadius(10)
    }
```

(Capture écran personnelle)

5.4.2.3.3 Cart

La classe *Cart* représente un panier d'achats. La classe est de type *ObservableObject*, ce qui signifie que SwiftUI peut observer les modifications de cette classe et mettre à jour l'interface utilisateur en conséquence.

La variable products de cette classe est un tableau d'objets *CartItem*. De plus, elle est de type *@Published*, ce qui signifie que chaque fois que cette propriété change, SwiftUI est informé et peut mettre à jour l'interface utilisateur.

La méthode *addToCart* prend un *Product* comme argument. Elle vérifie d'abord si le produit est déjà dans le panier en recherchant le premier élément dans le tableau products dont l'ID correspond à l'ID du produit. Si le produit est trouvé, sa quantité est augmentée de 1. Sinon, un nouvel objet *CartItem* est créé avec une quantité de 1 et est ajouté à la liste.

Figure 25 : Fonction pour ajouter un article au panier

```swift
func addToCart(product: Product) {
        if let index: Int = products.firstIndex(where: {$0.product.id == product.id}) {
            products[index].quantity += 1
        } else {
            products.append(CartItem(product: product, quantity: 1))
        }
    }
```

(Capture écran personnelle)

La méthode *removeToCart* fonctionne de manière similaire à *addToCart*, mais elle diminue la quantité du produit dans le panier ou supprime complètement le produit si la quantité est de 1.

Figure 26 : Fonction pour supprimer un article du panier

```swift
func removeToCart(product: Product) {
        if let index: Int = products.firstIndex(where: {$0.product.id == product.id}) {
            if (products[index].quantity == 1) {
                products.remove(at: index)
            } else {
                products[index].quantity -= 1
            }
        } else {
            print("Ce produit n'existe pas dans le panier")
        }
    }
```

(Capture écran personnelle)

Enfin, la fonction *getTotalAmount* calcule le montant total du panier. Elle parcourt chaque produit dans le tableau products et ajoute le prix du produit multiplié par sa quantité au total. Le total est arrondi à deux décimales et renvoyé.

5.4.2.3.4 APIRequest

La classe *APIRequest* permet l'envoi des requêtes à une API. Elle de type *ObservableObject*.

La variable results contient les produits obtenus de l'API. La variable *errorMessage* stocke un éventuel message d'erreur lors de l'appel à l'API.

La méthode *callAPI* est utilisée pour envoyer une requête à l'API. Elle commence par définir l'URL de l'API et tente de la convertir en une instance d'URL. Si elle réussit, elle utilise *URLSession.shared.dataTask(with:completionHandler:)* pour envoyer la requête.

Dans *URLSession.shared.dataTask(with:completionHandler:)* plusieurs cas sont gérés :

1. Une erreur de réseau se produit. Dans ce cas, le message d'erreur est mis à jour avec une description de l'erreur.
2. La réponse HTTP reçue n'est pas un code de succès (200-299). Le message d'erreur est mis à jour avec le code de statut HTTP.
3. Des données sont reçues. Dans ce cas, le code tente de décoder les données JSON en un tableau de produits. Si le décodage réussit, les résultats sont mis à jour et le message d'erreur est réinitialisé à nil. Si le décodage échoue, le message d'erreur est mis à jour avec une description de l'erreur de décodage.
4. Enfin, si un message d'erreur est défini à la fin de l'appel à l'API, les résultats sont réinitialisés à un tableau vide.

Figure 27 : Classe qui permet de faire une requête à une API

```swift
class APIRequest: ObservableObject {
    @Published var results: [Product] = []
    @Published var errorMessage: String? = nil

    func callAPI() {
        // Définition de l'URL de l'API en tant que chaîne de caractères
        let urlString: String = "https://fakestoreapi.com/products"
        // Essayer de convertir la chaîne de caractères en URL
        if let url: URL = URL(string: urlString) {
            URLSession.shared.dataTask(with: url) { data, response, error in
                DispatchQueue.main.async {
                    if let error = error {
                        self.errorMessage = "Erreur réseau : \(error.localizedDescription)"
                    } else if let httpResponse = response as? HTTPURLResponse, !
(200...299).contains(httpResponse.statusCode) {
                        self.errorMessage = "Erreur serveur : \(httpResponse.statusCode)"
                    } else if let data = data {
                        do {
                            let products = try JSONDecoder().decode([Product].self, from:
data)
                            self.results = products
                            self.errorMessage = nil
                        } catch {
                            self.errorMessage = "Erreur de décodage : \
(error.localizedDescription)"
                        }
                    }
                    if self.errorMessage != nil {
                        self.results = []
                    }
                }
            }.resume()
        }
    }
}
```

(Capture écran personnelle)

5.4.3 Évaluation de la facilité de développement

Le développement de ce prototype a été la plus grande surprise pour moi. Avant de commencer à coder les différents prototypes, j'avais des a priori par rapport à ce prototype. Je n'avais pas spécialement envie d'utiliser Swift, car des connaissances m'avaient dit que cela n'allait pas être simple. A cause de cela, j'ai réalisé ce prototype en dernier.

Pour le développement de ce prototype, je suis parti de zéro comme pour Flutter. J'ai dû tout d'abord apprendre les bases de Swift. Ensuite, j'ai pu commencer mon apprentissage avec SwiftUI. En comparaison à Flutter qui a été pour moi l'environnement de développement le plus difficile à appréhender, j'ai réussi à prendre en main cet environnement comme si j'avais l'impression de l'avoir déjà utilisé alors que cela n'est pas le cas. La courbe d'apprentissage a été rapide.

En termes de durée de développement, ce prototype m'a pris exactement le même temps que pour la PWA.

Le point le plus compliqué lors du développement a été la classe *APIRequest*. Je n'avais jamais été confronté à avoir un code de ce style pour effectuer un appel à une API. En revanche, une fois différentes ressources lues, cela est devenu plus clair pour le mettre en place. A contrario le point qui m'a facilité le développement a été la variable de type *@Environnement*. Cette annotation permet de créer des variables partagées accessibles dans toute l'application. Cela fonctionne de manière similaire au *Context* dans React. SwiftUI permet de mettre en place cela de manière plus facile que React et d'une manière intuitive.

Concernant la documentation, je l'ai trouvé complète, mais elle est mal structurée. Ce qui fait que pour trouver l'information que l'on cherche cela prend du temps. Plus d'une fois, j'ai consulté la documentation et comme je ne comprenais pas forcément ou ne trouvais pas, je devais chercher d'autres ressources.

Finalement, les a priori que j'ai pu avoir n'ont pas eu raison d'être, car c'est le prototype sur lequel j'ai le plus aimé travailler.

5.4.4 Avantages et inconvénients de l'environnement de développement choisi

5.4.4.1 Avantages

Swift étant un langage typé, dans mon cas ayant l'habitude de développer avec JavaScript souvent, j'oubliais de typer les variables, mais Xcode m'affichait directement un warning pour m'informer de mon oubli. Le typage permet de détecter directement lors du développement si lors de l'assignation d'une variable la valeur que lui est assigné est correct. Cela permet de gagner du temps et de prévenir les erreurs.

La conception d'interfaces utilisateur de SwiftUI a été très bien pensée, ce qui facilite la mise en place du style dans l'application. Cela ressemble aux widgets de Flutter, mais l'implémentation de SwiftUI est plus lisible et plus facile d'utilisation.

De plus, SwiftUI est dit « déclaratif » ce qui pour moi est plus intuitif car, j'utilise pour mes projets ReactJS et React Native qui sont eux aussi « déclaratif ». Ensuite, comme pour ReactJS et React Native SwiftUI, utilise aussi les composants. De mon point de vue, pour la maintenance du code cela est bénéfique, car chaque composant est indépendant.

5.4.4.2 Désavantages

De mon point de vue le plus grand désavantage est qu'il est obligatoire de développer une application iOS et une autre Android. Du point de vue des petites entreprises, voire des moyennes entreprises, cela représente un énorme coup qu'elles ne peuvent peut-être pas se permettre. Il faut payer deux développeurs et ensuite maintenir ces deux applications. Ensuite, il faut obligatoirement avoir un Mac ce qui est limitant je trouve. En comparaison Android permet le développement d'application mobile sur Mac ou Windows. De plus, il faut aussi prendre en compte que ce framework étant sorti en 2019 est encore jeune. En témoigne le nombre de répository Github au nombre de 52'500 alors que Swift en compte 223'000.

5.5 Ressenti performance

Dans ce paragraphe, je partagerai mon ressenti personnel en ce qui concerne l'utilisation des trois prototypes : PWA, cross-plateforme et native. Il est important de noter que les impressions que je décrirai ne sont pas les résultats de performances pures mesurés dans le code, mais plutôt du ressenti visuel de performances pour chacun des prototypes.

Lorsque la PWA est ouverte, tous les produits sont chargés très rapidement grâce au service worker. J'ai l'impression que le temps de chargement est un peu plus rapide que le prototype développé avec Flutter. De plus, si on quitte le site web et que l'on y revient quelques heures après, comme tout a été mis en cache, il n'y a quasiment aucun temps de chargement. C'est le prototype le plus rapide pour afficher tous les produits. Le prototype natif, quant à lui, affiche tous les produits à la même vitesse que le prototype Flutter (cross-plateforme). En ce qui concerne l'ajout des articles et le panier, la performance est identique pour les trois prototypes.

Pour finir, lors de la recherche d'un produit, la PWA possède un avantage par rapport aux images. Quand on commence à écrire par exemple « sam », sur tous les autres prototypes, on voit que les images prennent quelques millisecondes à se réafficher. Sur la PWA, les images restent affichées en tout temps.

Tableau 8 : Vue d'ensemble ressenti performance

	PWA (ReactJS)	**Cross-plateforme (Flutter)**	**Natif (SwiftUI)**
Temps de chargement des produits	Très rapide grâce au service worker	Un peu plus lent que la PWA	Comparable à Flutter
Temps de chargement après mise en cache	Presque instantané		
Ajout d'articles au panier	Aucun délai perceptible	Aucun délai perceptible	Aucun délai perceptible
Recherche de produits	Images restent affichées	Images prennent du temps à s'afficher	Images prennent du temps à s'afficher

6. Conclusion

Lorsque l'objectif est de développer une application mobile, une question surgit fréquemment : *« Quels Frameworks et technologies pour le développement d'application mobile ? ».* Cette problématique englobe la nécessité de concilier la performance des applications, l'optimisation des temps de développement et le contrôle des coûts.

Le travail de Bachelor présenté ici a abordé ce problème en développant des prototypes pour trois types d'applications mobiles : natives, cross-plateformes et Progressive Web Apps (PWA). Pour chaque prototype, plusieurs langages et environnements de développement ont été explorés, notamment Swift avec Xcode, JavaScript avec React Native et Dart avec Flutter.

Il ressort que chaque environnement de développement a ses avantages et ses inconvénients. Grâce aux développements de ces trois prototypes, j'ai pu me rendre compte de certains aspects importants dans le choix du framework et de la technologie à choisir. Quand on souhaite développer une application, il faut qu'elle soit performante. De ce fait, potentiellement, on peut se dire que l'on doit faire du natif. De mon point de vue, cela serait une erreur, car comme décrit dans le chapitre 5.5 Ressenti performance, lorsque l'utilisateur utilise chacune des applications en termes de performances, les différences ne sont quasiment pas visibles, il faut réellement y prêter attention. Cela veut dire, que si l'application envisagée ne demande pas beaucoup de ressources au téléphone, la performance ne sera pas le critère le plus important.

A contrario, si beaucoup de calculs doivent être effectués par le téléphone, il faudra prendre en compte ce critère au risque de se retrouver avec une application lente. De ce fait, dans ce type de cas, l'application native est le meilleur choix. En revanche, il faudra accepter de devoir développer une application pour iOS et une autre pour Android. En outre, si l'accès aux dernières nouveautés proposées par Apple aux développeurs est un critère essentiel, comme par exemple le Dynamic Island, le choix d'une application native devient alors incontournable.

Il est aussi à prendre en compte que Flutter possède des performances assez proches des applications natives. De ce fait, si l'application nécessite de la performance, Flutter peut être également choisi. De plus, Flutter utilise les widgets Material Components pour Android[28] et les Cupertino widgets[29] qui permettent à l'application d'avoir un design dit

[28] https://docs.flutter.dev/ui/widgets/material
[29] https://docs.flutter.dev/ui/widgets/cupertino

natif. En revanche, les dernières nouveautés iOS et Android ne seront pas proposées directement non plus.

Concernant les PWA, ce type d'application se révèle idéal pour ceux désireux d'avoir à la fois un site web et une application mobile. De plus, grâce au service worker pour les applications de type blog, cela est un point fort, car les utilisateurs peuvent utiliser l'application malgré une perte de réseau. Ensuite, en termes de prix de développement, cela est aussi avantageux[30]. En revanche, avant de se lancer dans le développement, il vaut mieux vérifier les fonctionnalités que l'application implémentera pour ne pas avoir de surprise par la suite. Par exemple les notifications ne sont pas disponibles sur iOS.

Pour finir, les applications cross-plateforme en termes de budget sont une alternative aux applications natives qui coûtent le plus cher. De plus, l'application fonctionnera sur iOS et Android. Du fait que ce type d'application fonctionne sur iOS et Android, elles se prêtent bien au MVP et par la suite si le projet est validé, il est possible de continuer le développement de l'application[31]. Si Flutter est choisi, il est aussi possible d'avoir un site web grâce au même code que l'application mobile.

Type d'application	**Performance**	**Coût de développement**	**Fonctionnalités spécifiques**	**Autres considérations**
Native	Très performante (meilleure pour les applications nécessitant de nombreux calculs)	Élevé (nécessité de développer une version pour iOS et une version pour Android)	Accès à toutes les fonctionnalités du système d'exploitation	
PWA	Performante (proche des applications natives)	Faible (un seul code pour toutes les plateformes)	Fonctionnalités limitées (pour iOS)	Parfait pour un site web et une application mobile Permet l'utilisation hors-ligne grâce au service worker
Cross-plateforme	Performante (proche des applications natives, si développé avec Flutter)	Modéré (un seul code pour iOS et Android)	Accès à la plupart des fonctionnalités du système d'exploitation	Idéal pour les MVP Possibilité d'avoir un site web avec le même code que l'application mobile si Flutter est utilisé

[30] https://medium.com/neoxia/pwa-vs-flutter-vs-react-native-vs-native-dc06a17ebf1a
[31] https://medium.com/neoxia/pwa-vs-flutter-vs-react-native-vs-native-dc06a17ebf1a

Bibliographie

FIRTMAN, Maximiliano, [sans date]. iOS PWA Compatibility—firt.dev. [en ligne]. Disponible à l'adresse : https://firt.dev/notes/pwa-ios/ [consulté le 17 juillet 2023].

React Native, 2023*Wikipédia* [en ligne]. Disponible à l'adresse : https://fr.wikipedia.org/w/index.php?title=React_Native&oldid=203482706 [consulté le 17 juillet 2023]. Page Version ID: 203482706

Native Modules Intro · React Native, 2023 [en ligne]. Disponible à l'adresse : https://reactnative.dev/docs/native-modules-intro [consulté le 17 juillet 2023].

React, [sans date] [en ligne]. Disponible à l'adresse : https://react.dev/ [consulté le 17 juillet 2023].

DOM virtuel et autres détails – React, [sans date] [en ligne]. Disponible à l'adresse : https://fr.legacy.reactjs.org/docs/faq-internals.html [consulté le 17 juillet 2023].

PAUL, Payal, 2020. 10 Main Core Concept You Need to Know About React. *Medium* [en ligne]. 4 novembre 2020. Disponible à l'adresse : https://payalpaul2436.medium.com/10-main-core-concept-you-need-to-know-about-react-303e986e1763 [consulté le 17 juillet 2023].

MOJEED, Ibadehin, 2022. What is the virtual DOM in React? *LogRocket Blog* [en ligne]. 16 août 2022. Disponible à l'adresse : http://blog.logrocket.com/virtual-dom-react/ [consulté le 17 juillet 2023].

VIGAN, Noé Joel, 2019. Kotlin:Comment convertir du code Java en Kotlin dans Android Studio. *Android Tutos* [en ligne]. 7 avril 2019. Disponible à l'adresse : https://androidtutos.com/convertir-un-code-java-en-kotlin-sous-android-studio/ [consulté le 17 juillet 2023].

Making a Progressive Web App | Create React App, 2021 [en ligne]. Disponible à l'adresse : https://create-react-app.dev/docs/making-a-progressive-web-app/ [consulté le 17 juillet 2023].

AVELINO, Daniel, 2023. Bachelor_application [logiciel] [en ligne]. 2 juin 2023. [consulté le 17 juillet 2023]. Disponible à l'adresse : https://github.com/danielAvelino26/Bachelor_application [consulté le 17 juillet 2023].

AVELINO, Daniel, [sans date]. React App. [en ligne]. Disponible à l'adresse : https://pwa--elegant-concha-5f5d2c.netlify.app/ [consulté le 17 juillet 2023].

Build Accessible React Apps with Speed ⚡ [logiciel] [en ligne]. 17 juillet 2023. Chakra UI. [consulté le 17 juillet 2023]. Disponible à l'adresse : https://github.com/chakra-ui/chakra-ui [consulté le 17 juillet 2023].

AVELINO, Daniel, [sans date]. E-Commerce. [en ligne]. Disponible à l'adresse : https://cross-plateforme--visionary-cocada-0a5a8b.netlify.app/ [consulté le 17 juillet 2023].

FONTAINE, Benoît, 2023. Comprendre le rendu dans SwiftUI. *Just-Tech-IT* [en ligne]. 20 avril 2023. Disponible à l'adresse : https://medium.com/just-tech-it-now/comprendre-le-rendu-dans-swiftui-8353c3a04624 [consulté le 8 août 2023].

GOIK, Robert, 2022. How does React work – deep dive into the React framework. [en ligne]. 27 janvier 2022. Disponible à l'adresse : https://tsh.io/blog/how-does-react-work/ [consulté le 8 août 2023].

tcm, 2022. Développement mobile : les applis React Native. [en ligne]. 31 mars 2022. Disponible à l'adresse : https://thecodingmachine.com/developpement-mobile-les-applis-react-native/ [consulté le 8 août 2023].

How to make PWAs installable - Progressive web apps | MDN, 2023 [en ligne]. Disponible à l'adresse : https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Tutorials/js13kGames/Installable_PWAs [consulté le 8 août 2023].

Quelles sont les limitations des PWA sur iOS ? - Tech-Tonic, [sans date] [en ligne]. Disponible à l'adresse : https://www.tech-tonic.fr/blog/quelles-sont-les-limitations-des-pwa-sur-ios [consulté le 8 août 2023].

NARAYAN, Praveen, 2020. The progress of PWA - mobile apps. [en ligne]. 7 octobre 2020. Disponible à l'adresse : https://blog.trigent.com/progress-of-pwa-mobile-apps/ [consulté le 8 août 2023].

BRONDIN-BERNARD, Nicolas, 2020. React-Native : Comprendre son fonctionnement en 5 minutes - Nicolas Brondin-Bernard. *Code-Garage* [en ligne]. 19 novembre 2020. Disponible à l'adresse : https://code-garage.fr/blog/react-native-comprendre-son-fonctionnement-en-5-minutes/ [consulté le 8 août 2023].

Service workers, [sans date]web.dev [en ligne]. Disponible à l'adresse : https://web.dev/learn/pwa/service-workers/ [consulté le 8 août 2023].

Material Components widgets, [sans date] [en ligne]. Disponible à l'adresse : https://docs.flutter.dev/ui/widgets/material [consulté le 13 août 2023].

Cupertino (iOS-style) widgets, [sans date] [en ligne]. Disponible à l'adresse : https://docs.flutter.dev/ui/widgets/cupertino [consulté le 13 août 2023].

Tout savoir sur le développement d'applications pour mobile, [sans date]*Spiria* [en ligne]. Disponible à l'adresse : https://www.spiria.com/fr/ressources/tout-savoir-sur-le-developpement-applications-mobile/ [consulté le 30 août 2023].