

Rapport développement Travaux d'Études et de Recherches

070 - Architecture Logicielle pour les Interface Homme Machine sur
wearable computer et dans des environnements connectés

Réalisé par:

Loïc Filippi (M2 - AL)
Enzo Briziarelli (SI5 - AL)
Jessy Dussart (SI5 - IOT)
Sylvain Marsili (SI5 - AL)

Encadré par:

Jean-Yves Tigli
Gerald Rocher

Ingénieurs Science Informatique
à Polytech Université Côte d'Azur
Premier semestre de l'année 2021-2022

Table des matières

Le sujet	3
Scope	4
Les besoins	4
Persona	4
Scénarios	5
L'état de l'art	6
L'existant	6
Les Frameworks	6
Les micro-frontends dans la vie quotidienne	7
Les critiques de l'existant	8
L'apport de notre projet	9
Approche du projet	9
Choix techno	9
Hypothèse de départ : "Le Tout Web"	9
NodeJs/NestJS pour sa modularité	9
React pour le multi devices	10
Première Piste	11
Single-spa	11
L'architecture	12
Coté Back-end	14
Fichier d'entrée JSON	14
Data-Retriever et Readers	14
Aggregator	15
Persistance	16
Un front-end dynamique	16
Communication	17
Exemple	18
Les limites	19
La multi-modalité d'interaction	19
Limite d'intégration de react native	20
Notre travail	20
Organisation	20
Critique	21
Les futur perspectives	21
Conclusion	22
Bibliographie	23
Articles web	23
Annexe	25

Le sujet

La modularité du back-end fait figure de proue dans le monde de l'ingénierie logicielle ces dernières années, notamment avec le passage en architecture [micro-services](#)¹ de la vaste majorité des applications modernes. Néanmoins, le front-end quant à lui est majoritairement omis de cette avancée et reste majoritairement monolithique et spécifique à chaque utilisation. Pourtant leur réutilisabilité est de plus en plus indispensable, tout particulièrement avec l'arrivée et le développement croissant des [wearable computers](#)² qui étendent le champ des appareils disponibles. Chaque appareil étant fondamentalement différent (téléphone, montre connecté, lunettes connectées, ou plus simplement ordinateur), leurs modalités d'affichage ainsi que d'interaction sont fondamentalement différentes.

C'est ici qu'intervient le concept de [micro-frontend](#)³, concept reposant sur des front-end atomiques réutilisables afin de proposer une bonne modularité. La ressemblance avec le principe des micro-services est frappante car il s'agit du même concept appliqué aux fronts. Cependant, il est nécessaire de disposer d'un outil afin de fusionner et d'agencer ses micro-frontends afin de produire un résultat selon l'appareil.

Notre travail consiste donc en la création d'un tel outil, en appuyant sur l'aspect multi-devices et multi-modalités d'interactions nécessaires aux wearables computers.

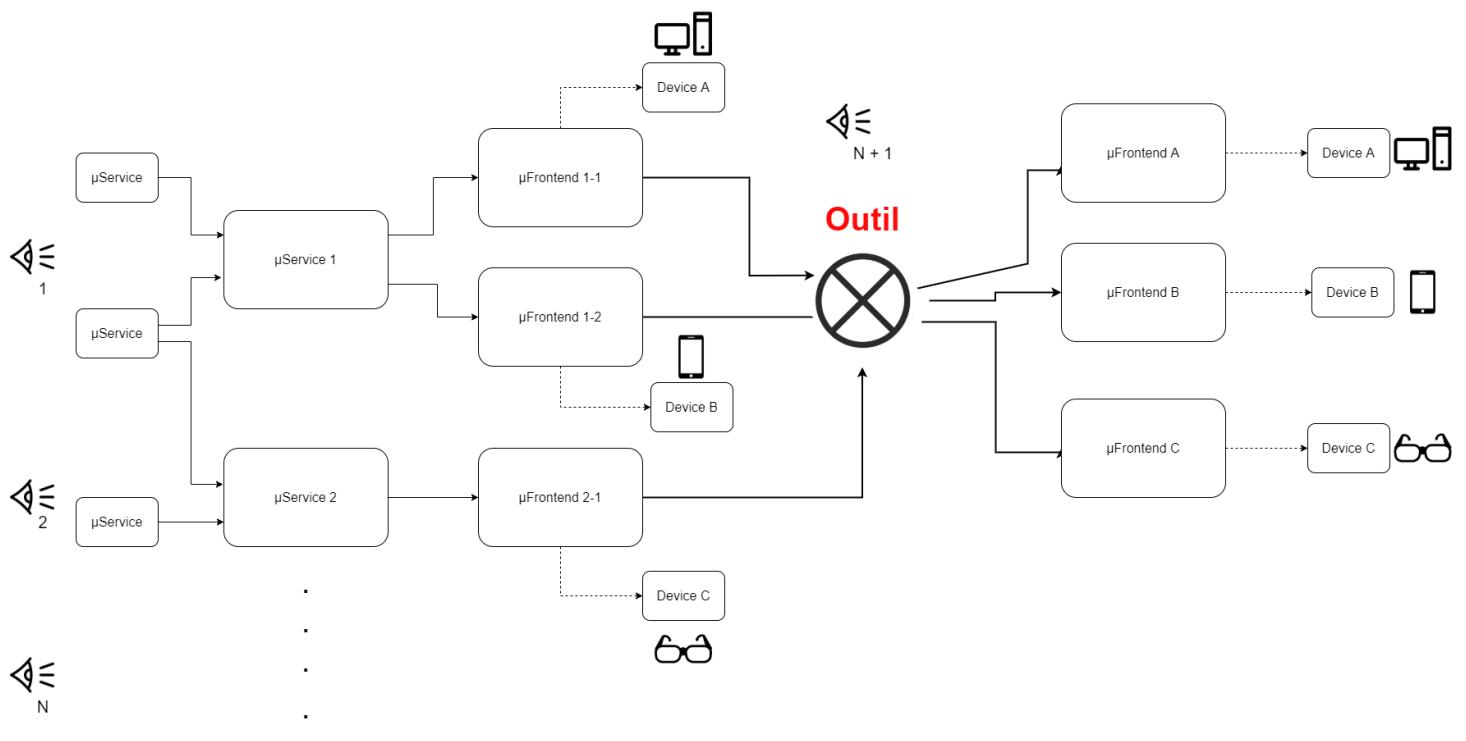


Figure 1 : Définition du fonctionnement et de l'utilité de notre Outil

Ci-dessus un schéma représentant l'utilisation de notre outil par un/plusieurs designers (Le designer de frontend étant notre utilisateur cible).

Un designer va donc définir ses micro-frontends liés à leurs micro-services (définissant la logique métier). Chaque micro-frontend cible un appareil précis et possède une fonctionnalité atomique (affichage d'une carte, lecture d'un texte via text-to-speech, lecture d'une vidéo, etc...).

Un designer (le même ou un autre) va utiliser notre outil pour récupérer tous les micro-frontends créés (par lui ou d'autre) et va générer des frontends *finaux*, c.a.d utilisable et optimisé pour un appareil cible.

Exemple: L'outil génère une page web si l'appareil cible est un ordinateur, il génère une application Android si l'appareil cible est un smartphone Android, etc...

1 - micro-service: Style architectural logiciel qui structure une application comme un ensemble de services faiblement couplés (indépendants) communiquant entre eux via une API.^[1]

2 - wearable computers: Appareil électronique de taille réduite qui se porte directement sur le corps, pouvant être simple (montre digitale, mesure de la fréquence cardiaque) ou complexe (montre connecté, lunettes connectées).^[2]

3 - micro-frontend: Se base sur la même logique fondamentale que les micro-services, mais appliqué aux frontends afin de casser leurs aspect monolithique et d'enfin ajouter de la modularité et de la réutilisabilité.^[1]

Scope

Notre objectif étant un outil de fusion et d'agencement de micro-frontend, notre périmètre d'application s'étend à tout ce dont à besoin directement et indirectement l'outil.

- L'outil doit être en mesure de définir et d'utiliser de multiples logiques de fusion et d'agencement des micro-frontends.
- De plus, il doit pouvoir générer le frontend final selon l'appareil cible, dans un langage choisi au préalable (multi-devices et multi-modale).
- Il devra aussi être en mesure de simplement récupérer les données d'entrées (correspondant aux micro-frontends).
- Enfin, des micro-frontends d'exemples devront donc être créés par nos soins afin de tester l'implémentation complète de l'outil.

Les besoins

Persona

Julien, designer web.

Karen, préparatrice de commandes dans une warehouse est une utilisatrice du produit désigné par Julien.

Dimitri, designer mobile pour polytech.

Scénarios

Scénario 1:

Dimitri veut pouvoir créer une application mobile pour polytech à partir de notre outil. Il veut pouvoir ajouter les éléments suivant dans son application:

- Un message text to speech de bienvenue sur l'app (micro-service)
- La vidéo de présentation de l'école (micro-frontend)
- Les logos fusionner de l'école (micro-frontend)
- La page web de l'école (micro-frontend)

Les différents éléments pour constituer son app sont éparpillés dans différents micro-services et micro-frontend que l'école lui fournit.

Grâce à notre outil, il va pouvoir ajouter via des requêtes ces différents éléments en utilisant les liens de ces différents services.

Scénario 2:

Julien veut créer un tableau de bord web adaptatif sur lequel seront affichés des informations et formulaires provenant de différentes sources. Il lui faut ainsi :

- La vidéo live de la porte d'entrée de la zone sécurisée (vidéo).
- L'image de la personne demandant l'accès fusionné à son nom et son grade (text_over_images).
- Un formulaire de demande d'accès temporaire (html).

Scénario 3:

Julien et **Dimitri** doivent créer un système de guidage dans la réalisation de jeux d'instructions étape par étape. Ils s'attendent à trouver :

- La liste de toutes les étapes sur le web et sur le téléphone.
- A chaque changement d'étape, l'instruction est envoyée et est lue par text-to-speech sur le téléphone.

Scénario 4:

Karen veut brancher sur ses lunettes Vuzix plusieurs éléments :

- checklist fournie par un micro-service
- map & position gps
- itinéraire optimisé dans l'entrepôt vers la position des différents objets

Alors, en utilisant notre produit, **Julien** va permettre d'afficher correctement les différents éléments selon les préférences de Karen.

L'état de l'art

L'existant

Comme vu précédemment, notre sujet à donc pour objectif principal de créer un support universel de micro-frontend pouvant être agencé et peuplé de manière dynamique avec des morceaux de frontend pouvant interagir ensemble et pouvant apporter leur propre logique backend si besoin.

Lors de nos recherches, nous avons pu trouver quelques articles de recherche en libre^[3] accès se rapprochant vraiment du corps de notre sujet ainsi que plusieurs framework web proposant différentes manières d'implémenter des micro-frontends basé sur un développement web dans des pages web avec des tutoriels très basique. Il existe aussi de nombreux articles web variés sur le sujet faisant le parallélisme entre les micro-frontends et les micro-services en retraçant leur principe de fonctionnement et leur utilisation dans de nombreuses pages web de sites de grandes entreprises bien connus et utilisés du grand public au quotidien.

Nous avons quand même pu constater que le développement de micro-frontends se fait toujours sur base web car son exécution et le rafraîchissement dynamique de l'interface est beaucoup plus aisé sur l'interpréteur plutôt que sur un exécutable compilé. Il est toutefois envisageable de faire tourner ces applications web en natif avec des framework comme React Native par exemple.

Les Frameworks

Pour les frameworks proposant des solutions de micro-frontends, toutes les solutions que nous avons pu rencontrer prônent une grande facilité de passage à l'échelle, une sécurité accrue, un développement plus efficace avec une segmentation des différentes fonctionnalités qu'une page web peut proposer ainsi qu'une meilleure résilience des produits développés. Les frameworks les plus connus tels que Bit, Angular ou encore Single-SPA se servent principalement d'une page web HTML/Javascript classique dans laquelle différents services vont venir remplacer un ou plusieurs éléments de la page afin d'afficher dynamiquement le frontend de leur service. Ces services peuvent aussi venir directement insérer du code HTML dans la page à l'aide des balises HTML communes au langage.

Grâce à cette méthode, les responsabilités de chaque élément de la page sont complètement fragmentées et chaque composant est géré indépendamment en tant que projet à part entière. L'agencement final ainsi que les communications entre les différents composants de la page se fait par la page parente de l'application grâce aux différentes méthodes du langage javascript qui est au corps de la majorité des solutions proposées. Cela permet une très grande liberté au niveau des frameworks utilisés par chaque micro-frontend et cela permet donc de créer une page alliant différentes technologies web comme Angular,

React ou encore Vue.js au sein de la même interface sans rencontrer de problème [Fig.2]. Cette approche permet donc à chaque équipe de production de travailler sans contrainte sur la plateforme de son choix et sans avoir à se soucier du travail des autres équipes du projet. Les communications inter-composantes sont faites généralement aux travers d'APIs dédiées mis en place par chaque micro-service ce qui permet d'établir des standards stables sans avoir à se soucier du fonctionnement interne des autres services.

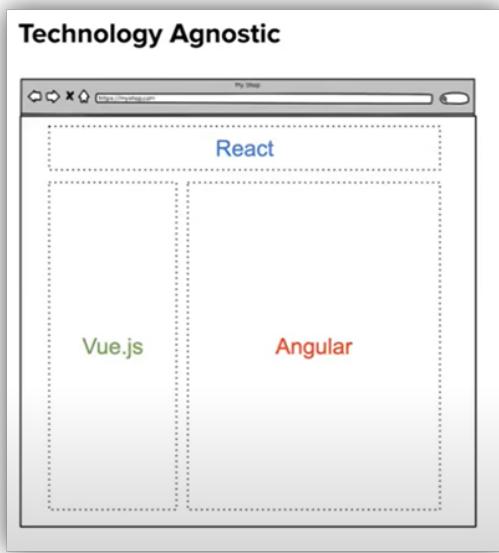


Figure 2 : Association de différents framework

Les micro-frontends dans la vie quotidienne

De nombreuses entreprises ont décidé de se servir de ces micro-frontends pour la création de leurs sites web pour toutes les raisons de facilité de développement cité précédemment. C'est ainsi que les sites comme celui d'Amazon, Ebay ou encore Alibaba par exemple répartissent les différentes fonctionnalités de leurs sites de e-commerce au sein d'équipes spécialisées chargées d'un seul microservices avec un développement plus vertical [Fig.3].

Pour ces grands groupes ayant bien souvent une évolution très rapide et exponentielle et pour qui leurs interfaces web sont à la base de leur activité économique et donc de leurs revenus, il serait pratiquement inconcevable d'opter pour une approche monolithique car cela augmenterait terriblement les temps de développement des mises à jours et l'application serait de moins en moins résiliente aux fils du temps. Les micro-frontends sont donc parfaitement bien adaptés pour le développement de très grosses applications web.

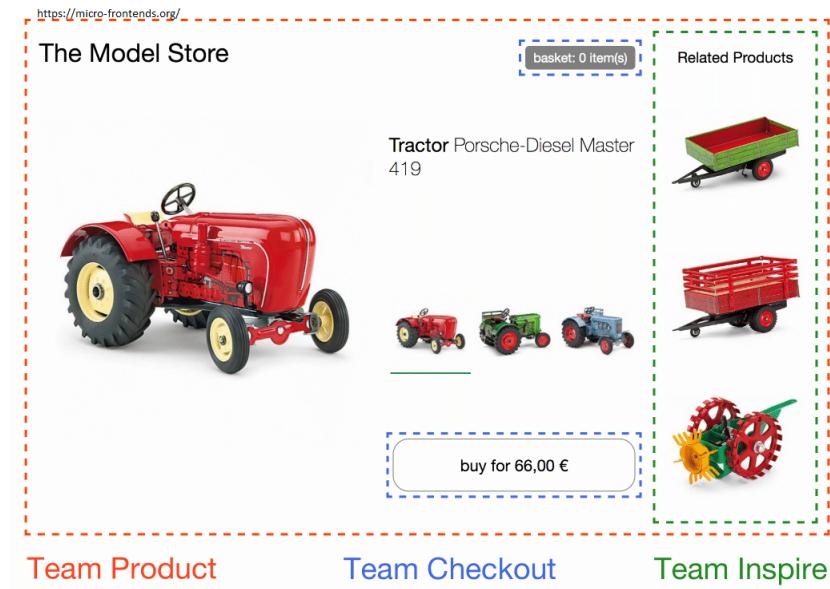


Figure 3 : Interface basée micro-frontend^[4]

Les critiques de l'existant

Le plus gros inconvénient actuel des frameworks de micro-frontends vient ironiquement de leurs grandes versatilité. En effet, les nombreuses interactions entre tous les services et les frameworks différents qui peuvent intervenir au sein d'une même page peuvent causer de nombreux problèmes d'incompatibilité. Certains services peuvent provoquer des conflits entre les différentes versions de librairie utilisées et les API entre services peuvent devenir obsolètes. Malgré un développement fragmenté il faut donc garder un maître d'œuvre pour définir ce qui peut être fait et comment cela doit être fait.

De plus, de par leurs interfaces à caractère labile, les micro-frontends sont présents exclusivement dans des applications basées sur un support web se servant d'un interpréteur plutôt que sur une application native compilée. Les applications PC ou Mobile se servant des micro-frontends comme par exemple Spotify sont en réalité des pages web déguisées en application afin de faciliter leur utilisation et de leur permettre un accès hors connexion. Les applications seront donc plus lente et moins optimisées mais elles s'adapteront à plus de plateforme et d'architecture différentes.

Les applications micro-frontends actuelles sont aussi peu modulaires d'un point de vue externe à l'équipe de développement, une fois que l'agencement global de la page est déterminé, les éléments sont certes placés dynamiquement mais il n'y a plus de changement un fois lancé. Les services ne peuvent être retirés dynamiquement sans impacter sensiblement les autres et ils ne peuvent être remplacés sans relancer complètement l'application. De plus, leur emplacement sur l'interface reste fixe et prédéterminé par l'équipe gérant le produit fini.

L'apport de notre projet

Par rapport aux solutions déjà existantes que nous avons pu évoquer, notre projet sera orienté plus vers la modularité pour les développeurs externes à notre projet et donc aussi potentiellement pour les clients de notre produit. Notre application apportera des méthodes pour placer dynamiquement et à n'importe quel moment des éléments de design et des fonctionnalités codés par d'autres développeurs. Il pourront par exemple ajouter du texte, des images, des vidéos, des éléments en text-to-speech ou encore directement du HTML dans notre interface pour avoir l'application qui leur convient en fonction de leur besoin. Du matériel mobile pourra être branché en cours d'utilisation (comme un GPS ou une boussole) et grâce à notre API mise en place, l'interface pourra s'adapter à ce nouveau périphérique et elle l'affichera suivant les préférences de l'utilisateur. Nous fournissons aussi des outils simples pour permettre de faire différents agencements basiques.

A termes, notre but est de permettre à notre application d'accepter plusieurs langages et protocoles pour avoir une plus grande versatilité et pour permettre aux futurs développeurs d'avoir un plus grand panel de technologies à utiliser. Nous souhaitons aussi permettre aux utilisateurs d'utiliser notre application et leurs services sur des supports variés comme sur un ordinateur, des lunettes connectées ou encore un casque audio sans interface visuelle pour s'adapter à un maximum d'environnement et de situations différentes.

Approche du projet

Choix techno

Hypothèse de départ : "Le Tout Web"

De par sa nature et ses divers appareils concernés, le périmètre d'application de notre sujet est très large et peut englober plusieurs technologies variées. C'est pourquoi nous nous sommes restreint aux langages web uniquement, afin de pouvoir délimiter précisément notre champ effectif sur la période dédié au projet et ainsi ne pas dépasser les délais, tout en conservant les deux caractéristiques principales de notre outil : multi-devices et multi-modalités d'interaction. En effet, la combinaison HTML et Javascript nous offrent un fondation solide sur laquelle s'appuyer, l'important étant maintenant de choisir les bonnes bibliothèques.

NodeJs/NestJS pour sa modularité

Coté Backend, les technologies choisies sont NodeJS avec NestJS: NodeJS^[5] est une plateforme logicielle en JavaScript afin de créer des applications JS, communément des serveurs web, avec une spécialisation dans la gestion de montée en charge et de l'événementiel.

NestJS^[6] est un [framework](#)¹ récent pour applications serveurs NodeJS.

L'avantage principal du framework Nest, outre son aspect open-source, est sa modularité, intégré dans sa conception, permettant notamment l'ajout de bibliothèques ou paquets Javascript récents (Exemple, l'ajout du support natif du paquet KafkaJS permettant d'intégrer de la communication événementielle via [Apache Kafka](#)²).

Un autre avantage non négligeable de Nest étant sa popularité grandissante. En effet, dans le cas des technologies web, une grande popularité implique une plus grande communauté active de développeurs, facilitant les mises à jour fréquentes et l'entraide via des espaces d'échanges dédiés tels que StackOverflow.

1 - Framework: désigne un ensemble de composants logiciels créant une fondation, un socle, à une application. Un framework se distingue d'une simple bibliothèque car beaucoup plus complexe.

2 - Apache kafka: Projet open-source de communications événementielles de messages via un système de bus et de transaction. Kafka est pensé pour être "temps-réel": résilient et résistant à la charge.^[7]

React pour le multi devices

Coté Frontend, les technologies choisis sont React et React Native:

React^[8] est une bibliothèque JavaScript dédiée aux interfaces utilisateurs (frontends).

React Native^[9] est une bibliothèque JavaScript, directement liée à React, permettant de générer des interfaces utilisateurs cross-plateforme (Smartphone, Desktop, etc...) en utilisant du code natif React.

L'avantage principale de React est sa capacité à gérer de multiples appareils sur une base de code quasi-identique, avec React pour le rendu classique sur ordinateur et React Native pour le rendu sur Android et iOS.

Un autre facteur déterminant étant l'omniprésence de React dans les papiers, recherches, et autres tutoriels dédiés aux micro-frontends de manière générale. Cela assure donc de disposer du plus de matériel possible et d'une plus grande communauté active.

Première Piste

Single-spa

Lors de notre analyse de l'existant, le framework *single-spa* [\[10\]](#) était un de ceux qui semblait le plus proche de notre scope. Les critères d'agencement de micro-frontends sur une page global, et l'utilisation des micro-frontends (conçus avec différents frameworks frontend) en faisait partie.

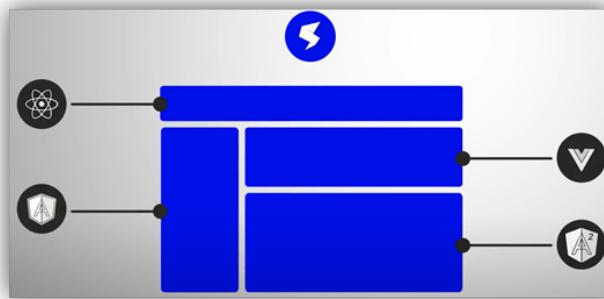


Figure 4: Composition possible de front avec Single-spa

Afin d'avoir un aperçu et une idée de ce que notre projet pouvait apporter, nous avons décidé de faire nos premiers pas dans ce framework. Nous avons mis en place deux micro-services simples avec deux micro-frontends associés. Ensuite, l'app global prend en compte ces deux micro-frontends pour n'en former qu'un.

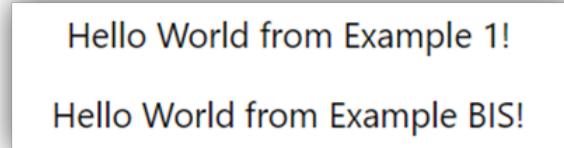


Figure 5: Exemple basique créé avec Single-spa

Single-spa a tout de suite montré des limites quant à notre scope. En effet ce dernier ne permet que de gérer des micro-frontends écrits dans des frameworks front tel que React, Angular, Vue ... Il ne permet pas de prendre en compte juste d'autres formes de données front (un HTML, une image, un texte ...). Il n'est pas possible de faire de vraies fusions entre les différents micro-frontends (exemple basique: 'text1' + 'text2' = 'text1 text2'). De plus, un changement de code dans les micro-frontends doit être effectué afin de permettre à ces derniers d'être pris en compte par le front global.

Ces inconvénients nous ont permis de mieux définir notre scope et la portée de notre projet.

L'architecture

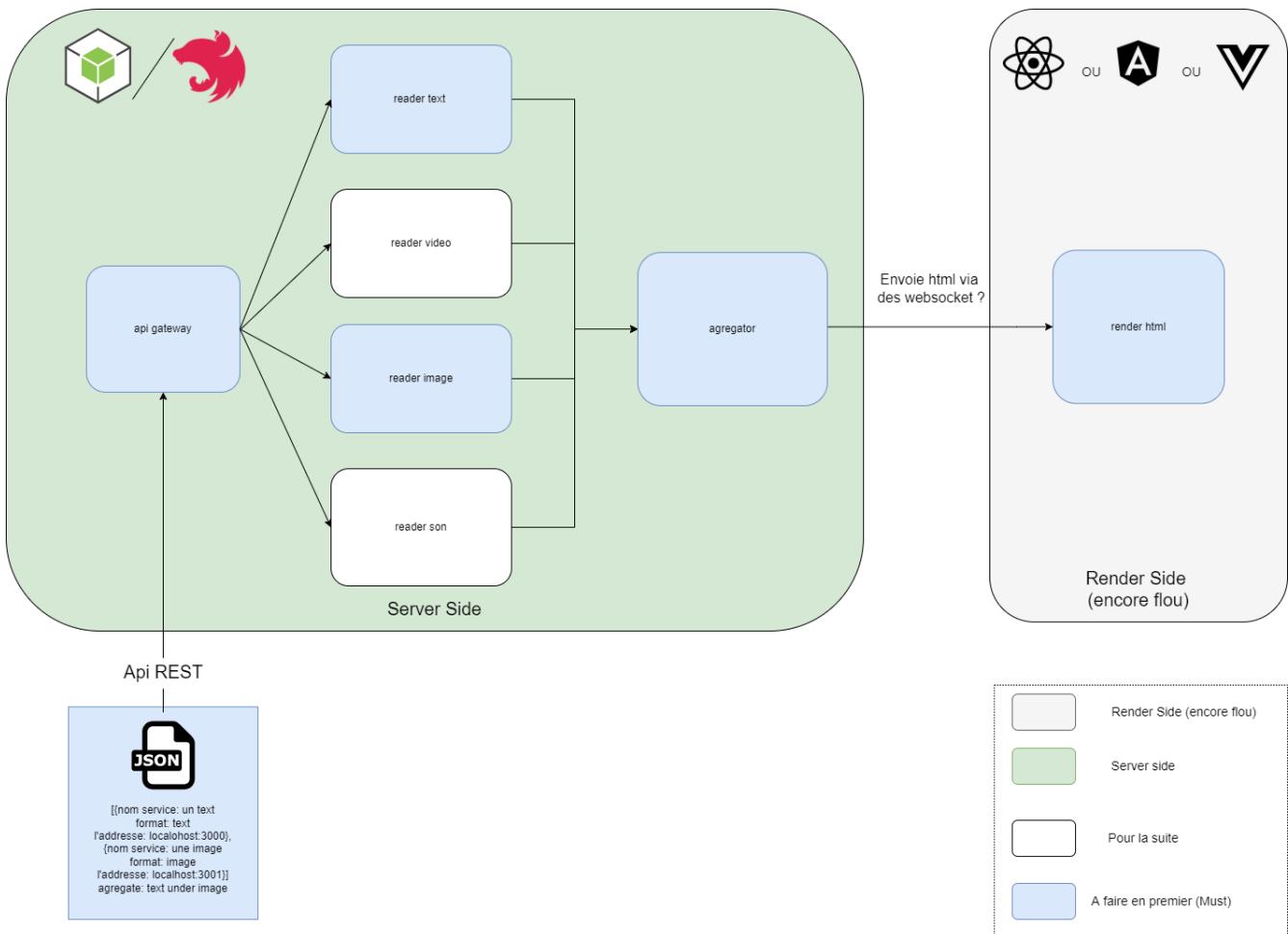


Figure 6: Premier jet de notre architecture

Dès notre première architecture, il apparaissait clair de devoir découper notre outil en 3 parties distinctes:

- La récupération de la donnée: Ici avec un fichier descriptif au format JSON, fourni via une API REST, indiquant la stratégie de fusion à adopter ainsi que les ressources (micro-frontends) à récupérer, en spécifiant leur adresses, types, etc...
- Le coeur de la logique métier, là où les ressources sont traités: Application de la logique d'agencement via le service *aggregator* après lecture de la ressource selon son type (service *reader*-... correspondant)
- La génération du frontend (Render): Avec un langage Web cible nous permettant d'appliquer correctement la logique d'agencement.

N.B : Lors du choix des technos, nous avons opté pour NodeJS/NestJS niveau backend, mais la techno du rendu niveau frontend était encore incertaine, mis à part que l'hypothèse de départ "Tout Web" devait être respecté, nous avions donc le choix entre React, Angular, et Vue.

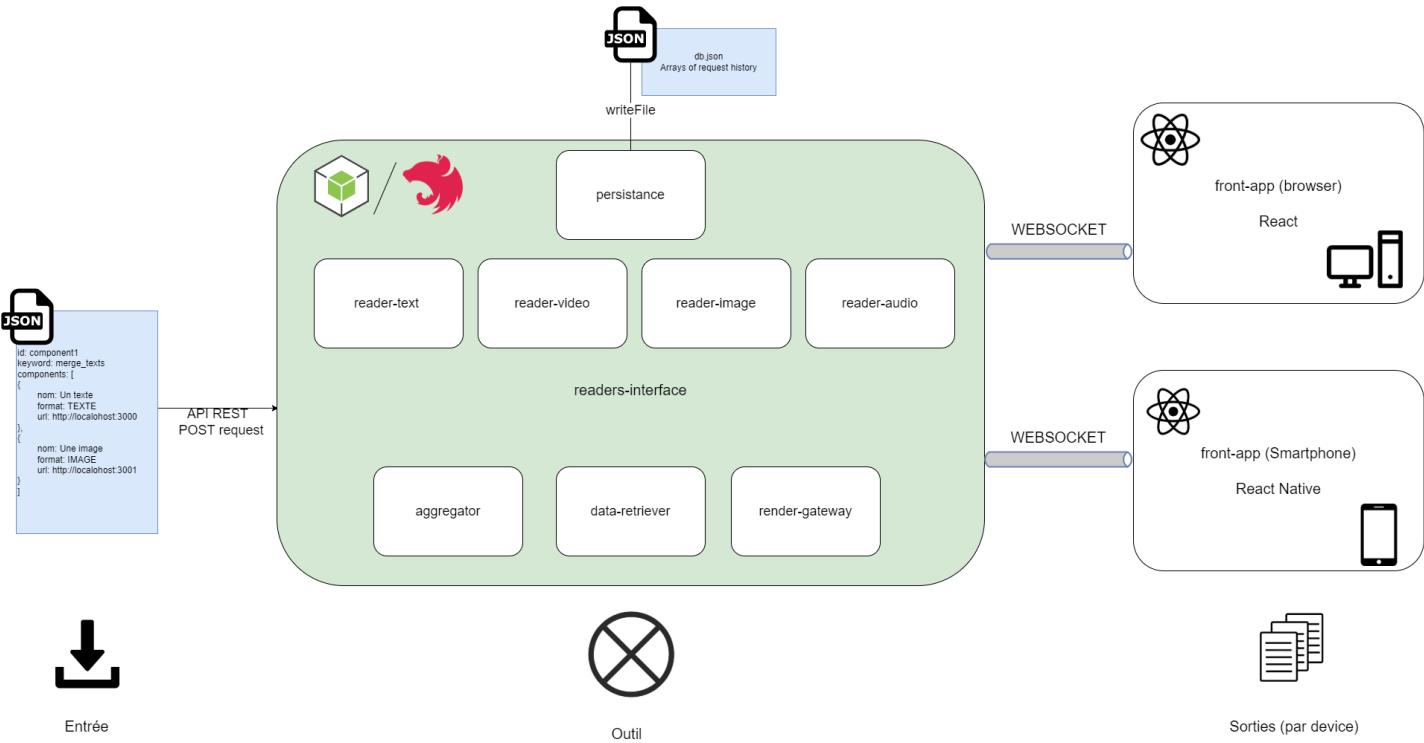


Figure 7: Architecture implémentée

Ci-dessus, l'architecture implémentée aujourd'hui, version finale de notre outil. Nous pouvons remarquer quelques changements, tels que:

- Côté backend, les composants de la logique métier sont au sein d'un monolithe. En effet, le métier ne va pas subir de charge trop importante, il a donc été décidé d'imbriquer les composants nécessaires (*readers, aggregator, data-retriever*) au sein d'un monolithe restreint.
- Une fois la génération terminée, notre outil va transmettre le code cible via une websocket à notre application côté frontend pour le rendu.
- Côté frontend, nous avons tranché sur les technologies de rendu: React (Web) et React Native (smartphone Android et iOS). Chaque appareil a donc son moteur de rendu propre afin de respecter l'aspect multi-devices et multi-modalités.

Coté Back-end

Fichier d'entrée JSON

```
{
  "Id": "Image-6",
  "keyword": "RANDOM",
  "components": [
    {"name": "Logo Polytech",
     "url": "https://th.bing.com/th/id/OIP.pa6Ble6lxyWGw3zskhU0QAHaEt?pid=ImgDet&rs=1",
     "fileFormat": "IMAGE"}
  ]
}
```

Figure 8: Structure JSON en entrée

Le fichier d'entrée au format JSON est composé de la manière suivante :

- **keyword** spécifie la stratégie d'agencement à utiliser
- **components** liste les micro-frontends à intégrer

Chaque micro-frontend doit être décrit de la manière suivante:

- **id** est l'identifiant unique d'un composant
- **name** correspond au nom d'usage du micro-frontend
- **url** est l'adresse ou la donnée peut être récupérée. Ici, sous la forme d'un container Docker (Un container par micro-frontend donc, sur le modèles des micro-services)
- **fileFormat** désigne le format de la donnée (Texte, Image, Audio, Vidéo, HTML, TTS). Ici, TTS (Text-To-Speech) désigne du texte qui est destiné à être lu par le synthétiseur vocal.

Data-Retriever et Readers

Le premier composant va être chargé de récupérer les données brutes des micro-frontends via une simple requête HTTP au service concerné.

Le second est responsable d'encadrer le micro-frontend récupéré dans des tags html correspondant à son format. Il existe un composant Reader par format de données disponibles (Texte, Audio, Vidéo, etc...). Les formats disponibles sont explicités dans un fichier dédié.

```
export class ReaderVideoService {

  constructor() {}

  createTags(data: string) : string{
    return `<video id="videoPlayer" width="650" controls muted="muted" autoplay>
              <source src="${data}" type="video/mp4" />
            </video>`
  }
}
```

Figure 9: Exemple, lecture d'une vidéo via le composant **ReaderVideoService**

La donnée récupérée est encadrée de tags html afin de créer l'élément de lecture vidéo.

Aggregator

Le composant Aggregator est chargé d'appliquer la logique de fusion et d'agencement. Il utilise les données des micro-frontends précédemment encadrés par les tags html.

La stratégie appliquée par ce composant dépend de la stratégie désirée par le designer, renseigné via le mot clé **keyword** du fichier d'entrée.

Lorsque aucune stratégie n'est demandé (keyword à vide ou non reconnu), la stratégie par défaut s'applique : "order-matters". Cette stratégie définit que l'agencement des composants est flexible selon leurs place prise sur l'écran, et qu'il seront positionnés à la suite par ordre de placement dans le fichier d'entrée. Cette stratégie étant volumineuse, je vous invite à vous référer au code source.

Voici plutôt un exemple d'application d'une stratégie simple, la fusion de textes (mot clé "merge-texts") :

```
async aggregate_merge_texts(components: ComponentDTO[]): Promise<string>{
    let resultTags: string = '<div>';
    let resultText: string = '';

    for(const elem of components){
        resultText += await this.dataRetriever.getDataFromService(elem);
        resultText += ' '; //Put a space in between texts
    }

    resultTags += this.readerTextService.createTags(resultText);
    resultTags += '</div>';
    return resultTags;
}
```

Figure 10: Exemple, merge d'un text via le composant **AggregatorService**

La méthode va récupérer les données (plusieurs textes) via le *DataRetriever*, concaténer les n-textes, puis appliquer le texte résultant au *ReaderText* afin de produire la sortie encadrée attendu pour le rendu HTML, comme expliqué dans la partie suivante.

Persistance

Nous nous sommes rapidement confrontés à un problème avec notre frontend. A chaque refresh de la page, tous les composants créés via le backend disparaissaient. En effet, les composants ne sont pas persistés, et donc supprimés à chaque refresh. Deux solutions se présentaient alors: tout d'abord sauvegarder les données côté front avec le cache de la page web (*sessionStorage* et *localStorage*), ou sauvegarder côté back. Mais notre choix s'est très vite orienté vers une gestion interne dans le backend.

Cela nous permet plusieurs possibilités:

- garder la structure de manière de composant créé de manière permanente pouvant permettre au développeur de retrouver sa configuration dans l'outil
- le développeur peut également directement changer la structure via le fichier de persistance ou la récupérer

Afin de mettre en place cette persistance, nous avons utilisé un fichier JSON (*db.json*) (voir Annexe 3) afin de garder les éléments que le développeur a créé. Dans le futur, ce système pourra être modifié par un système plus adapté si besoin

Les données sauvegardées sont représentées par les messages JSON envoyés à notre backend. À chaque refresh du frontend, un événement est envoyé afin de récupérer ces messages et relancer le processus de création de balise et d'affichage de composant.

Un front-end dynamique

La composition de micro-frontend nous a imposé une certaine dynamicité du frontend global. Ce dernier doit réagir aux ajouts des différents micro-frontend que le développeur veut mettre en place dans notre app. L'ajout de composant dynamique est un atout majeur que notre frontend React met en place. En effet, la bibliothèque React nous propose différentes fonctions permettant de créer des composants à la volée, notamment la fonction *createElement()*. C'est cette dernière que nous utilisons.

```
const createComponent = (data) => {
  var props = {htmlContent: data.html, key: data.id}
  return React.createElement(DynamicComponent, props);
}
```

Figure 11: Fonction pour ajouter un composant

Le frontend global permet de créer dynamiquement des composants basés sur le langage HTML que notre backend nous fournit, grâce à l'utilisation de JSX, l'extension syntaxique de JavaScript que React permet d'utiliser.

Nous proposons pour le moment deux types de frontend global:

- Le premier utilisant React, et pouvant être interprété directement via les navigateurs web.
- Le second utilisant React Native, est capable de créer directement des applications Android et IOS.

L'HTML nous permet d'être agnostique du type d'application utilisée. En effet, même si ce balisage est orienté web, grâce à l'utilisation de React et React-Native, il peut très facilement s'intégrer sur différentes applications, que ce soit mobile ou web, et ça sans changer les balises que notre backend nous envoie. Il suffit juste de changer le balisage JSX directement dans les deux types de front.

```
const DynamicComponent = ({htmlContent: initialHtmlContent}) =>{
  const htmlContent = useState(initialHtmlContent);
  return (
    <View style={{ height: 200 }}>
      <WebView originWhitelist={['*']} source={{ html: htmlContent[0] }}/>
    </View>
  );
}

function DynamicComponent({htmlContent: initialHtmlContent}) {
  const htmlContent = useState(initialHtmlContent);
  return (
    <div dangerouslySetInnerHTML={{ __html: htmlContent[0] }} />
  );
}
```

Figure 12: Notre composant dynamique, à gauche en React et à droite en React Native

Comme cité plus haut, différentes stratégies de fusions ont été mises en place entre nos composants. Outre ces fusions, un autre besoin est apparu, en effet il fallait prendre en compte le placement des composants entre eux. Pour permettre cela, nous avons décidé de gérer ce positionnement de manière dynamique, grâce à la librairie *react-grid-layout*. Cette librairie permet d'organiser le layout global d'une application React sous forme de grille. Les composants sont alors déplaçables, et leur taille peut être réglée directement depuis le front.. Ainsi, cela permet une grande modularité du front.

Communication

Le but de notre projet est de permettre aux développeurs de pouvoir rajouter des micro-frontend à tout moment, et de manière dynamique, dans le front global. Pour ça, le développeur envoie une requête via l'api REST de notre backend avec pour message un JSON. Ce JSON permettra de structurer le ou les micro-frontend voulu selon les choix du développeur (différent type de contenu, de fusion...).

Nous nous sommes alors, assez vite aperçus d'une première contrainte lors de la création de notre outil. En effet, une fois que notre backend a créé les balises HTML, il fallait alors envoyer à notre frontend global ces informations.

En général, nous avons l'habitude que ce soit le frontend qui demande les informations au backend selon les actions de l'utilisateur. Or, ici c'est l'inverse. Ainsi nous avons dû créer une communication de type *WebSocket* entre le backend et le frontend afin que ces derniers puissent communiquer de manière bidirectionnelle. Ce *WebSocket* vas supporter plusieurs type d'évènements:

- render: envoyé par le back au front pour ajouter un composant suite à une requête du développeur
- refresh: envoyé du front au back pour récupérer les composants déjà ajouter par le développeur, cela lors d'un refresh de la page front

```
async render(message: HtmlObjectDto) {
  this.server.emit('render', message);
}
```

Figure 13: Événement render du Websocket côté serveur back

Exemple

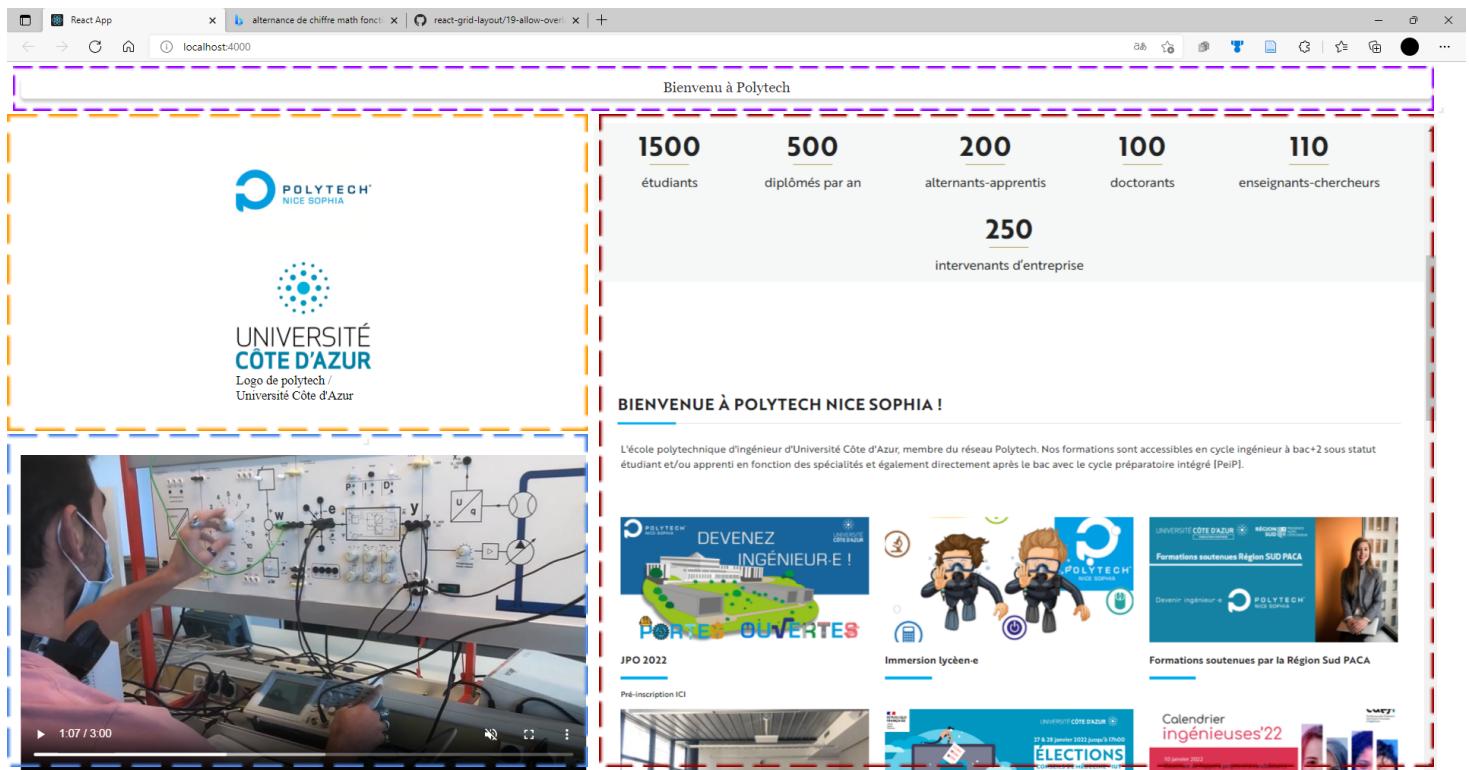


Figure 14 : Exemple de front end global créé depuis notre outil

- **En violet**, un composant TTS (Text to Speech), citant "Bienvenu à Polytech" lors d'un clic dessus
- **En orange**, un composant avec la stratégie de fusion "aggregate_text_over_images", avec deux IMAGE et leur légendes TEXT .
- **En bleu**, un composant VIDEO, avec une vidéo fournie par l'un de nos micros services (Présentation de polytech).
- **En rouge**, une composant HTML, la page web d'accueil de Polytech Nice sous forme d'IFrame.

Nos composants sont soit créés depuis des liens des micro-services que nous avons instanciés, soit depuis des liens web.

Ce résultat est obtenu grâce à un fichier JSON (db.json) (voir Annexe 3).

Cette exemple est également responsive (voir Annexe 4)

Les limites

La multi-modalité d'interaction

Un des points crucial sur lequel le projet porte est axé sur la multimodalité des interactions, de fait, il est nécessaire de pouvoir s'assurer que le client puisse afficher les informations sur des supports variés tels qu'un ordinateur, un téléphone, une montre connectée, une paire de lunettes connectée, voire même juste un support audio sans écran...

Ces exigences engendrent de nombreux problèmes, certains bien connus et étudiés aujourd'hui comme la mise en page adaptative selon la taille de l'écran du support. Ainsi que d'autres un peu plus insoupçonnés tel que la cohérence de la donnée sur tous les supports, et ce tout en assurant l'adaptation de l'information au support utilisé.

Afin de pallier les problèmes de mise en page adaptative, nous avons utilisé un système d'affichage à partir d'une [grille](#)¹ dite "responsive". Ainsi il nous est possible de gérer la taille des éléments et leur disposition selon plusieurs "breakpoints" qui correspondent à des différents paliers de tailles d'écran.

Cette adaptation apportant un premier élément de réponse n'est cependant pas suffisante pour avoir la prétention de convenir à toutes les différentes tailles d'écran de manière optimale.

Le système de persistance des données présenté plus haut a ainsi permis de régler un problème qui concernait la cohérence entre les affichages sur les différents supports. En effet, dès lors qu'il a été mis en place on pouvait être sûr que chacun des appareils accédant à la page récupérait tous les mêmes micro-frontends ajoutés au préalable.

Mais alors, deux nouvelles problématiques sont apparues, la première, pouvant facilement être résolue par l'utilisation de clés d'identification provient évidemment du fait que tous les périphériques récupèrent les infos, alors qu'on aurait peut-être voulu limiter l'affichage de certaines informations à un seul navigateur web. Ainsi, on pourrait à l'aide de ces clés faire correspondre une persistance à un ou plusieurs utilisateurs / utilisations.

La seconde problématique apparue est, elle, plus difficile à appréhender et est toujours une limite dans notre projet. Dans le cadre de notre sujet, l'adaptation de l'information au support utilisé aurait pu être une fonctionnalité apportant de la valeur à notre produit.

Prenons comme exemple, un scénario où il aurait été intéressant d'afficher une liste entière d'opérations à effectuer sur l'ordinateur, mais n'afficher que les informations relatives à l'étape actuelle sur les lunettes, nous sommes dans l'incapacité de réaliser ça avec notre produit.

1 - Système de mise en page en grille pour React [React-Grid-Layout](#)^[11]

Limite d'intégration de react native

Nous avons décidé d'utiliser le framework d'applications mobiles [React Native](#)¹ qui nous permet d'utiliser React avec les fonctionnalités natives des plateformes tels qu'Android ou IOS, tout en étant le plus proche possible de notre hypothèse "tout web".

En effet, il nous suffisait d'ajouter ce nouveau frontend en plus de celui géré par React, pour que l'on puisse être capables de re-récupérer les informations envoyées par le même backend et de les afficher.

Cependant, au fur et à mesure du développement de nouvelles fonctionnalités, la demande constante de mise à jour de ces deux frontends devient pénible, et pas forcément justifiée. De fait, notre application ne prend pas en compte des interactions qui seraient propres aux téléphones, et notre frontend React étant adaptatif fait que le projet se consulte facilement sur un navigateur de téléphone. Ces deux arguments couplés au caractère chronophage de la mise à jour perpétuelle des deux front-ends en parallèle a engendré une limite d'intégration de React Native au sein de notre projet.

1 - Framework React Native <https://reactnative.dev/>.

Notre travail

Organisation

Durant les deux premières semaines du projet, nous nous sommes principalement concentrés sur la définition du scope, la compréhension du sujet, et l'état de la l'art. Ensuite nous nous sommes attardés sur la mise en place de solutions existantes (Single-spa), afin de cerner encore plus précisément le sujet et de trouver ce que nous pouvions apporter en plus de l'existant. Enfin nous avons avancé avec un Walking Skeleton sur le projet, tout en prenant en compte les différents retour de nos encadrant.

Afin de s'organiser au mieux, nous avons réparti les différentes features entre nous, avec pour délai de production environ 1 semaine avant de refaire de nouveau brainstorming sur la suite. Des issues sont ouvertes sur notre repo Git, représentant les features ajoutées, chacune sont associées à un membre de l'équipe. Nous suivons l'avancement de chacun sur le kanban projet Git, et également via différents salons textuels Discord. Nous avons aussi utilisé des diagrammes (draw.io) afin de suivre l'évolution de notre architecture.

Critique

Nous avons eu beaucoup de mal à cadrer le sujet, malgré les indications fournies par nos encadrants, nous avons mis du temps, et avons dû faire des tests avant de partir dans une direction qui semblait convenir. Ainsi notre outil a mis quelques semaines à prendre forme notamment sur des features comme les fusions, que nous aurions voulu encore plus développer.

Nous avons aussi remarqué au fil du temps, que certaines idées de conception ont été prises

afin d'avoir un outil fonctionnel rapidement, mais que ces dernières n'étaient pas forcément les plus adaptées.

Par exemple, l'utilisation du JSON pour la création de composant n'est pas le plus adapté. En effet, l'utilisation d'un DSL, plus proche du domaine et permettant une plus grande flexibilité, semblait plus appropriée. Malheureusement, le manque de connaissance sur ce sujet nous a freiné quant à sa mise en place, même si à présent nous avons une idée de comment nous aurions pu l'implémenter.

Certaines fonctionnalités basiques manquent aussi pour le moment, par exemple nous n'avons pas gérer la suppression de composant (même si cela est possible en modifiant directement le db.json), nous concentrant principalement sur les différents types de format qu'il est possible de traiter. Il n'est actuellement pas possible de gérer directement le placement des composants sur l'écran depuis notre json, il est uniquement possible de le faire depuis le front.

Nous nous sommes aussi moins concentrés sur la partie native (la feature layout grid n'existe pas dans l'app), car notre front est responsive.

Malgré, ces manques, l'ajout de fonctionnalités comme la possibilité de faire du Text to Speech, de permettre une certaine modularité du front grâce à react-grid-layout ou encore pouvoir gérer différents types d'entrées, apportent à notre solution un plus vis-à-vis de l'existant.

Nous avons aussi mis en place plusieurs micro-services et micro-frontends afin de démontrer l'utilisation et la gestion de notre outil.

Les futur perspectives

D'après les différents points cité plus haut, nos perspective semble s'accorder avec les points suivants:

- ajout d'un DSL
- suppressions/modifications des composants (sans le faire depuis le fichier db)
- gestion du layout grid depuis le back
- ajout de nouvelles features d'interaction pour le wearable computer (Speech to Text, gestion du gyroscope ...)
- ajout de nouvelles stratégies de fusions (superposition intelligente des composants, fusion par catégorie ...)
- améliorer notre app mobile et notre back afin de prendre en compte les features uniquement possible sur mobile (gyroscope, notification ...)

Conclusion

Notre objectif était, la création d'un outil permettant d'agréger des micro-frontend et données afin de créer un frontend global, modulable et adaptatif.

La réalisation de l'état de l'art nous a permis de constater la faisabilité de notre projet et sa place dans l'existant. Nous avons pu voir que les micro-frontends sont très prometteurs pour les entreprises d'aujourd'hui dont la taille ne cesse d'augmenter et pour qui la simplicité de développement est un facteur clé. Même si aujourd'hui plusieurs travaux et framework existent sur les micro-frontends, c'est un domaine qui reste quand même assez jeune et beaucoup de travail reste à faire. Nous avons donc pu assez facilement identifier les éléments sur lesquels nous pouvions nous baser et nous avons aussi pu identifier les points à améliorer et les fonctionnalités qu'il serait intéressant d'ajouter.

L'outil devait permettre de créer un frontend final sur un appareil cible, et d'effectuer de la fusion, de l'agencement de micro-frontend de manière intelligente selon les choix des utilisateurs.

Actuellement notre outil permet de faire certaines fusion basique, d'agencer les composants de manière dynamique, de prendre en compte différentes entrées (micro-frontend, image, vidéo ...). Sans générer de frontend, il est lié directement à un frontend React ou React Native. Notre outil possède toutefois des limites, comme des fusions peu développées ou encore une configurabilité restreinte des composants. Malgré ces limites, des solutions ont déjà été envisagées pour les débloquer.

Au final, ce TER fut assez complexe à prendre en main, mais au fur et à mesure de nos analyses de l'existant et du développement de notre outil, nous avons cerné vers où nous voulions nous diriger.

Pour terminer, nous tenons à remercier nos encadrants Jean-Yves Tigli et Gérald Rocher pour leur aide et leur suivi tout au long de ce projet.

Bibliographie

- [1] <https://martinfowler.com/articles/micro-frontends.html> (microservices)
- [2] <https://internetofthingsagenda.techtarget.com/definition/wearable-computer>(Wearable computers)
- [3] M. Mena, A. Corral, L. Iribarne, et J. Criado, « A Progressive Web Application Based on Microservices Combining Geospatial Data and the Internet of Things », IEEE Access, vol. 7, p. 104577-104590, 2019, doi: 10.1109/ACCESS.2019.2932196.
- [4] <https://micro-frontends.org/> (Micro-frontends)
- [5] <https://nodejs.org/en/about/> (NodeJS)
- [6] <https://nestjs.com/> (NestJS)
- [7] <https://kafka.apache.org/> (Apache Kafla)
- [8] <https://reactjs.org/> (React)
- [9] <https://reactnative.dev/> (React Native)
- [10] <https://single-spa.js.org/> (Single-SPA)
- [11] <https://github.com/react-grid-layout/react-grid-layout> (React-Grid-Layout)

Articles web

J. Saring, « 11 Micro Frontends Frameworks You Should Know », Medium, nov. 18, 2020. <https://itnext.io/11-micro-frontends-frameworks-you-should-know-b66913b9cd20> (consulté le nov. 24, 2021).

« Micro Frontends », martinfowler.com.
<https://martinfowler.com/articles/micro-frontends.html> (consulté le nov. 24, 2021).

Donnez un Microfrontend à vos Microservices:
<https://medium.com/@ylerjen/donnez-un-microfrontend-%C3%A0-vos-microservices-f6c422b2bb46>

Microfrontends: An approach to building Scalable Web Apps:<https://medium.com/@areai51/microfrontends-an-approach-to-building-scalable-web-apps-e8678e2acdd6>

Micro Frontend Architecture:

<https://levelup.gitconnected.com/micro-frontend-architecture-794442e9b325>(1)

Pavlenko, A.; Askarbekuly, N.; Megha, S.; Mazzara, M. Micro-Frontends: Application of Microservices to Web Front-Ends. 18(1)

Yang, C.; Liu, C.; Su, Z. Research and Application of Micro Frontends. IOP Conf. Ser.: Mater. Sci. Eng. 2019, 490, 062082. <https://doi.org/10.1088/1757-899X/490/6/062082>.(1)

Création d'une interface utilisateur composite basée sur des microservices -

<https://docs.microsoft.com/fr-fr/dotnet/architecture/microservices/architect-microservice-container-applications/microservice-based-composite-ui-shape-layout>

React-Grid-Layout is a grid layout system for React.

<https://github.com/react-grid-layout/react-grid-layout>

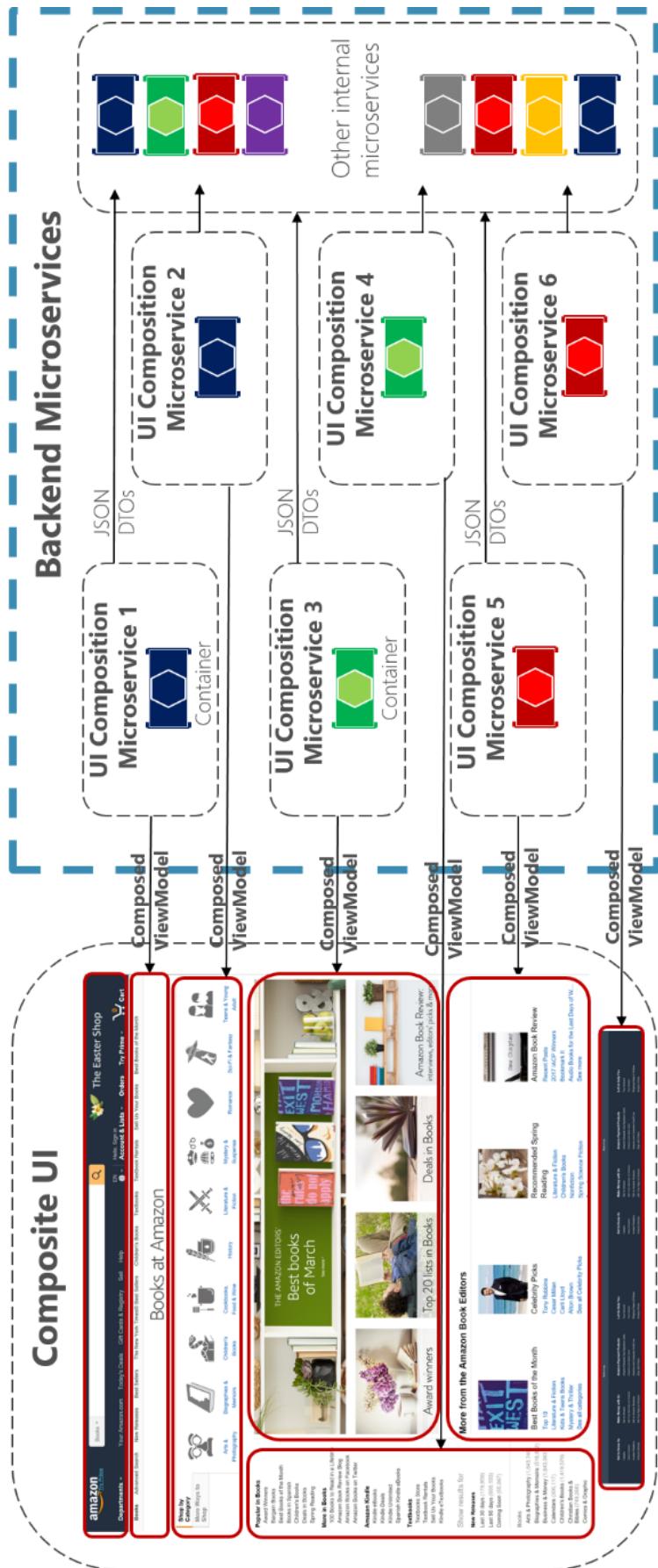
Annexe



Annexe 1- Utilisation des lunettes avec deux affichages : un sur le téléphone et un sur les lunettes.

Composite UI generated by microservices

<https://docs.microsoft.com/fr-fr/dotnet/architecture/microservices/architect-microservice-container-applications/microservice-based-composite-ui-shape-layout>



Annexe 2- Interface utilisateur composite générée par des microservices (Amazon)

```
[  
  {  
    "id": "page",  
    "keyword": "RANDOM",  
    "components": [  
      {  
        "name": "Page wiki polytech",  
        "url": "https://polytech.univ-cotedazur.fr/",  
        "fileFormat": "HTML"  
      }  
    ]  
  },  
  {  
    "id": "images",  
    "keyword": "text_over_images",  
    "components": [  
      {  
        "name": "Légende",  
        "url": "http://localhost:3001/",  
        "fileFormat": "TEXT"  
      },  
      {  
        "name": "logo polytech",  
        "url": "https://th.bing.com/th/id/R.970db754bf82e038461012770717af7?rik=U210SeZj",  
        "fileFormat": "IMAGE"  
      },  
      {  
        "name": "logo uca",  
        "url": "https://geoazur.oca.eu/images/GEOAZUR/Logos/UCA_logo.png",  
        "fileFormat": "IMAGE"  
      }  
    ]  
  },  
  {  
    "id": "video",  
    "keyword": "RANDOM",  
    "components": [  
      {  
        "name": "Video présentation",  
        "url": "http://localhost:3002/",  
        "fileFormat": "VIDEO"  
      }  
    ]  
  },  
  {  
    "id": "tts",  
    "keyword": "RANDOM",  
    "components": [  
      {  
        "name": "Bienvenue",  
        "url": "http://localhost:3001/2",  
        "fileFormat": "TTS"  
      }  
    ]  
  }  
]
```

Annexe 3 - Exemple de Fichier db.json pour la persistance



Annexe 4 - Affichage sur mobile de l'application web responsive créé via notre outil