Mémoire professionnel

YNOV Ingésup M2 – année 2017-2018

Mastère expert informatique et systèmes d'information

Migration architecturale et microservices

Dans le cadre d'une application SaaS, quel est l'impact d'une migration architecturale en microservices sur les équipes de développement ?

Applications pour experts-comptables connectés :
Produisez, partagez, fidélisez.

Étudiant : Étienne TROUILLET

Entreprise : IBIZA Software

Tuteur entreprise : Alain BOUILLÉ

Directeur de mémoire : Pascal RONGET

Remerciements

Je tiens à remercier tout particulièrement les personnes suivantes et à leur témoigner toute ma reconnaissance pour l'expérience enrichissante et pleine d'intérêt qu'elles m'ont fait vivre durant cette année d'alternance au sein de l'entreprise Ibiza Software, et l'aide qu'elles m'ont apportée lors de mon travail sur ce mémoire :

Monsieur **Silviu LUPU** (manager) : pour sa présence en continu sur le projet et la liberté qu'il m'a donnée pour la réalisation du mémoire.

Monsieur **Alain BOUILLE** (tuteur) : pour m'avoir encadré depuis mon arrivée dans l'entreprise Ibiza, et pour son écoute.

Monsieur **Pascal RONGET** (directeur de mémoire) : pour ses précieux conseils tout au long de l'année et lors des séances de méthodologie.

Monsieur **Johann WOJTOWICZ** (directeur d'Ibiza) : pour la confiance qu'il m'a accordée dans la réalisation de cette année d'alternance dans l'entreprise et de ce travail.

Et enfin tous ceux qui m'ont encouragé, de près comme de loin : famille, amis et collègues.

Table des matières

Remerciements....1

Introduction....4

Chapitre 1 : Détail de l'environnement....6

1. Présentation de l'entreprise....6

1.1. Histoire de l'entreprise....6

1.2. Activités et spécificités....6

1.3. Diagnostic du marché de l'entreprise....7

1.4. Ma mission dans l'entreprise....9

2. Détails de la stack logicielle existante....10

2.1. Couches de l'application....10

2.2. Détails plus techniques....11

3. Développement actuel et limites....12

4. Le projet d'évolution....14

4.1. Limites techniques....14

4.2. Objectifs du projet....15

4.3. Enjeux de la problématique....16

5. État de l'art....17

5.1. Les microservices....17

5.2. Du monolithe aux microservices....20

6. Mise en application dans Ibiza....26

Chapitre 2 : Hypothèses et développement méthodologique....28

1. Première hypothèse : l'environnement du web et des outils front-ends....28

1.1. Définition de l'hypothèse....28

1.2. Méthodologie....29

1.3. Analyse....29

1.4. Mise en application chez Ibiza....36

1.5. Conclusion de l'hypothèse....38

2. Deuxième hypothèse : l'architecture microservices face aux limites de l'application Ibiza....39

2.1. Définition de l'hypothèse....39

2.2. Confrontation via l'enquête....40

2.3. Impact pour le logiciel d'Ibiza....44

2.4. Futur du logiciel et de l’architecture....49
2.5. Conclusion de l’hypothèse....51
Chapitre 3 : Préconisations et perspectives....52
1. Énonciation des préconisations....52
2. Actions faites par l’entreprise....57
3. Apports et limites de la réflexion....58
3.1. Entreprise....58
3.2. Étudiant....58
Conclusion....60
Glossaire....61
Bibliographie....62
Table des illustrations....63

Introduction

Le Web est un domaine en constante évolution et son impact sur les entreprises est permanent et en croissance. Le passage au numérique est toujours en cours dans certains milieux où des applications auparavant cloisonnées sur des ordinateurs locaux deviennent alors des applications web, car elles sont décentralisées par le biais d'internet. Les logiciels de gestion d'entreprise, les intranets, et finalement les ERP eux-mêmes, sont alors sujets à l'évolution du web et à ses technologies toujours en permanente réinvention.

Cela va faire bientôt 4 ans que je travaille chez Ibiza Software, entreprise développant son logiciel SaaS[1] de gestion comptable et paie. Ces années passées chez eux m'ont permis de découvrir le monde de l'application web. C'est un sujet qui me passionne aussi, il était donc naturel pour moi d'orienter mon thème de mémoire sur les applications web et plus précisément sur un sujet très actuel : la migration d'une application.

Une migration est l'action par laquelle une entreprise va décider d'orienter le développement d'un logiciel pour l'évoluer partiellement ou complètement. Cela impactera les trois prochaines années pour la réalisation par les équipes de développement, et au moins les cinq suivantes pour l'utilisation par les clients et la gestion des itérations permettant d'être toujours en phase avec les avancées du milieu.

Cette idée d'évolution vient directement de mon travail actuel dans l'entreprise. Nous avons en effet commencé un projet dans ce sens, après avoir pris conscience que la stack, c'est-à-dire l'ensemble des composants du logiciel, approchait de leur limite. Plusieurs éléments, dont certaines limites de l'application actuelle, que je détaillerai dans ce mémoire, ont permis des remises en question et le début d'une réflexion sur un chantier d'évolution logiciel. J'ai donc décidé de me pencher sur le sujet, et notamment de réfléchir sur les impacts que peut avoir cette évolution pour l'entreprise, en suivant un type d'architecture dont la fiabilité est connue.

Dans le cadre d'une application SaaS, quel est l'impact d'une migration architecturale en microservices sur les équipes de développement ?

C'est la problématique à laquelle je vais répondre tout au long de mon mémoire. Dans un premier chapitre, je ferais le tour de mon environnement de travail en entreprise, du statut actuel de l'application d'Ibiza et de l'état de l'art couvrant le type d'architecture visé et les impacts d'une migration. Puis je me concentrerais dans un deuxième chapitre sur le développement de plusieurs hypothèses en lien avec la problématique, pour lier les principes remontés de la littérature avec le cas

1 Voir glossaire.

4.3. Enjeux de la problématique

La problématique est donc directement liée au projet d'évolution d'Ibiza. C'est une migration partielle ou complète, car l'équipe sait qu'il sera obligatoire de réécrire la partie front-end de l'application, qui doit être complètement restructurée, d'où l'intérêt d'utiliser un framework existant qui permettrait d'avoir un langage plus homogène entre les équipes. La démonstration de cette problématique peut aider l'entreprise à entreprendre le projet dans le bon sens, en découvrant dans un premier temps ce qui est proposé par la littérature lors d'un projet de ce type, et aussi comment bien appréhender l'architecture microsrervices, celle visée au départ par l'équipe du projet. Les différents retours d'expériences peuvent aussi être importants et permettent de s'aider des projets réalisés pour comprendre le déroulement mené par d'autres et leurs retours sur ce travail. Tous ces éléments vont être décrits dans l'état de l'art que j'ai réalisé et que je vous expose dans la partie suivante. Ensuite, la définition d'hypothèses dans le deuxième chapitre me permet de confronter cet état et les besoins actuels d'Ibiza, pour aider les équipes à se concentrer sur certains points de réflexion au départ, et à juger de l'intérêt des éléments remontés par la littérature pour le projet.

5. État de l'art

Dans un premier temps, cet état de l'art va définir l'architecture microservices et son application dans le domaine du web. Il est important de comprendre son fonctionnement, qui change complètement de celui mis en place actuellement dans Ibiza. La différence se fera ressentir en confrontant cette architecture avec celle utilisée par Ibiza, dite monolithique. Enfin, un aperçu de certaines expériences de migration entre ces architectures permettra de positionner les premiers impacts que peut avoir ce projet pour les équipes et le logiciel.

5.1. Les microservices

5.1.1. Définition des microservices

L'objectif de cette partie est de faire un tour des différentes définitions et structuration de l'architecture microservices. La définition la plus connue est la suivante : « Microservices are small, autonomous services that work together. » (Les microservices sont des petits services autonomes qui travaillent ensemble) (Newman, 2015, p. 16). Elle permet, sans rentrer dans la technique, de mettre en avant les deux principes fondateurs des microservices, qui sont :

- L'existence de services : c'est-à-dire de module dont la finalité est différente. Un service peut travailler avec un autre, mais chacun aura ses propres objectifs.
- Et des services qui sont petits (« micro »), de façon générale, un microservices à une seule fonction, mais il la fait bien.

Dit d'une façon différente, James Lewis indique que l'architecture en microservices est une approche pour développer une application comme une suite de petits services, chacun s'exécutant dans son propre environnement (Lewis, et al., 2014). Le microservices est donc un type de service présentant plusieurs caractéristiques clés. Quand on parle de l'architecture microservices, on parle d'un système basé sur une multitude de microservices, en incluant toutes les caractéristiques de cohabitation, de mise en relation des services, de leurs déploiements, monitoring, etc.

On insiste sur un point clé de ces services : l'autonomie, qui comprend plusieurs critères :

- Isolation d'exécution : c'est-à-dire que le service doit avoir son propre environnement d'exécution à lui seul.
- Isolation de déploiement : ce service pourra être mis en production, lancé, de façon indépendante, sans requérir un changement chez le client (celui qui utilise le service, peut être un autre service, un logiciel à part entière, un utilisateur …). Le danger est que le service soit trop couplé à la structure du logiciel, « ce qui diminue l'autonomie, et requière une

coordination additionnelle avec le consommateur quand des changements sont mis en place » [Traduction libre] [CITATION New15 \l 1036].

Pour vérifier ce critère, il faut donc se demander en permanence l'impact sur le déploiement et l'environnement du service, si des changements sont effectués et si des erreurs surviennent. C'est une caractéristique que l'on appelle la résilience : la capacité du microservices à résister par lui-même aux actions pouvant le mettre à défaut.

À ce point, une définition technique rentre en place, car pour qu'un service soit le plus autonome possible, on utilise généralement une communication dite API (Application Programming Interface) qui permet de définir l'interaction d'un service avec l'extérieur. Même si le microservices est autonome dans son fonctionnement, il lui faut des indications pour lancer un traitement, et pouvoir renvoyer des données si besoin. C'est cet interfaçage qui permet la communication avec l'environnement de l'application.

D'autres critères entrent aussi en compte dans cette architecture :

- L'hétérogénéité technologique. C'est un principe clé pour les équipes, car cela signifie que chaque service peut avoir son propre langage de développement (grâce à l'autonomie). Ce critère permettra notamment d'influencer sur l'organisation des équipes, et surtout d'être à jour sur les nouvelles technologies au fur et à mesure de leurs évolutions.
- La scalabilité (ou passage à l'échelle), définie par la possibilité de résister à de fortes demandes en performance en dédoublant un service, pour permettre de balancer les demandes. C'est un des points phare de cette technologie, qui impact énormément sur l'infrastructure mise en place pour le fonctionnement des services, et qui a besoin de monitoring pour faciliter l'automatisme de déploiement en fonction de la demande cliente.

« Les microservices sont, d'une certaine façon, des bonnes pratiques à mettre en place pour réaliser une architecture orientée service » (C. Pautasso et al., 2017). Nous avons vu l'autonomie, l'hétégorénéité, la scalabilité, il y a aussi l'orchestration, le déploiement et d'autres.

5.1.2. État des lieux

Il est important aussi de voir quels sont aujourd'hui les microservices mis en place, et comment les équipes ont travaillé dessus au fur et à mesure. Il en existe beaucoup, je me suis penché sur des cas de chercheurs et d'entreprises ayant travaillé sur ce type de système.

Plusieurs chercheurs se sont intéressés à ce sujet, pour permettre de centraliser les différentes applications de cette architecture, et les points qui peuvent différer. La littérature m'a montré que

pour beaucoup, les microservices peuvent être mis en place de différentes façons, et aujourd'hui c'est une architecture qui n'est pas fixée réellement, notamment concernant les méthodes d'application des principes. Chaque entreprise et équipe utilisent des façons différentes de production de ces services, tout en essayant de respecter les différents principes cités plus hauts. Pour accompagner ce raisonnement, deux chercheurs ont regroupé différentes manières de travailler les microservices, afin de voir de façon globale les différences notables pour le développement. « Pour aider les développeurs à choisir le pattern le plus approprié, notre objectif est d'identifier et de caractériser les différentes architectures microservices utilisés » (D. Taibi et al., March 2018).

Par exemple, les chercheurs ont défini trois grands principes de gestion des microservices, des patterns clés, dont les approches peuvent être distinctes :

- L'orchestration et la coordination des services : certains microservices vont utiliser des API gateway, c'est-à-dire les points d'entrées définis pour chaque microservices, une feuille de route à suivre lorsque l'on veut effectuer des requêtes sur un ou plusieurs services. D'autres vont utiliser le Service Discovery Patterns, principe consistant à définir dynamiquement les interactions avec les services, en utilisant notamment un service qui enregistre les différentes instances déployées. Un pattern plus hybride existe aussi, combinant les deux précédents via un message-bus, une sorte de tunnel utilisé pour les communications.
- La stratégie de déploiement des services, séparés en deux pratiques principales : les services multiples par hôte, ainsi chaque service à son propre environnement (via VM ou conteneurs), mais sur la même machine ; ou chaque service a son propre hôte de déploiement (une seule machine).
- Les techniques de stockage de données : soit chaque service à sa propre base de données privée, l'approche la plus simple et la plus recommandée pour de petites tailles de données ; soit les données sont stockées sur un cluster ; soit on utilise une base de données partagée entre les services.

Ainsi, l'architecture se définit par plusieurs principes fondateurs, permettant de respecter des règles propres aux microservices. Mais lors du développement plus technique de l'architecture, plusieurs possibilités s'offrent à l'équipe pour mettre en place cette architecture. « Les microservices sont applicables dans de multiples domaines, mais seulement si vous avez les besoins qui les rendent nécessaires. » (C. Pautasso et al., 2017). Il faut ensuite utiliser celles qui correspondent le mieux au logiciel et à l'environnement de l'équipe.

5.2. Du monolithe aux microservices

Pour la suite de l'étude, je me suis penché sur la « migration », impliquant donc un passage d'une ancienne architecture à une nouvelle. Pour cela, je me suis orienté au départ sur l'architecture dite monolithe, encore très utilisée. Ce choix a été facile pour plusieurs raisons :

- C'est une architecture opposée aux microservices, et souvent mise en comparaison.
- Elle est très utilisée dans la littérature en exemple de migration (d'une architecture monolithique à une architecture microservices).
- C'est l'architecture actuelle du logiciel d'Ibiza, qui sera donc le point de départ pour la migration vers la nouvelle architecture.

5.2.1. Comparaison avec l'architecture monolithique

C'est une architecture encore très présente, étant la façon traditionnelle de développer une application en un seul bloc logiciel, relié à une base de données, puis déployé en une seule fois et pouvant être copié pour répondre à des besoins de scalabilité. Plus techniquement, « un serveur monolithique consiste à exécuter toute la logique logicielle dans le même processus, et vous permet d'utiliser les fonctionnalités premières de votre langage de programmation, pour diviser votre application en classes, fonctions et espaces de noms » [CITATION Lew14 \l 1036]. Cette architecture a directement été reprise des applications lourdes, démarrée sur des postes utilisateurs. C'est une manière assez naturelle de construire une application au départ : toute la logique tourne sur un seul processus qui exécute des requêtes à travers d'internet.

Les différences avec l'architecture microservices vont donc être multiples et opposées : les fonctionnalités n'ont aucune autonomie, car sont intégrées au même bloc logiciel. Tout étant lié, les mêmes technologies de fond sont utilisés, et leur mise à jour impactera fortement l'environnement. Le déploiement de l'application est faite de façon globale : ainsi si l'on veut installer des patchs permettant de régler certains bugs, toute l'application sera à redéployée. Dans le temps, plus l'application monolithique grandit, plus il est difficile de garder une structure propre et clair du système. Le découpage par service, au contraire, permet à la structure modulaire de s'étendre indépendamment, et améliore nettement la qualité des fonctionnalités car elles ne sont plus couplés avec d'autres.

Le schéma suivant permet de se représenter les différences avec une architecture microservices :

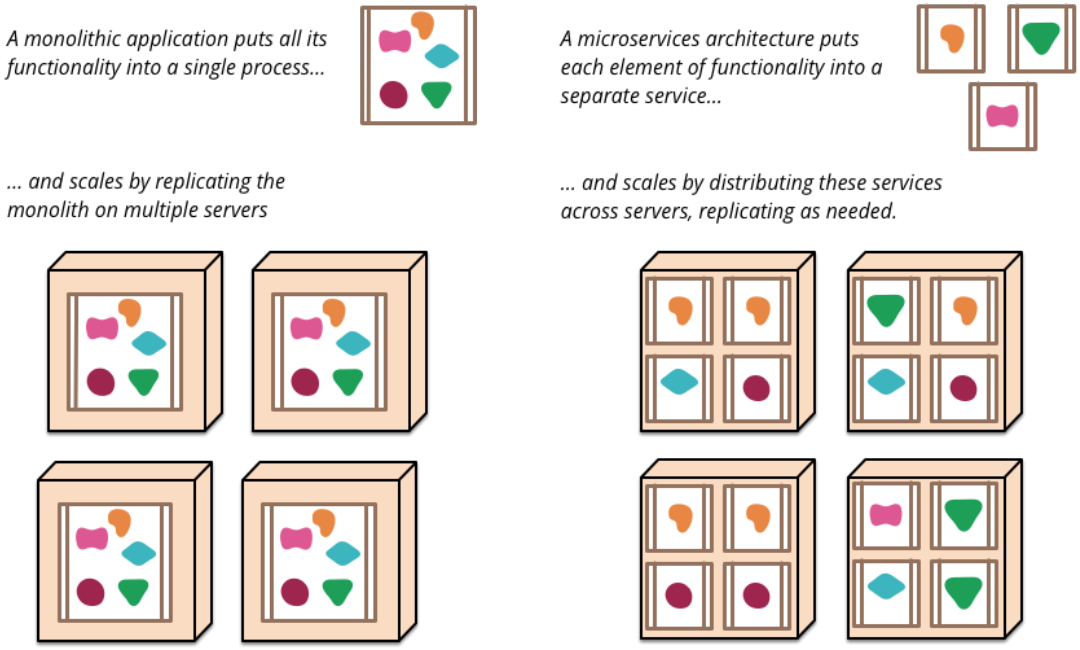

Figure 2 : Monolithes et microservices

(Source: Martin Fowler)

Ce schéma explique les principales différences de structuration en quelques phrases :

- Une application monolithique met toutes ses fonctionnalités dans un seul processus …
- … et effectue des mises à l'échelle (scale) en répliquant le monolithe sur de multiples serveurs.
- Une architecture microservice met chacune des fonctionnalités dans un service séparé…
- … et effectue la mise à l'échelle en distribuant ces services à travers des serveurs, répliqués selon les besoins.

La limite du monolithe réside dans les nouveaux besoins des applications actuels. « Même dans le cas d'une application moyenne, l'architecture monolithique devient petit à petit difficile à maitriser, spécialement pour les applications demandant de fortes scalabilité » [CITATION Gui15 \l 1036]. La demande cliente est un des points majeurs qui peut limiter aujourd'hui ce type d'architecture. Il faut par contre bien faire la différence entre une demande de quelques milliers d'utilisateurs, considérés comme une basse demande, et à l'opposé les demandes de quelques millions d'utilisateurs (c'est le cas pour Facebook, Netflix, Google) où les enjeux de rapidités, d'accessibilités et de réponses sont bien supérieurs.

Il faut donc peser le pour et le contre. « Sans aucun doute, l'application monolithique est la plus simple à déployer. Et c'est le plus gros avantage de cette solution » [CITATION Gui15 \l 1036]. Le plus gros problème de cette architecture est lorsque l'application devient trop grande, trop complexe, et à ce moment-là, la maintenance devient lourde. « La productivité est ralentie par la grandeur du code

source de l'application. Très souvent cela aboutira à une baisse de qualité du code. » [CITATION Dim14 \l 1036]. En prenant en compte l'évolution des technologies de développement, l'architecture monolithique rend complexe la mise à jour du code source, surtout si le projet est imposant. Ce sont plusieurs points qui sont confirmés aujourd'hui par l'expérience que nous faisons de cette architecture avec le logiciel d'Ibiza.

La migration à partir d'une architecture monolithique est très courante, et c'est un travail de réflexion large, pas seulement technique. « Ce que la plupart des logiciels [qui ont migré] avaient en commun est l'incapacité de produire des projets à terme avec leur architecture monolithe existante, d'où la décision de migrer vers des architectures orientées service (SOA) et particulièrement des microservices » [CITATION Ces \l 1036]. C'est donc un sujet aussi large, au niveau de l'entreprise et de la gestion de projet.

Le dernier point important concerne la base de données. Dans une l'architecture monolithique, une seule base permet l'enregistrement et la lecture des données de l'application. Dans le cas des microservices, les bases pourront être décentralisées à travers les services, chaque fonctionnalité pouvant accéder à une base différente, ou partagée :

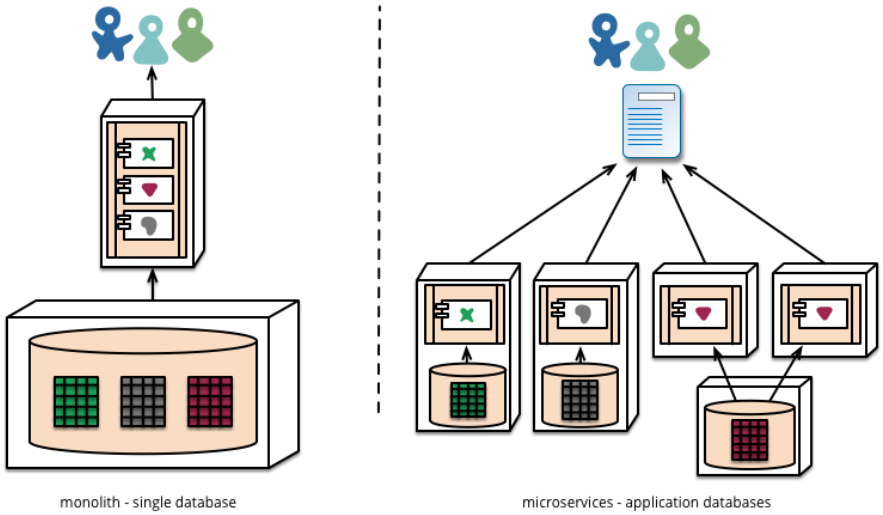

Figure 3 : Base simple et base décentralisée
(Source: Martin Fowler)

Cette différence de mise en place des bases peut avoir des avantages techniques, et permet une meilleure structuration des données dans l'application.

5.2.2. Études de migrations

Pour cette partie de la revue de littérature, je me suis aussi penché sur des articles de chercheurs ayant étudiés et regroupés des connaissances sur les migrations d'architecture monolithe vers des

microservices. Cela permet de rassembler les points clés lors de ces migrations qui pourront aider les équipes actuelles dans la mise en place de ce processus.

Pour cette partie, je me suis aidé notamment de deux cas concrets d'évolutions :

- Présente lors d'une conférence internationale de l'IEEE, l'exemple de cette entreprise française, MGDIS, montrant comment ils ont défini la granularité de leur découpage en microservices, ainsi que leurs méthodes de déploiement utilisés [CITATION Gou17 \l 1036].
- Le cas de la migration d'un module appelé FX Core de la banque Danske Bank (Danemark), mission très complexe qui a amené une réflexion à long terme de l'entreprise. L'article en ressort les aspects positifs et négatifs, tout en discutant des points clés de la migration. (Dragoni & Al., 2017).

Plusieurs points ont été évoqués pour justifier le besoin de lancer un projet de migration, notamment :

- Assurer une évolution à long terme des logiciels.
- Faciliter l'interopérabilité (l'interfaçage) avec d'autres produits, services et logiciels.
- Évoluer vers un logiciel plus ergonomique, notamment concernant les interfaces utilisateurs front-ends, et étendre les fonctionnalités.

Ces retours d'expériences permettent de valider certains aspects de migrations avec des cas concrets : « Depuis quelques années, il est devenu évident qu'il est très compliqué d'identifier le travail des équipes de développement lorsqu'elles sont toutes rattachées à la même application monolithique ». [CITATION Ris15 \l 1036]. Nous avons eu l'avantage à Ibiza de découper les équipes, dès le départ, selon un développement métier, ce qui a permis de mieux combler les lacunes d'identification de développement. Petit à petit, pourtant, certains développements plus techniques ont commencé à voir le jour dans les équipes.

Il s'ensuit alors un travail d'analyse et de questionnement pour valider une nouvelle architecture et ensuite approuver son intégration dans l'environnement des équipes. La plus grande question lorsque l'on passe d'une architecture monolithique à microservices est le choix de découpage de l'application : comme vu précédemment, une application monolithe comprend tout le fonctionnel tandis que pour une architecture microservices il est bien séparé et indépendant.

Intervient alors le terme technique de « granularité » : c'est de décider de la finesse de l'application, quelles sont les fonctionnalités qui vont être scindées, tout en veillant à respecter les principes des microservices (autonomie, scalabilité, hétérogénéité, etc). « Le découplage des fonctionnalités existantes du monolithe peut prendre beaucoup de temps, c'est pour cela que ce travail de

factorisation en microservices doit être fait petit à petit » [CITATION Kal18 \l 1036]. La migration doit donc se faire au fur et à mesure en réfléchissant à l'optimisation du découpage pour le travail futur des équipes.

Deuxième point important relevé par la littérature est la méthode de déploiement, après découpage, qui doit alors être compatible avec la nouvelle architecture. En effet, les technologies de déploiements sont totalement différentes entre le monolithe, où tout est en un seul bloc, et les microservices qui, à l'inverse, scindent les fonctionnalités en blocs déployés indépendamment : il faut donc un écosystème permettant d'appeler chaque service, d'enregistrer les lancements, de rediriger les demandes clientes selon les besoins finaux (car un service s'occupe d'une demande en particulier). « If you don't approach deployment right, it's one of those areas where the complexity can make your life a misery. » (Si votre approche du déploiement n'est pas correcte, c'est un domaine où la complexité peut faire de votre vie un vrai calvaire) (Newman, 2015, p. 188). Autrement dit, c'est un domaine complexe à mettre en place, mais qui est un point central pour éviter que l'environnement soit trop difficile à maintenir.

Le troisième point est lié au précédent : il faut définir les règles de communication entre les services pour qu'ils puissent travailler ensemble. C'est l'orchestration des services, comme relevé précédemment dans la définition des microservices. Il faut définir une méthode de connexion prenant en compte tout l'écosystème, mais permettant une prise en main rapide et une maintenance facilitée. Sam Newman insiste sur la prise en compte des erreurs lors du développement de cette brique : 'Failure is everywhere' (les échecs sont partout). Il assure que l'échec d'une partie du système est inéluctable même si nous faisons du mieux possible pour les éviter. Il donne l'exemple des disques durs, sur lesquels reposent tous les systèmes dans la couche infrastructurelle, qui sont aujourd'hui plus fiables que jamais, mais peuvent se détériorer. Plus l'on utilise de disques durs, plus haute est la probabilité d'erreur pour une unité individuelle. Ainsi, dans le développement de la complexité du système, il faut toujours penser à la possibilité d'erreurs, d'échecs système, pour les remonter correctement et éviter des échecs en cascade.

5.2.3. Impacts des migrations

Les équipes des deux cas d'évolution observés ont remonté plusieurs points intéressants, permettant de mesurer l'impact qu'a eu pour eux cette évolution :

- L'augmentation de la réutilisation tout au long du développement : en effet, la séparation en fonctionnalités clés permet de les utiliser dans des contextes différents pour un même objectif (c'est par exemple le cas d'un service d'authentification). Cela est influé par une meilleure structuration du logiciel.
- La facilité de remplacement des services, notamment des mises à jour et assurance qualité des fonctionnalités déployés.
- La hausse de performance en environnement de production, avec dans certains cas une rapidité supérieure à 150% comparé aux performances du logiciel initial.
- La baisse de support haut niveau, dû à une meilleure qualité logicielle.

On peut regrouper ces impacts en deux points : le premier regroupe les impacts directs sur le logiciel (performance, qualité, structuration), le deuxième concerne les changements que cela a influé sur l'entreprise de façon plus générale : une baisse du support (donc un gain de ressources). Le plus gros point remonté dans tous les cas est la capacité à maitriser la scalabilité du système, bien supérieur aux autres architectures, notamment à celle monolithique.

Plusieurs grandes entreprises de l'IT, comme Amazon, Netflix et Uber, ont aussi montré comment cette architecture avait impacté leur marché. Dans leur cas, cela leur a permis d'augmenter la masse de données à gérer au même moment, au vu de leur trafic important, grâce à un Back-End complètement décentralisé et gérant plusieurs applications simultanément. Leur objectif est aussi de se préparer pour la suite, avec en 2020 plus de 4 milliards de personnes connectées à internet, plus de 25 millions d'applications disponibles et 5200 Gigabits de données par personne sur la terre[9]. Ces acteurs majeurs du web valident que l'architecture microservices est aussi un bon moyen de rester à niveau pour la demande au jour le jour et celle à venir, qui ne cessera de croître.

Un autre impact important peut être celui des coûts : pour Wallmart, après avoir migré en microservices, ils ont économisé plus de 20% en coût d'infrastructure, en gagnant plus de 40% de puissance de calcul, réduisant le nombre de machines nécessaires pour répondre à la demande du moment. Pour MGID, un des cas d'évolution observé, leur gain s'est fait ressentir côté support, en y diminuant le personnel dédié.

6. Mise en application dans Ibiza

Cette architecture à plusieurs avantages en l'appliquant au logiciel d'Ibiza :

9 Rapport EMC Digital Universe, 2014.

- Le découpage en services permet, pour les équipes, de se centrer sur leur développement métier en ayant un environnement de travail dédié. La gestion du code source sera plus spécifique, car soumise au métier, sans impacter les autres équipes.
- Cela permet aussi, en prenant en compte l'environnement général, de mieux séparer les différentes couches de l'application, en ayant par exemple des déploiements à des périodes différentes. Les équipes plus techniques, notamment l'équipe de développement framework, pourront intervenir sur d'autres couches de l'application sans impacter directement les couches métiers.
- La mise en place de ces services permettra l'ajout de nouveaux processus comme l'intégration de phases de tests automatisés, que chaque équipe mettra en place au fur et à mesure de l'évolution et du déploiement de leur service.

La réflexion sur la possibilité de faire évoluer le logiciel existant était centrée sur plusieurs limites remontées par la stack actuelle, notamment :

- Limite en termes de performance, les temps de réponse doivent être toujours plus rapides à contrario du nombre de données qui augmentent jour après jour Ce sont les changements apportés par la Big Data et la révolution digitale.
- Limite pour la maintenance logicielle, où la modification d'une petite fonctionnalité va impacter beaucoup d'autres éléments, liés à la complexité amenée au fur et à mesure des années de travail des équipes sur un « framework » datant d'une dizaine d'années et plus conforme aux attentes du web aujourd'hui, en termes de pattern et de principes guides.
- Limite pour la gestion de projet, avec des processus qui ont du mal à évoluer, à cause des limites précédentes et des habitudes prises qui ne sont aujourd'hui plus en accord avec une gestion précise, du fur et à mesure, gérée par des itérations avec des phases préparatoires, de tests, et de déploiement toujours complexe.

En regardant l'apport de l'architecture microservices, ces limites peuvent être solutionnées grâce aux différents processus de base mis en place dans cette architecture :

- Les performances peuvent être maitrisées grâce aux processus de scalabilité, et la capacité à répondre de manière plus intelligente à une demande cliente via l'orchestration.
- Les services étant autonomes, ils n'ont pas d'impact les uns aux autres pendant une maintenance ou un déploiement.
- Ce type d'architecture nécessite, pour les équipes, d'être constamment en phase avec le système construit. L'ajout de processus complet pour la gestion de projet, depuis les phases préparatoires aux phases de déploiement, devra être plus abouti et réfléchi pour convenir à ce type d'architecture.

Pour l'entreprise, d'autres avantages sont à noter :

- Au niveau du support, l'ajout de processus avant le déploiement permettrait de mieux fiabiliser l'application et de réduire les temps de maintenance par la suite.
- L'évolution technique pourra impacter fortement le marketing, par exemple l'ajout d'une application mobile sera plus facile et peut répondre à un besoin du marché.

Pour le moment, le projet d'évolution est en phase d'analyse, notamment :

- Analyse des besoins d'une nouvelle architecture (besoins existants, en attente, futurs)
- Recherche sur les architectures existantes, construction de matrices de sélections et essais techniques d'outils connus.
- Proposition de deux ou trois types d'architectures techniques qui répondent au mieux aux besoins et aux capacités actuels de l'entreprise.

Chapitre 2 : Hypothèses et développement méthodologique

Suite à l'état de l'art effectué sur l'architecture microservices et son impact sur les entreprises, j'ai voulu démontrer deux hypothèses opérationnelles, pour confronter la littérature et les besoins de l'application d'Ibiza avec le marché du développement web, domaine principal de l'entreprise en plus de l'édition.

Nous l'avons vu lors du premier chapitre, la partie front-end est une brique importante de l'application, amenée à être revue. Lors de la migration, l'utilisation d'outils tels des framework ou des librairies front-end pourrait être un avantage pour les équipes. La première hypothèse se concentre sur ce point, qui est un élément central de la migration et qui fait aussi partie de la réflexion concernant l'architecture microservices.

1. Première hypothèse : l'environnement du web et des outils front-ends

1.1. Définition de l'hypothèse

Un des premiers points que l'équipe a mis en avant lors des discussions concernant l'évolution architecturale est l'utilisation d'un framework front-end. En effet, nous l'avons vu lors de la description du logiciel actuel, la couche front (cliente) est actuellement une des plus complexes, car elle contient tout une partie du code métier de l'application. Or, il devient très important pour l'entreprise de vouloir simplifier cette partie, surtout en ce qui concerne la formation des collaborateurs. De plus, la nouvelle architecture basée sur les microservices inversera la tendance de développement : la partie front-end devra se séparer du « business », c'est-à-dire des fonctionnalités métiers pures, pour les déplacer côté serveur (ce qui renforce d'ailleurs beaucoup de points, comme la rapidité et la sécurité des données calculés). Cette partie front pourra aussi être scindée, pour suivre l'évolution poussée par les microservices sur le Back-End. Elle ne doit pas être dissociée de la réflexion lors de la mise en place de l'architecture, car c'est un point central de l'application d'Ibiza.

Hypothèse 1 : la mise en place d'une nouvelle technologie front-end lors de la migration, basée sur des outils existants, est le moyen le plus fiable de faciliter la formation de nouveaux développeurs, notamment lors du recrutement.

Cette hypothèse met en parallèle :

- le besoin de recrutement dans l'entreprise de développeurs pouvant être formés plus rapidement. En effet, Ibiza a du mal à intéresser les développeurs du marché, par un manque de technologies récentes.
- l'utilisation d'un outil front-end plus en raccord avec l'industrie du web aujourd'hui, qui permettra d'optimiser cette couche.

1.2. Méthodologie

Pour valider l'hypothèse, le premier objectif est de confirmer que les outils front-end sont utilisés et bien ancrés sur le marché du développement web. En effet, la mise en place d'un framework de ce type nécessitant une refonte de code non négligeable, il est important de vérifier leur utilisation à l'heure actuelle par les entreprises mondiales et françaises.

Pour cela, ma méthodologie se résume en trois points :

1) analyse de statistiques détaillées concernant l'utilisation des frameworks, comparé à une utilisation de JavaScript légacy (sans outils), comme aujourd'hui sur la partie front-end d'Ibiza.
2) analyse de l'évolution des principaux outils depuis ces quatre dernières années.
3) confrontation avec le marché actuel du développement web en France, grâce aux offres d'emplois actives sur le marché.

Les outils utilisés sont soit des framework, soit des librairies : de manière générale une librairie sera moins complète et orientée sur une fonctionnalité en particulier. Par abus de langage, ces deux types d'outils sont souvent regroupés sous le terme de framework front-end, indiquant un outil ou un ensemble d'outils permettant la création d'interfaces complexes, liés avec des interactions de fond (données, appels serveur, cache…).

1.3. Analyse

1.3.1. Analyse des technologies JavaScript front-end

L'objectif ici est de montrer la position actuelle des technologies JavaScript utilisées dans le cadre du développement web. Plus précisément, et cela pour renforcer la démonstration, je me suis focalisé sur une population spécifique, qui est le développement front-end utilisant des technologies JavaScript, séparée en deux groupes distincts. Le premier groupe fait référence aux entreprises utilisant le JavaScript légacy ou "vanilla", c'est-à-dire l'utilisation suivant les fonctionnalités natives du

langage de programmation JavaScript sans ajout d'outils, définit sous les standards ECMA[10] et en suivant leurs évolutions. Le deuxième groupe en comparaison représente la population de développeur utilisant des frameworks web basés sur le langage JavaScript.

Pour montrer le statut du marché actuel, je me suis aidé d'une collecte de donnée liée à une étude du site StateOfJS, ayant répertorié l'utilisation de framework front-end (basé JavaScript) mondialement. La population est large (plus de 20.000 développeurs interrogés) et mondiale (32 pays représentés). Cette étude montre que, mondialement, l'utilisation sans outils représente 23% des cas d'usage en entreprise, le reste représentant l'utilisation de frameworks et d'outils. L'utilisation de ses outils est partagée à 86% par les quatre frameworks les plus utilisés en front-end : React, VueJS, AngularJS et Angular.

Du point de vue français maintenant, il est intéressant de remarquer que la proportion d'utilisation dans les deux groupes est très proche de celle mondialement énoncée. En effet, elle représente 22% du sondage, pour l'utilisation du JavaScript vanilla, contre 78% pour une utilisation à l'aide d'outils. Le deuxième groupe est représenté par les quatre frameworks cités plus tôt dans 87% des cas. La tendance Français est donc très proche de celle mondiale en termes d'utilisation technologique et d'outils basés sur JavaScript.

Les résultats sont synthétisés via le tableau suivant :

Groupe analysé	Mondialement	France
JavaScript legacy	23%	22%
Avec framework	77%	78%

Tableau 2 : synthèse de l'analyse d'utilisation des outils JavaScript

Cette analyse permet de confirmer que l'utilisation d'un framework est un choix majoritaire dans les entreprises aujourd'hui situées dans le secteur du développement web. De plus, ce sont des outils que l'on apprend à maîtriser dès les écoles d'informatique, notamment Angular ou React (ou les deux), car ils permettent d'être très rapidement intégré dans l'environnement logiciel lors de l'entrée dans une nouvelle équipe. Ils proposent l'utilisation d'un environnement connu par le développeur, qui, au fur et à mesure de son expérience avec l'outil, pourra le comprendre et l'utiliser de façon plus instinctive.

10 Voir glossaire.

1.3.2. *Analyse de l'évolution des frameworks*

J'ai trouvé intéressant de me pencher sur la croissance des quatre frameworks principaux AngularJS, Angular, React et VueJS depuis leur apparition. En effet, une équipe aura une préférence pour un framework dont l'évolution est stable ou en hausse, impliquant dans ce cas qu'il est maintenu par une équipe et mis à jour.

La courbe suivante montre les tendances de questions publiées sur la plateforme StackOverflow, mondialement connue pour être un forum d'échange entre les développeurs professionnels ou non, mettant en avant l'activité communautaire des quatre outils.

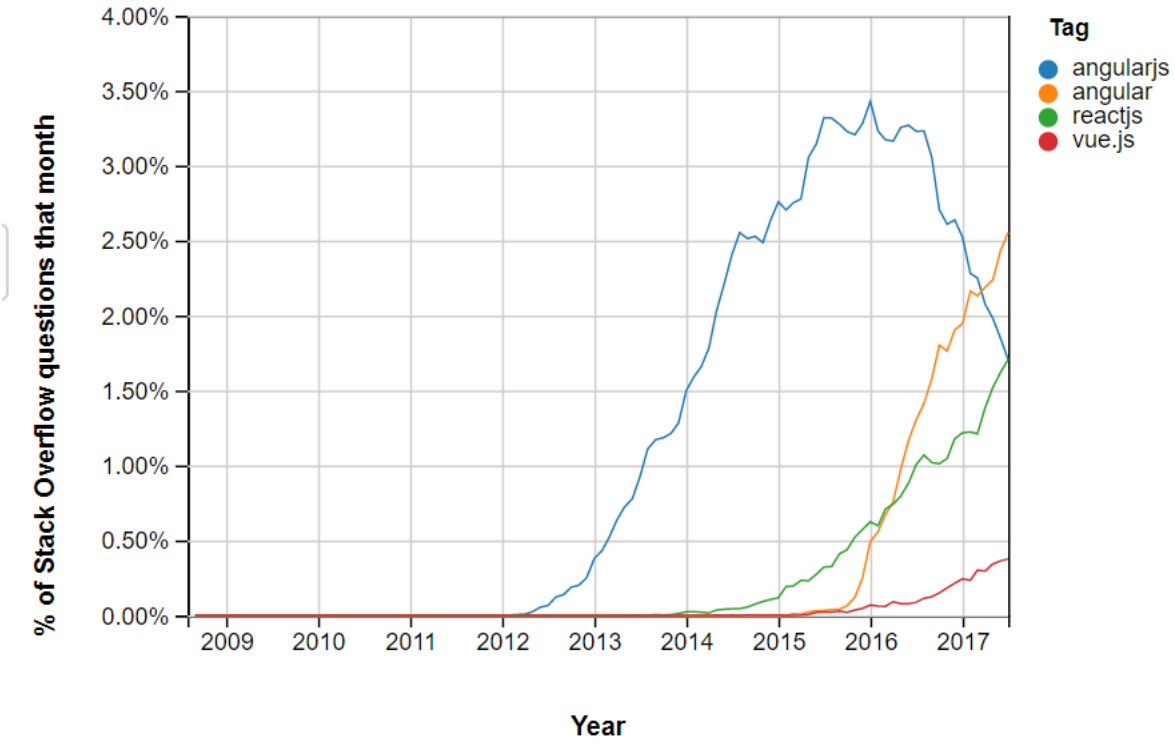

Figure 4 : tendance des quatre frameworks JavaScript sur la plateforme StackOverflow

On remarquera que le framework AngularJS est en baisse de tendance, car il est maintenant remplacé par Angular (aussi appelé Angular 2+), qui est une refonte complète du framework, et utilisé pour les nouveaux développements. AngularJS est encore maintenu par l'équipe d'origine pour éviter que les projets l'utilisant soient à refaire, mais son utilisation pour les nouveaux projets est à proscrire.

Il est important de souligner que ces frameworks sont tous en Open Source[11], c'est-à-dire que leur code source est disponible à tout le monde sur internet, et que la communauté mondiale collabore

11 Voir glossaire.

avec l'équipe de développement du framework pour faire évoluer le logiciel. Cela montre que la communauté de développeur à un fort impact sur l'utilisation mondiale de ces outils.

Le tableau en Annexe 2 montre plusieurs critères de comparaisons entre les trois frameworks Angular, React et VueJS, qui sont en concurrence lorsqu'une équipe doit choisir un outil lors de la mise en place d'un nouveau projet. Angular et React sont plus matures et ont une communauté plus large, et donc plus utilisés pour le moment. Le choix entre ces deux framework penchera surtout sur le besoin d'avoir un langage plus typé grâce au TypeScript pour Angular, permettant d'intégrer des développeurs Back-End plus facilement (similarité avec le C# et le Java), tandis que React sera moins complet au départ, ce qui peut être intéressant si l'équipe veut décider de son environnement. React est initialement une librairie, orienté sur l'interfaçage et les interactions entre les composants du site, mais est très vite complété avec d'autres modules en fonction des besoins des développeurs.

Pour confirmer que l'utilisation des frameworks est bien implantée dans les entreprises, j'ai voulu renforcer l'observation du marché par celui du recrutement actuel des développeurs web. Mon objectif est de vérifier que les offres d'emplois en France concordent avec les tendances remarquées plus haut. En effet, le recrutement en développement web est orienté sur les technologies utilisées par l'entreprise, qui a un besoin particulier, notamment via la maîtrise d'un ou plusieurs outils front-end.

1.3.3. Analyse du marché de l'emploi du développement web

Pour cette analyse, j'ai réalisé une étude basée sur la méthode booléenne de recherche d'informations. Cette méthode met en relation des données grâce à une représentation mathématique simple de la théorie des ensembles. Plus précisément, j'utilise une requête booléenne sur une base de données duquel je veux retirer mes informations précises. « Les opérateurs booléens sont les mots qui vous permettent de construire un langage pour interagir avec un moteur de recherche. » (Link Humans, 2015).

Dans ce cas d'analyse, l'ensemble est représenté par les offres d'emplois actives dans le domaine du développement web, et datant des trois derniers mois au maximum. Je me suis aidé des données de deux sites internet spécialisés dans le recrutement et la mise en ligne d'offres d'emplois : Indeed.fr et linkedIn.fr. J'ai utilisé mes requêtes booléennes sur leur moteur de recherche d'offres : en effet, les moteurs utilisent une méthode de recherche universelle basée sur les opérateurs booléens, ce qui facilite l'extraction d'informations sur des sites comme ces deux-là, mais aussi sur des moteurs de recherche (comme Google par exemple), des bibliothèques en ligne, etc.

1.3.3.1. Requêtes et critères

Le premier travail consiste à construire les requêtes qui me permettront d'extraire les données. J'ai choisi deux types de données à récupérer, via deux requêtes différentes :

- la première requête représente l'ensemble des offres d'emplois front-end demandant des compétences en JavaScript, tout en n'incluant pas les quatre frameworks principaux.
- la deuxième requête représente les offres d'emplois demandant la connaissance d'au moins un des quatre frameworks cités plus haut et/ou du JavaScript.

Ces requêtes correspondent aux deux ensembles suivants :

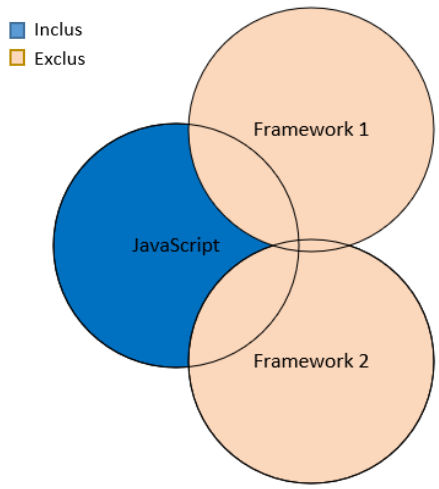

Figure 5.1 : Résultat requête 1

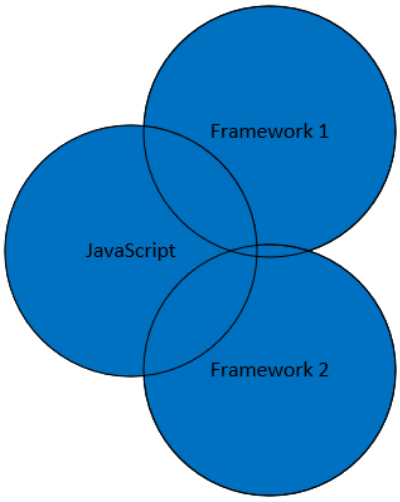

Figure 5.2 : résultat requête 2

L'objectif est de comparer les résultats de ces deux requêtes avec les données remontés des deux analyses précédentes, pour l'utilisation du JavaScript légacy (comparable au résultat de la première requête) ou d'un des frameworks front-end (résultat de la deuxième requête).

Ainsi, suivant la description des opérateurs booléens et du fonctionnement de l'ensemble de données, ma première requête consiste à extraire les offres d'emplois de développeur dont les compétences en JavaScript sont demandées, en excluant tous les autres frameworks. Cet ensemble représente les offres les plus similaires à ce que demande Ibiza aujourd'hui. En comparaison à cela, la deuxième requête prend en compte les offres demandant un ou plusieurs des frameworks voulus, en plus du JavaScript. Cela représente donc un ensemble de réunion. On pourra ensuite confronter la proportion d'offres équivalentes à celle d'Ibiza aujourd'hui, par rapport aux offres faisant intervenir les autres outils mondialement utilisés.

Les critères de sélections permettent alors de définir l'environnement des requêtes, afin d'être le plus précis possible. Suite aux critères, des mots-clés sont définis et utilisés dans les moteurs. Les mots-clés d'exclusion permettent une plus grande précision dans la recherche.

Définition des critères :

- Chaque emploi doit contenir exactement l'expression "Développeur front", représentant les offres d'emplois orientés front-end. Le métier d'intégrateur fait souvent parti des offres, donc à exclure, car ne rentre pas dans le cadre de l'emploi visé.
- Chaque offre doit contenir la référence au langage JavaScript. NodeJS est exclu de cette analyse, car orienté back-end, et est souvent remonté avec le mot-clé JavaScript.
- Les frameworks inclus dans la recherche sont les quatre les plus utilisés, le reste étant négligeable dans cette analyse.

Ces critères permettent de définir les mots-clés des requêtes :

Critères	Mots-Clés	Exclusion
Critère 1	Développeur front	Intégrateur
Critère 2	JavaScript	NodeJS
Critère 3	AngularJS, Angular, React, VueJS	

Tableau 3 : critères et mots-clés des requêtes

Les requêtes via les moteurs de recherchent utilisent des mots clés spécifiques pour les connecteurs logiques : AND, OR, NOT. L'utilisation des guillemets permet d'indiquer une expression complète à inclure dans la recherche.

La première requête est définie comme suit :

"Développeur front" AND JavaScript NOT (Angular OR AngularJS OR React OR VueJS) NOT Intégrateur NOT NodeJS

Ont inclus tous les résultats dont le poste correspond à un développeur front, et demandant du JavaScript. On exclut ensuite la réunion des autres frameworks ('Angular OR AngularJS OR React OR VueJS') ainsi que les mots-clés exclus, grâce au connecteur NOT.

La deuxième requête est la suivante :

"Développeur front" (JavaScript OR Angular OR AngularJS OR React OR VueJS) NOT Intégrateur NOT NodeJS

Cette fois-ci, on restreint les offres aux développeurs front, et demandant une ou plusieurs compétences parmi le JavaScript et les quatre frameworks.

1.3.3.2. Résultats

Les résultats des deux requêtes sur chacun des sites internet sont représentés sur le graphique suivant, en nombre d'offres d'emplois remontés :

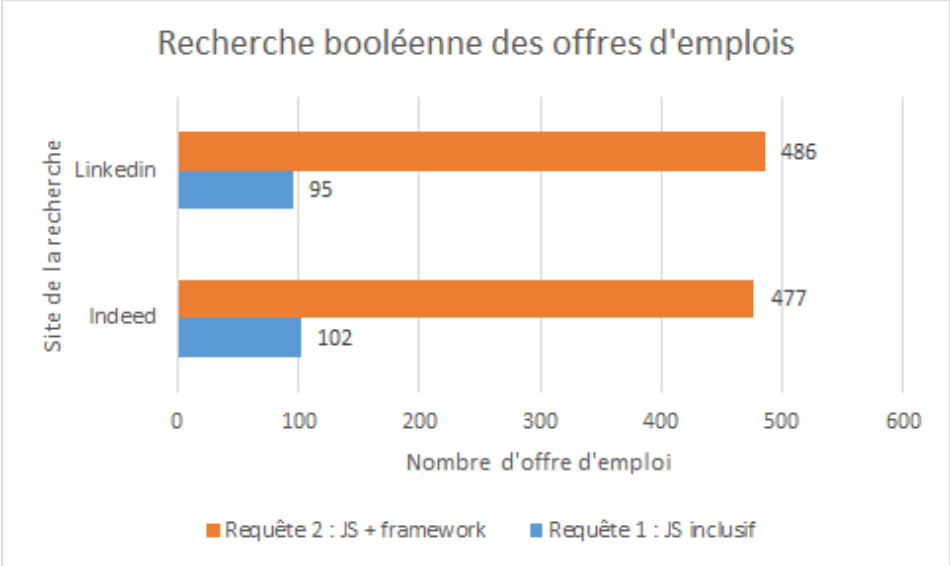

Figure 6 : résultat de la requête booléenne sur les offres d'emplois

La proportion d'offres ne demandant que du JavaScript, comparée à celles prenant en compte les frameworks, est de 19% et 21%, respectivement pour Linkedin et Indeed. On peut donc en conclure que les offres demandant des compétences en JavaScript exclusivement sont bien moins importantes que celles faisant intervenir les frameworks.

La réalité du marché de l'emploi, analysé sur la période actuelle, est très proche des cas d'utilisations du JavaScript avec ou sans framework, analysé via l'étude précédente.

Par cette analyse qualitative basée sur des statistiques et l'analyse du marché de l'emploi actuel, on remarque que moins d'un quart des entreprises travaillent sur des technologies de JavaScript natives en front-end. L'utilisation des frameworks comme outils d'aide au développement dans des équipes est un avantage, concernant notamment la formation des nouveaux développeurs entrants dans l'entreprise.

1.4. Mise en application chez Ibiza

Non avons donc validé, grâce aux trois études précédentes, que les frameworks front-end sont beaucoup utilisés dans le développement Web. Il faut maintenant valider le fait qu'ils auront un impact positif sur la formation et le développement au sein de l'entreprise.

Au niveau d'Ibiza, ce type de framework permettra de faciliter le travail des équipes à plusieurs niveaux.

1.4.1. Meilleure structuration des développements

En effet, les différents frameworks permettent une structure du code source similaire à tous les développements et à tous les projets. Un certain découpage peut être décidé par les équipes (séparation des modules dans le projet par exemple, gestion hiérarchique). De plus, ces outils permettront l'utilisation de la même version du code source, tout cela étant géré dans le projet par des Package Manager (type npm, yarn), qui permettent de définir les versions des différents modules utilisés dans le projet, et partagés par tous les collaborateurs.

1.4.2. Interface responsive

La plupart de ces framework incluent des outils permettant de gérer automatiquement les versions mobiles des sites internet. Pour cela, les équipes utilisent souvent des composants d'interfaces utilisateurs tels Material ou Bootsrap, qui découpent l'application en différentes briques, et dont le développeur définit certaines règles de positionnement en fonction de la taille d'écran. Sachant qu'aujourd'hui, le mobile correspond à plus de 60% du trafic global d'internet, il est nécessaire de prendre en compte ces utilisateurs lors du développement d'une interface web. De plus, Ibiza avait commencé à mettre en place une application mobile il y a cinq années de cela, mais le manque de ressource pour développer l'application n'a pas permis de la garder à jour et pleinement fonctionnelle. Le besoin d'une partie mobile est clairement un objectif pour le futur, auquel peut répondre très facilement ce genre d'outil, pour des fonctionnalités peu complexes. Dans le cas où l'interface demande beaucoup de ressources, et une mise en forme bien plus spécifique au mobile, la création d'une application mobile est envisageable plus tard.

1.4.3. Gain de temps sur la maintenance de librairie framework

Aujourd'hui, certains composants, comme les tableaux et grid dans l'application sont gérés par l'équipe framework d'Ibiza. En 2017, la maintenance de la DataList, utilisée en comptabilité pour les interfaces de balance et de saisie comptable, a pris entre 150 et 200 jours/homme (soit deux tiers

d'une année) en développement par un des membres de l'équipe framework. Cela est un investissement non négligeable, sachant que cet outil pourrait être remplacé par un autre déjà existant et géré par une entreprise externe. Par exemple, non avons récent testé un outil appelé AgGrid, très similaire à la grid d'Ibiza au niveau des besoins fonctionnels, mais bien plus rapide autant lors du chargement des données, que lors de fonctionnalités comme une gestion de filtre, groupage, etc. Des outils comme celui-ci permettraient à l'équipe framework de perdre moins de temps sur la maintenance d'outils développés en interne, utilisés côté UI, et qui seront voués à disparaître.

Ces apports sont poussés par les objectifs des frameworks aujourd'hui, dont le point fort est la productivité gagnée par une mise en commun d'une certaine architecture, d'un langage assez spécifique, mais qui permet de faire beaucoup de raccourci. Bien sûr, l'utilisation de framework a aussi des inconvénients, notamment l'évolution très rapide de ces composants. En effet, un framework comme Angular ou react a des évolutions tous les 2 à 3 mois. Il faut donc que les équipes se maintiennent à jour des nouveautés et assurent la mise à niveau du framework dans le projet. Certaines grosses évolutions nécessitent une refonte de code, qui peut être assez conséquent dans certains cas. Aujourd'hui, les organisations assurent qu'aucune grosse mise à jour n'entraînera une refonte complète des projets basés sur ces outils et des plans de déploiement sont accessibles à tous.

1.4.4. La formation

Enfin, l'utilisation d'un framework front-end permet le recrutement de développeurs connaissant déjà l'outil, et qui ont juste à se former sûr les données spécifiques de chaque équipe (comptabilité, révision …), et à s'approprier les méthodes de gestion de projet utilisées par les équipes (sprints mis en place, outils de gestion de projet). Aussi, un panel plus large de développeurs connaissant ces outils est présent sur le marché front-end, ce qui ouvre plus de possibilités de recrutement. Bien sûr, d'autres points sont aussi recherchés par les développeurs, comme un bon cadrage de projet, une précision sur la planification des projets, et d'autres facteurs humains.

1.5. Conclusion de l'hypothèse

Nous avons donc confirmé que :

- les frameworks sont utilisés par plus de 75% dans les entreprises en développement web,
- Ils sont en constante évolution et que leur tendance actuelle est croissante (sauf l'outil AngularJS qui est à éviter pour de nouveaux projets),
- Le marché de l'emploi français est en phase avec les observations précédentes,
- L'utilisation d'un framework front-end aura des avantages pour les équipes d'Ibiza.

Suite aux observations établies dans la première partie, la migration de la partie front-end est la seule façon d'intégrer un nouvel outil, car le code source doit être complètement retravaillé, en séparant la partie business. Profiter du chantier d'évolution de l'architecture est un bon moyen pour intégrer le processus de refonte du front-end, et la mise en place d'outils intéressants pour les équipes. L'hypothèse énoncée au début de ce chapitre est donc validée.

La partie front-end est en bonne voie pour être améliorée. La deuxième hypothèse a pour objectif de confronter différents points stratégiques de l'évolution en microservices avec certaines limites de l'application actuelle, afin de montrer que cette nouvelle architecture permet de répondre à certains problèmes rencontrés notamment par les équipes d'Ibiza.

2. Deuxième hypothèse : l'architecture microservices face aux limites de l'application Ibiza

2.1. Définition de l'hypothèse

Hypothèse 2 : l'architecture microservices est un bon moyen de remédier aux limites actuelles du logiciel, et aussi de bien s'équiper pour évoluer avec le web dans les années à venir.

Deux choses sont à dégager lors de l'analyse :

- Montrer que l'architecture répond à différentes limites et à des besoins actuels (scalabilité, suivi des évolutions, etc),
- Montrer que cette architecture permet de suivre les tendances du web.

L'état de l'art en chapitre 1 a permis d'avoir un aperçu de plusieurs points clés liés à une architecture microservices. En résumé, la littérature a permis de montrer deux choses :

- plusieurs éléments techniques sont à mettre en place lors de la conception de l'architecture (autonomie des services, scalabilité, automatismes),
- ces éléments et leur application dans un projet peuvent varier en fonction des entreprises et des différents besoins du projet.

Mon premier objectif est de valider les différents points techniques amenés par la littérature, notamment comparés à des projets réels dans des entreprises du web ; pour ensuite mettre en parallèle les points clés de l'architecture microservices et les limites du logiciel d'Ibiza aujourd'hui.

Pour confronter les résultats de l'état de l'art en Chapitre 1 et le statut des projets d'architecture microservices sur le terrain, j'ai réalisé une enquête.

2.2. Confrontation via l'enquête

2.2.1. Méthodologie

Cette enquête a pour objectif de mettre en parallèle les éléments importants lors de la conception d'une architecture microservices tirés de la littérature, et l'avis de professionnels qui ont travaillés sur des projets utilisant cette architecture. Pour cela, je me suis appuyé sur les différents éléments remontés par la littérature pour construire les questions.

La population visée a été très précise, j'ai moi-même envoyé le lien du questionnaire aux personnes et entreprises qui correspondaient aux critères que j'ai choisis. L'enquête a visé :

- des professionnels du web ayant travaillés sur des architectures microservices.
- des directeurs techniques d'entreprises du secteur d'Ibiza qui utilisent des technologies du web et une architecture microservices.

Je voulais interroger des personnes ayant déjà travaillées avec une architecture microservices et pouvant avoir eu une expérience en migration architecturale. Pour cela, j'ai contacté directement des entreprises du secteur et des professionnels (par mail et via LinkedIn notamment). Le nombre de réponses a été de 21 sur les 42 professionnels et entreprises contactés, soit un taux de 50%, sachant que l'enquête a été réalisée entre mi-juin et fin juillet 2018.

L'enquête a compris huit questions au total. Deux thèmes principaux ont été abordés : la conception d'une architecture microservices et la migration depuis une architecture monolithique.

Chaque question comprenait des critères tirés de la littérature, que les professionnels pouvaient classer et évaluer selon leur expérience, l'objectif étant de valider l'importance des critères comparés aux autres. Les questions détaillées sont disponibles en Annexe 3.

2.2.2. Réponses

2.2.2.1. Conception architecturale.

Huit points ont été mis en concurrence lors de la phase de conception. Ce sont des principes et techniques mis en place dans une architecture microservices, tirés de l'état de l'art en Chapitre 1. Mon but était d'ordonner les points principaux en fonction de l'expérience des professionnels interrogés.

Pour cela, chaque critère était à classer en fonction de son degré d'importance lors de la phase de conception, de 1 (peu/pas important) à 5 (très important). Une case « N/A » permettait d'indiquer qu'un critère n'étant pas à prendre en compte lors de la conception.

Voici le résultat de l'enquête concernant les huit points expérimentés :

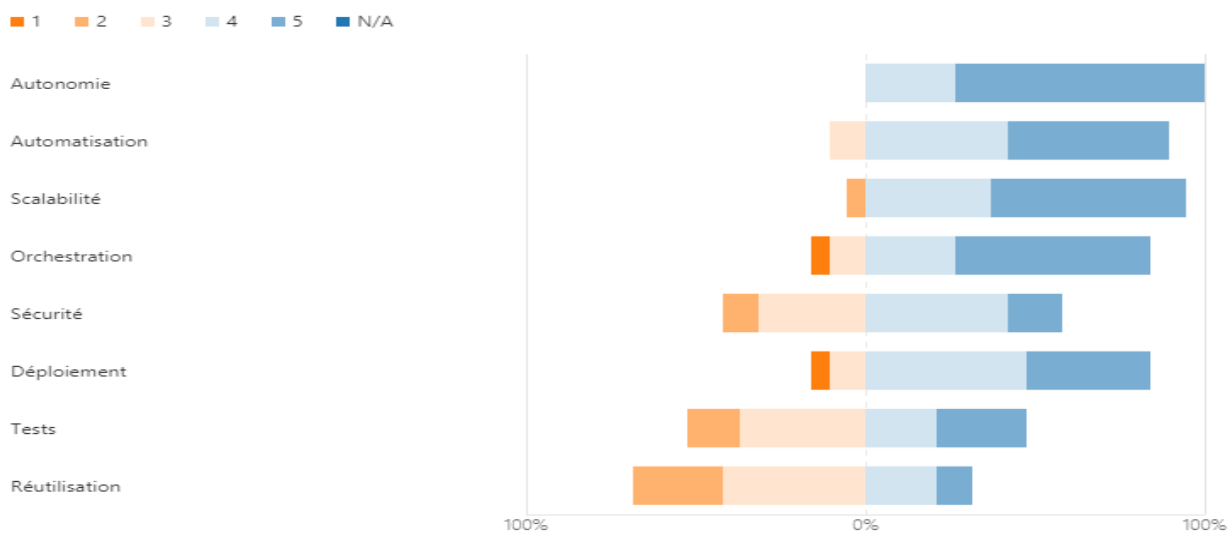

Figure 7 : résultat de l'enquête - huit critères lors de la conception

Selon ces résultats, cinq points sur les huit ont un impact important lors de la conception :

- l'autonomie, qui concerne notamment le principe d'autonomie d'un microservices qui doit être dans un conteneur propre, et gérer une seule fonctionnalité principale.
- l'automatisation, qui concerne les processus mis en place pour simplifier et automatiser plusieurs points indispensables (déploiement, mise à jour, tests).
- la scalabilité, principe permettant la multiplication d'un microservices lors de besoins et de demandes en hausse dans le cycle de vie de l'application.
- l'orchestration, qui concerne tous les outils mis en place pour coordonner les services et leur communication.
- le déploiement, phase de lancement en production des microservices.

Les trois autres points sont remontés comme moins important lors de cette phase : la sécurité, les tests et le principe de réutilisation.

Ce résultat montre que les cinq premiers critères sont les plus importants à approfondir au début de la réflexion d'architecture, notamment sur les processus mis en places et les solutions techniques de fond. Les trois points moins importants sont soit très spécifiques et moins liés à la conception (notamment la sécurité, qui est plutôt orienté sur l'infrastructure et l'authentification), ou amenés plus tard (notamment les tests qui sont mis en place lors de la rédaction des cahiers des charges). Ce résultat peut donc aider les équipes à orienter leur travail lors de la conception sur des critères jugés plus importants au départ.

2.2.2.2. Avantage comparé au monolithique

Une deuxième question intéressante à explorer est l'avantage de l'architecture microservices comparée à celle monolithique. Six critères étaient distingués, et trois de ces critères sont sortis du lot, avec un taux de réponse important :

- la maîtrise de la scalabilité (95%)
- les processus de déploiement (80%)
- faciliter l'échange et l'interopérabilité entre les services (80%)

Les deux premiers critères sont liés à la question précédente, ce sont des points forts de l'architecture, il n'est donc pas étonnant de les retrouver comme points forts de comparaison entre les deux architectures. Le troisième critère montre que l'interaction avec l'extérieur et entre les services est aussi un point fort et un avantage face à une architecture monolithique.

Il faut aussi remarquer un critère qui a été assez disputé avec 60% de réponse : le choix d'une technologie différente entre les services. Cela montre que, dans certains cas, le choix technologique n'est pas au cœur de la décision et du travail des équipes. En effet, c'est un point qui est plus difficile à aborder, car si les équipes mettent en place des technologies différentes par service, elles auront une spécialisation différente et il sera plus difficile pour des développeurs de passer d'une équipe à une autre (prendre en compte une phase de formation par exemple).

2.2.2.3. Architecture monolithique aujourd'hui

Bien que ce soit une architecture assez ancienne, l'enquête montre qu'elle est encore jugée comme utilisable pour des nouveaux projets. Plus précisément :

- 32% considèrent que c'est une architecture applicable à de nouveaux projets,
- 63% considèrent qu'elle est applicable dans des cas très précis, donc que c'est une architecture à ne plus considérer par défaut,
- 5% considèrent que cette architecture est dépassée et plus applicable.

Il est donc intéressant de noter que cette architecture n'est pas encore complètement sortie des techniques de construction logicielle web, mais qu'elle est plutôt mise en second plan lors du choix d'architecture technique.

2.2.2.4. Points clés de la migration

Ensuite, mon objectif a été de confronter trois points importants remontés par la littérature lorsque l'on effectue une migration, comme nous l'avons vu dans la littérature :

- le découpage des services,
- les méthodes de déploiement,
- les règles d'orchestration.

Le résultat de l'enquête sur cette question est très homogène : respectivement 34%, 32% et 34% de taux de réponse. Cela montre qu'en effet ces trois points sont à prendre en compte au même niveau lors de la migration. C'est très intéressant, car, même si la littérature a permis de les remonter spécifiquement, ils n'étaient pas classifiés en niveau d'importance. Les réponses montre qu'ils sont vus au même niveau, donc à travailler avec les mêmes capacités et sont aussi important les uns que les autres.

2.2.2.5. Impacts de la migration

Le dernier point remonté par l'enquête concerne les critères principaux d'impacts de la migration monolithique en microservices.

Comme pour la question précédente, les quatre points remontés par la littérature ont été mis en concurrence :

- la réutilisation des services
- la facilité de mise à jour du logiciel en production
- la hausse de performance
- une meilleure qualité logicielle et une baisse de support bas-niveau

Les taux de réponse ont été respectivement de 18%, 32% ,29%, 18%. Ainsi, selon le retour de cette enquête, le passage sur une architecture microservices permettrait d'impacter plus fortement la facilité de mise à jour et les performances, et un peu la réutilisation ainsi que l'impact sur la maintenance. Il faut aussi remarquer que même si cette question montre que deux points sont remarqués comme plus impactant, les deux autres peuvent aussi jouer, et que cela dépendra également de l'objectif visé par l'équipe lors la migration.

2.2.3. Conclusion de l'enquête

Cette enquête a permis de classifier et de valider certains critères remontés par la littérature, en ce qui concerne l'architecture microservices et la migration depuis une architecture monolithique. Les différents résultats vus précédemment vont me permettre d'appuyer la suite de mon résonnement sur des points plus précis et plus important pour la confrontation avec Ibiza.

2.3. Impact pour le logiciel d'Ibiza

Dans cette partie, nous allons confronter différents points clés de l'architecture microservices vus précédemment, avec les limites et les besoins actuels du logiciel. Cela permettra de valider ou non que l'architecture mircroservices permet de répondre aux limites actuelles, et est donc un bon choix pour le projet de migration. Pour ce faire, j'ai décidé de me focaliser sur les points principaux sur lesquels s'articule la phase de conception de l'architecture, en prenant les cinq critères remontés par l'enquête : autonomie, automatisation, scalabilité, déploiement, orchestration. Chaque point sera redéfini succinctement pour repositionner leurs objectifs principaux, et ensuite voir ces avantages face aux limites d'Ibiza.

2.3.1. Autonomie

Principe de base du microservices, l'autonomie est une caractéristique importante à mettre en place et à respecter. Comme vu dans la littérature, ce principe implique une isolation d'exécution et une isolation de déploiement.

Pour Ibiza, cela nécessite de définir en premier lieu une granularité des microservices qui permettra d'utiliser au mieux ce principe. En effet, pour qu'un microservices respecte une autonomie propre à son fonctionnement et à son déploiement, il faut qu'il contienne toutes les fonctions nécessaires à son exécution, dans la limite de ce qu'il lui est donné de faire.

Prenons l'application actuelle. Dans l'architecture monolithique d'Ibiza, aucun module ne peut fonctionner par lui-même : en effet, l'application va charger en premier des éléments nécessaires au lancement de chaque module, notamment le framework d'Ibiza et le Cache d'Ibiza (librairie de fonctions utilisables dans toute l'application). Si on prend le code source de la comptabilité à l'heure actuelle, il ne pourra être utilisé en autonomie, car il n'a pas été développé au départ pour respecter ce principe, et cela en est de même pour tous les modules internes au logiciel. Certaines fonctionnalités, plutôt rares, ont déjà été sorties du gros de l'application, et sont déployées en tant que services, mais ne sont pas indépendantes de l'environnement.

Aussi, nous avons parlé de granularité comme une décision de découpage du logiciel monolithique en microservices. Pour que cette granularité puisse être fonctionnelle dans le cadre d'Ibiza, il faut que chaque microservices puisse s'exécuter d'abord en suivant ses propres fonctions, et ensuite qu'il puisse appeler d'autres services dans le cas où il doit déclencher des fonctionnalités développées à l'extérieur de son propre environnement.

En effet, les interactions entre les modules sont assez présentes dans le cycle de vie de l'application : par exemple, la saisie d'une facture dans le module de facturation va déclencher des écritures dans le module comptable. Il en est de même lors de génération de charges fournisseurs du module révision. Chaque module a donc une possibilité d'interagir avec un autre. Mais le principe d'isolation ne se pose pas : l'application actuelle étant déployée entièrement, si le module de révision fonctionne, celui de comptabilité aussi, donc la génération sera possible.

À ce niveau, le principe d'autonomie pose donc pour Ibiza :

- Un besoin de granularité pour une isolation d'exécution avec des fonctionnalités autonomes,
- Mais qui doit être assez souple pour un appel d'exécution depuis l'extérieur,
- Et dont la technique permettra une isolation de déploiement.

Concernant la granularité, il sera plus difficile à définir que par un simple découpage existant pour chaque module. En effet, par le fait que l'application actuelle soit définie en modules, et que les équipes de développements métiers sont aussi définies en fonctions de ces modules, il pourrait être facile de mettre en place un découpage par module uniquement. Cela serait certes plus agréable dans la gestion des microservices : chaque équipe s'occupe du sien, et le déploie à tout moment. Oui, mais on en oublie le terme « micro » des « microservices » : le module de comptabilité à aujourd'hui beaucoup de fonctionnalités, qui, regroupés dans un seule service, serait une petite application à elle seule, ce que nous voulons éviter. Un microservices doit faire une seule chose. Il doit faire de la génération d'écriture, ou de la génération de rapprochement bancaire. Mais il ne peut pas cumuler les deux.

Cela dit, on sait aussi aujourd'hui que ces fonctionnalités sont liées. Un rapprochement bancaire n'a pas d'intérêt si la génération d'écriture de banque n'existe pas dans l'application, car aucune donnée ne pourrait être traitée, étant inexistante. Ces fonctionnalités sont autonomes techniquement (exécution, déploiement), mais sont utiles dans l'environnement complet du logiciel. Elles sont liées par un des cinq principes majeurs : l'orchestration.

C'est en effet le liant des fonctionnalités. Là où un microservices est développé comme autonome et scalable, il ne sera utile que s'il est intégré à un environnement orchestré, car il pourra être enregistré puis appelé et exécuté selon une demande d'un client ou d'un autre service. Il pourra générer des données et les insérer dans la base, bref : il sera pleinement fonctionnel et acteur de l'environnement.

Le principe d'autonomie et de la définition d'une bonne granularité permettront en premier lieu de recentrer le travail des équipes. Au lieu de mélanger des fonctionnalités dans une seule application,

les microservices auront un seul objectif, ce qui facilitera en premier leur développement futur : si cette fonctionnalité doit être améliorée, ou maintenue, l'équipe sait quel service est à évoluer.

De plus, cela permet d'utiliser des technologies différentes en fonction des besoins. Par exemple, à l'heure actuelle et en prenant en compte les compétences des développeurs, on sait que pour certains services nécessitant la manipulation d'une masse importante de données, on préfèrera l'utilisation d'une technologie C#. Pour le cas d'un service d'enregistrement de données simples via l'agrégation de plusieurs autres données, l'utilisation du NodeJS permettra de construire un service fiable et rapide, très facilement.

L'autre aspect, mais non le moindre, est la résolution d'erreur. Dans l'application monolithique, la file d'appels des fonctionnalités se suit. Si l'une d'elles plante et que, pour une raison ou pour une autre, l'erreur est mal remontée à la fonction initialement déclenchée, le développeur mettra un certain temps à débugger. De plus, une erreur dans la file fait arrêter tout le déroulement. Si les fonctionnalités sont dans des microservices, il est non seulement plus facile de détecter d'où vient le problème, mais il sera aussi restreint à un seul service, et non à tous les autres, ce qui ne pourra faire réagir ni l'application en entière (sauf erreur dans l'infrastructure réseau), ni la fonctionnalité présente qui a demandé un traitement au service, car ils ne sont pas dans le même environnement. C'est donc un gain en de stabilité dans le traitement des erreurs.

2.3.2. Scalabilité

Ce principe est un point clé qui impactera grandement les performances de l'application. Le plus gros problème de l'architecture monolithique est la dépendance des fonctionnalités en matière d'infrastructure. Elles sont toutes regroupées sur le même système, donc en cas de forte demande sur une fonctionnalité, elle monopolise beaucoup de ressources (mémoire, processeur), au détriment des autres. Pour éviter cela, l'application d'Ibiza est dupliquée entièrement sur plusieurs machines. Ce n'est pas effectué en temps réel, donc lorsque 7 machines sont déployées, seules une ou deux seront utilisées pleinement lors de petites demandes (en été, la nuit), et les autres seront démarrées, généreront des frais, mais ne seront pas utilisées. De plus, au sein même de chaque duplication du logiciel, la consommation ne sera pas homogène.

L'avantage d'un microservices consiste à renforcer la scalabilité par duplication de ce microservices seul, en cas de forte demande sur la fonctionnalité mise en place par le service. Cela n'impacte pas le système des autres services, car ils sont indépendants. Au lieu de redéployer toute l'application pour cette fonctionnalité, seul son service est déployé, ce qui est un gain de ressource non négligeable à ce moment. Mais finalement, lorsque tous les microservices seront déployés, le gain pourra être en

matière de coût d'infrastructure, grâce à une maitrise de la consommation système (le nombre de serveurs à louer), mais aussi la facilité des fonctionnalités à s'adapter aux fluctuations des demandes clientes. L'enjeu derrière, et l'on revient au principe cité précédemment, est la mise en place d'un bon système d'orchestration, qui est responsable de la duplication des microservices en cas de besoin.

2.3.3. Déploiement

Le déploiement est aussi très important, car il définit les techniques de lancement des microservices, pour la création d'un système complet nécessaire au bon fonctionnement de chaque fonctionnalité. Aujourd'hui, des techniques utilisant des conteneurs sont mises en place pour lancer rapidement chaque environnement, et alléger la création d'un système d'exploitation, partagé, au lieu de le dupliquer comme c'est le cas avec des machines virtuelles. Pour les développeurs d'Ibiza, cela facilite beaucoup les tests à effectuer sur les fonctionnalités, cloisonné au test d'un seul microservices pour chacune.

Le déploiement concerne aussi l'intégration continue. C'est un processus qui lie les microservices avec la gestion des builds et la gestion de version. L'objectif est de garder une cohérence et une synchronisation entre les mises à jour effectuées sur le code source du service, souvent situé sur un serveur de gestion de code source (comme Git), et la version du microservices déployé en production. L'intégration continue est un processus techniquement déployé comme un service, qui est situé sur un serveur spécifique et effectuant des vérifications sur les nouvelles versions mise à jour par les équipes de développement : compilation, validation des différents tests unitaires et fonctionnels notamment.

Ce processus est actuellement absent de l'application d'Ibiza. Outre le fait que beaucoup de tests manquent, la vérification des nouvelles versions est effectuée manuellement, lors des phases de validation des versions tous les trois mois. Ensuite, la compilation et le redéploiement de l'application sont manuels. Dans un environnement microservices, cela n'est plus envisageable, car l'environnement est trop complexe pour une intervention manuelle lors de la mise à jour des services. C'est un processus qui permet de sécuriser le déploiement de nouvelles versions, grâce aux vérifications préalables et une synchronisation constante entre les microservices et les codes déployés par les équipes.

2.3.4. Orchestration

Le service d'orchestration permet la centralisation des moyens de communication. Les microservices créés sont autonomes, scalable et se déploient correctement. Mais quel microservice faut-il appeler pour effectuer une tâche particulière ? Dans le cas d'une chaîne d'action à effectuer, dans quel ordre

doit-on les appeler ? C'est le service d'orchestration qui se charge de résoudre ces interrogations. Il connaît l'objectif de chaque microservice, et redirige correctement les requêtes clientes au service correspondant. Il sait aussi gérer les erreurs remontées par un microservice.

De plus, c'est aussi un bon moyen d'avoir une vision en temps réel du fonctionnement de l'environnement de l'application : combien de microservices sont lancés, quels types de requêtes sont fréquentes, quels sont les services les plus utilisés à tel moment de la journée, du mois, etc. Toutes ces informations sont intéressantes pour permettre aux équipes de comprendre comment les clients utilisent l'environnement, et ensuite d'effectuer les modifications nécessaires (scinder les fonctionnalités d'un service si son utilisation n'est pas homogène, poser les limites de déploiement des microservices, optimiser ceux qui consomment beaucoup de ressources…). L'application actuelle ne nous permet pas de remonter rapidement et précisément toutes ces informations.

2.3.5. Automatisation

L'automatisation est au cœur d'une architecture microservices, c'est l'ensemble des processus qui permettent le déploiement, le monitoring, les tests et l'intégration continue : bref, tout ce qui peut être fait de façon automatisée, dans un environnement où beaucoup de processus suivent des tâches précises et récurrentes.

Cela permettrait notamment d'alléger les processus lors du déploiement d'une nouvelle version, et de faire gagner du temps à beaucoup d'acteurs de l'entreprise. Les responsables produits n'ont pas à tester les mêmes choses à chaque version majeure, l'équipe d'infrastructure n'a pas à déployer manuellement les différentes briques de l'application, les codes sources écrits par les équipes de développement sont vérifiés de façon plus précise et suivent un processus détaillé. Tous ces éléments peuvent être présents pour faciliter la gestion de l'application actuelle, mais seront encore plus obligatoires dans le cas d'une architecture microservices.

2.4. Futur du logiciel et de l'architecture

Cette nouvelle architecture offre un suivi et une possibilité d'évolution non négligeable dans les années à venir. Chaque microservices peut être évolué sans refondre tout ou parti de l'application. Que ce soit pour une évolution technologique, un ajout de fonctionnalité, une réécriture du service, l'impact sur l'environnement à ce moment est minime et a plusieurs avantages :

- Le nouveau service peut être déployé en parallèle des anciens, dans le cas où il faut intervenir sur des services extérieurs pour changer la méthode d'appel.
- Si un changement technologique est réalisé, cela n'impacte pas les autres services, et en cas de plan plus global, la refonte sera effectuée en suivant la structure des services et sera faite au fur et à mesure.

Ce découpage, que l'on a défini comme choix central dans l'architecture microservcices, permettra aussi dans le futur de mieux réagir aux différentes évolutions du logiciel. De nouveaux modules peuvent être créées, d'autres supprimés, et cela plus facilement. Certains services peuvent être rendus accessibles de l'extérieur, en faisant un écosystème ouvert pour ouvrir des fonctionnalités à des partenaires ou aux clients.

Aussi, le découpage front-end et Back-end est très clair. Comme nous l'avons vu au début de ce chapitre, la partie front-end peut être gérée par un framework complet JavaScript. Toute la partie de développement business est sur le Back-End, qui est aussi responsable de la récupération des données pour les renvoyer au client. Ce découpage permet facilement l'ajout de plusieurs front-end : en effet, l'ajout d'une application mobile nécessite par exemple une partie Cliente différente soit via la création d'une application mobile native (Android, iOS) ou via la création d'une interface web spécifique. Les services présents en Back-end pourront tout à fait être utilisés pour cette nouvelle application, en fonction des besoins. De même, imaginons que l'application Ibiza soit un jour découpée (une application paie, une application comptable) pour permettre de proposer à des clients une interface orientée sur un seul métier spécifique, il en est de même de la réutilisation des microservices.

Bref, comparé à l'architecture actuelle d'Ibiza, il est évident que l'architecture microservices, même si elle demande une technique plus complexe, est un grand avantage pour l'évolution futur.

Voici un tableau permettant de résumer les différents points remontés :

Points-clés	Apport pour la nouvelle application
Autonomie	Meilleure séparation des fonctionnalités pour agir sur plus de paramètres indépendants. Séparation des technologies, dépendant du type de données traitées par le service. Isoler les erreurs.
Scalabilité	Gestion de la demande plus flexible, axée sur un ou plusieurs microservices. Mieux maîtriser les ressources utilisées par l'infrastructure. Augmenter les performances en temps de réponse, temps d'exécution et accessibilité à tout moment.
Déploiement	Intégration continue pour fiabiliser le déploiement des services. Utiliser des technologies orientées conteneurs qui construisent des environnements spécifiques par services et autogèrent les ressources. Multiple environnement pour s'adapter aux besoins des fonctionnalités.
Orchestration	Gérer l'accès aux différentes fonctionnalités : enregistrement, redirection des appels, remontés d'erreurs.
Automatisation	Faciliter tous les processus. Du développement aux tests et au déploiement. Éviter de répéter manuellement des processus qui prennent du temps et qui sont automatisables facilement.

L'application actuelle en monolithe à deux avantages comparée à une architecture microservices :

- Le déploiement de l'application est fait en une seule fois. Même si c'est un processus long et lourd.
- Les développeurs font facilement les liens entre les fonctionnalités, car elles sont sur le même bloc de code source. Dans l'environnement découpé en services, il faudra utiliser des services intermédiaires pour gérer les demandes.

générale et pour une démonstration précise m'a permis de connaître davantage la place de chacun dans le monde du développement web. Ce travail a donc été enrichissant par la découverte et l'étude d'un aspect du web, et de voir son application concrète sur un cas d'entreprise spécifique, ce qui enrichit mon expérience professionnelle. Ce travail m'a aussi permis de participer aux discussions en tant que membre de l'équipe framework, de toucher à de nouvelles technologies dans les essais menés par l'équipe et d'apercevoir la complexité de ce type de projet dans une entreprise informatique comme celle d'Ibiza.