

REPUBLIQUE DU CAMEROUN
Paix-Travail-Patrie

UNIVERSITE DE YAOUNDE I

CENTRE DE RECHERCHE ET DE FORMATION
DOCTORALE EN SCIENCE, TECHNOLOGIE ET
GEOSCIENCES

UFRD Sciences de l'Ingénieur



REPUBLIC OF CAMEROON
Peace-Work-Fatherland

UNIVERSITY OF YAOUNDE I

POSTGRADUATE SCHOOL OF SCIENCE,
TECHNOLOGY AND GEOSCIENCES

Doctorale research unit for engineering and
application

EVALUATION DE LA MISE EN ŒUVRE DES ARCHITECTURES MICRO-SERVICES : APPLICATION AUX PLATEFORMES SPRING BOOT ET JAVA EE

Mémoire de fin de cycle master

Présenté et soutenu par

MASSAGA NGONDA ARISTIDE MARIE
12P062

En vue de l'obtention du diplôme Master 2 Recherche

Spécialité : Génie Informatique

Sous la Direction de :

- **Pr. KOUAMOU Georges Edouard**

Devant le jury composé de :

- **Pr. Thomas BOUETOU**, Président
- **Pr. Georges Edouard KOUAMOU**, Rapporteur
- **Dr. Bernabé BATCHAKUI**, Examineur

Année académique 2019-2020

DEDICACE

*Je dédie ce travail à ma **Famille**.*

REMERCIEMENTS

Ce travail est l'aboutissement de nombreux efforts et sacrifices qui n'aurait jamais été accompli sans l'aide et le soutien des personnes chères :

J'adresse mes remerciements les plus sincères au **Pr Thomas BOUETOU**, qui me fait l'honneur en acceptant de présider le jury d'évaluation de ce travail.

J'exprime ma gratitude envers le **Dr Bernabé BATCHAKUI**, pour avoir accepté d'examiner mon travail.

J'adresse mes sincères remerciements à mon encadreur **Pr Georges Edouard KOUAMOU** pour m'avoir donné l'opportunité de travailler sur ce sujet dont la thématique est fascinante. Je le remercie pour son grand soutien scientifique et moral, pour les conseils, suggestions et les encouragements qu'il m'a apporté pendant l'élaboration de ce mémoire.

Mes remerciements s'adressent également à l'ensemble du corps enseignant du Master II de l'ENSP pour leurs précieux enseignements, ainsi qu'à tous mes camarades de la promo 2018.

A mes parents MASSAGA Emmanuel et AKAMBA Joséline, à mes frères et sœurs : Gwladys, Yannick, Pierrette, Emmanuelle, Marc, Audrey et à toutes les grandes familles MASSAGA et NDENGUE pour tout leur soutien, leurs aides, leurs conseils.

Je remercie mes amis, de la paroisse Sainte Thérèse de l'Enfant Jésus de Nkol-Nguié, pour leurs conseils, leur attention et leur sollicitude.

J'exprime mes amitiés à toutes les personnes qui me sont chères.

Mes remerciements s'adressent aussi à tous ceux qui ont contribué de près ou de loin à la réalisation de ce travail et dont les noms ne sont pas mentionnés dans ce document.

Je ne saurais finir sans remercier le **Seigneur tout puissant** sans qui tout ceci ne pourrait être possible.

RESUME

Avec l'essor du cloud computing, on a assisté une conception des applications de plus en plus distribués. C'est dans ce contexte, qu'est né le style architectural micro-services, héritier des styles par composant et des styles orientés services. Ce style offre de nombreux atouts mais aussi des problèmes, liés notamment au caractère très distribué des services. Ce travail de master recherche a pour objectif, de construire un modèle d'évaluation des plateformes couramment utilisés pour la mise en œuvre des applications dans le style micro-service. Il consiste d'une part, en un ensemble de critères d'évaluation, obtenus en divisant l'implémentation du style en plusieurs domaines de conception suivant la méthodologie de conception piloté par domaine (DDD), puis en recherchant les patrons de conception qui font office de bonne pratique pour l'implémentation des atouts et problèmes liés au style ; et d'autre part une fonction d'évaluation qui attribue une note à une plateforme tout en tenant compte des exigences spécifiques de l'application développée. En appliquant ce modèle aux plateformes Spring Boot et JAVA EE, avons constaté que Spring Boot obtient un score de 96,3% tandis que JAVA EE est à 44,4%. Ces scores témoignent de l'effort nécessaire pour conformer une application avec les préoccupations de ce style de développement.

Mots clés : architecture logicielle, micro-service, modèle d'évaluation, Spring Boot, JAVA EE.

ABSTRACT

With the rise of cloud computing, we have witnessed an increasingly distributed design of applications. It is in this context that the micro-services architectural style was born, the heir to the component-based and service-oriented styles. This style offers many advantages, but also problems, linked in particular to the highly distributed nature of services. The aim of this master's research work is to build an evaluation model for platforms commonly used to implement applications in the micro-service style. It consists, on the one hand, of a set of evaluation criteria, obtained by dividing the implementation of the style into several design domains according to the Domain Driven Design (DDD) methodology, and then searching for design patterns that serve as good practice for the implementation of the assets and problems related to the style; and on the other hand, an evaluation function that assigns a score to a platform while taking into account the specific requirements of the application developed. Applying this model to the Spring Boot and JAVA EE platforms, we found that Spring Boot scores 96.3% while JAVA EE scores 44.4%. These scores reflect the effort required to conform an application with the concerns of this style of development.

Keywords: software architecture, micro-service, evaluation model, Spring Boot, JAVA EE.

TABLE DES MATIERES

DEDICACE.....	ii
REMERCIEMENTS	iii
RESUME.....	iv
ABSTRACT	v
TABLE DES MATIERES	vi
LISTE DE FIGURES	ix
LISTE DES TABLEAUX.....	x
GLOSSAIRE	xi
INTRODUCTION GENERALE.....	1
Cadre du mémoire	1
Problématique.....	2
Objectif.....	3
Organisation du travail	3
CHAPITRE 1 : ETAT DE L'ART	5
Introduction	6
1.1 Architectures logicielles	6
1.1.1 Historique et évolution	6
1.1.2 Définition	7
1.1.3 Langage de description des architectures (ADL)	9
1.1.4 Styles architecturaux	11
1.1.5 Bilan	14
1.2 L'architecture micro-service.....	15
1.2.1 Émergence du style micro-service	15
1.2.2 Définitions notables.....	17
1.2.3 Caractéristiques du style micro-service.....	17

1.2.4	Intérêts du style micro-service	20
1.2.5	Problèmes de conception et d'implémentation	20
1.2.6	Patrons de conception.....	22
1.3	Évaluation des applications micro-services.....	23
1.3.1	Exigences requises pour une évaluation.....	24
1.3.2	Système de références pour l'évaluation.....	27
	Conclusion.....	29
CHAPITRE 2 : MODÈLE D'ÉVALUATION DES PLATEFORMES POUR L'IMPLÉMENTATION DU STYLE MICRO-SERVICE		31
	Introduction	32
2.1	Exigence d'implémentation par domaine	32
2.1.1	Gestion des données	33
2.1.2	Gestion des tests (Testing)	36
2.1.3	Déploiement	36
2.1.4	Préoccupations transversales.....	37
2.1.5	Style de communication	38
2.1.6	API externe.....	39
2.1.7	Découverte de service	42
2.1.8	Fiabilité.....	45
2.1.9	Sécurité.....	45
2.1.10	Observabilité	46
2.1.11	Modèles d'interface utilisateur	47
2.2	Modèle d'évaluation	48
2.2.1	Critères d'évaluation	48
2.2.2	Fonction d'évaluation.....	50
	Conclusion.....	52

CHAPITRE 3 : ÉVALUATION DES TECHNOLOGIES SPRING BOOT ET JAVA EE	53
Introduction	54
3.1 Évaluation de Spring Boot 2.2.2.....	54
3.1.1 Présentation	54
3.1.2 Recherche des valeurs de la fonction h	57
3.1.3 Calcul de degré de compatibilité	60
3.2 Évaluation de JAVA EE 7	60
3.2.1 Présentation	60
3.2.2 Recherche des valeurs de la fonction h	68
3.2.3 Calcul de degré de compatibilité	69
3.3 Interprétation	69
Conclusion.....	70
CONCLUSION GENERALE ET PERSPECTIVES	72
Bilan	72
Perspectives	72
REFERENCES BIBLIOGRAPHIES	73

LISTE DE FIGURES

Figure 1 : Architecture par composant [12].	8
Figure 2 : Graphique de dégradation du logiciel en fonction de l'architecture [12]	8
Figure 3 : Assemblage d'éléments architecturaux pouvant être décrit à l'aide d'un ADL [5].	10
Figure 4 : Architecture orientée service [7]	14
Figure 5 : Comparaison application micro-service et monolithique [12]	18
Figure 6 : Implémentation du pattern SAGA, par chorégraphie [9]	34
Figure 7 : Implémentation du pattern SAGA, par orchestration [9]	35
Figure 8 : Passerelle API – un seul point d'entrée [9]	41
Figure 9 : Passerelle API - Point d'entrée selon le type de client [9]	42
Figure 10 : Découverte de service côté client [9]	43
Figure 11 : Découverte de service côté serveur [9]	44
Figure 12 : Modèle d'évaluation	50
Figure 13 : Représentation de l'espace de la fonction évaluation	51
Figure 14 : Architecture du framework Spring [1]	55
Figure 15 : Architecture d'un projet Spring Cloud [25]	57
Figure 16 : Modèle architectural MVC [19]	61
Figure 17 : Architecture d'application JAVA EE [19]	65

LISTE DES TABLEAUX

Tableau 1 : Evaluation de quelques applications micro-services de référence [13].	28
Tableau 2 : Critère d'évaluation	48
Tableau 3 : Tableau des valeurs de la fonction h, pour la technologie Spring Boot 2.2.2	57
Tableau 4 : Tableau des valeurs de la fonction h, pour la technologie JAVA EE 7	68

GLOSSAIRE

API : Application Programming Interface

UML: Unified Modeling Language

ADL: Architecture Description Languages

SOA: Service Oriented Architecture

UDDI: Universal Description Discovery and Integration

HTTP: Hypertext Transfert Protocol

SRP: Single Responsibility Principle

DDD : Domain-Driven Design

SGBD : Système de Gestion des Bases de Données

IP : Internet Protocol

JAVA EE : Java Enterprise Edition

INTRODUCTION GENERALE

Cadre du mémoire

L'augmentation de la puissance de calcul des ordinateurs a permis aux constructeurs de logiciel d'offrir de plus en plus de fonctionnalités aux utilisateurs, avec pour conséquence directe une augmentation de la taille, de la complexité des logiciels et dans plusieurs cas une réduction des facteurs de qualité des logiciels tels que : la maintenabilité, la réutilisabilité, l'efficacité, l'extensibilité, etc.

Au début, les problèmes de complexité ont été résolus par les développeurs en choisissant les bonnes structures de données, en développant des algorithmes, et en appliquant tacitement le concept de séparation des préoccupations. Une autre solution apportée depuis le milieu des années 1980, par la communauté scientifique et l'industrie du logiciel a été de concevoir et de réaliser les logiciels autour d'un squelette abstrait, on parle ainsi d'architecture logicielle.

Bien que le terme "architecture logicielle" est relativement nouveau pour l'industrie, les principes fondamentaux du domaine ont été appliqués sporadiquement par les pionniers de l'ingénierie du logiciel. L'objectif de l'architecture logicielle est d'offrir une vue d'ensemble et un fort niveau d'abstraction afin d'être en mesure d'appréhender un système logiciel. L'architecture propose une organisation grossière du système comme une collection de pièces logicielles. Dès lors, de plus en plus de systèmes sont passés d'une architecture dite organique (basée sur le génie du développeur présent) à des architectures cohérentes et structurées.

Dans la pratique logicielle au quotidien, des approches standardisées pour des problèmes architecturaux récurrents ont été développées, ce sont les styles architecturaux [1]. Ceux-ci constituent des solutions standardisées et réutilisables à des problèmes récurrents d'architecture, toutefois, un bon architecte doit choisir un style qui corresponde aux besoins du problème particulier à résoudre. C'est ainsi, qu'avec le développement d'internet et du cloud computing, on assiste à une conception de plus en plus distribuée des systèmes informatiques. Dans ce contexte, un style architectural s'est vu propulsé au-devant de la scène : il s'agit du style micro-service, qui descend du style SOA et qui permet de créer une application sous la forme d'une suite de services de granularité fine, chacun s'exécutant dans son propre processus et pouvant être déployés de manière indépendante.

De nombreux langages et technologies se sont développés pour permettre la mise en œuvre de ce style architectural. Dès lors, on peut s'interroger sur le niveau de compatibilité de ces technologies avec les spécificités de ce style : Est-ce que ces technologies permettent de conserver les principaux atouts du style ? Est-ce qu'elles permettent d'atténuer les problèmes sous-jacents à cette organisation du logiciel ?

Problématique

Tout logiciel est doté d'une architecture, qui est le résultat d'une série de choix effectuée à toutes les phases du cycle de développement logiciel. Celle-ci pouvant être implicite (architecture dite organique) ou explicite. L'intérêt du développement centré sur l'architecture est de décrire explicitement l'architecture. Cela requiert de voir l'architecture logicielle comme une phase du cycle de développement logiciel à part entière [6].

Dans cette perspective, de plus en plus de sociétés choisissent le style architectural micro-services pour développer leurs systèmes, car celui-ci est supposé apporter plus de liberté aux développeurs et d'indépendance dans la maintenance et le déploiement des services; d'élargir la pile technologique de l'entreprise; de simplifier la compréhension et la prise en main d'un service; d'apporter une plus grande isolation des pannes; d'apporter une meilleure organisation du code, cette fois autour des capacités commerciales. Toutefois, ce style pose un certain nombre de problèmes. Au niveau conceptuel les problèmes majeurs sont ceux de la décomposition du système en services et de la granularité de chaque service. En effet, comment doit se faire la décomposition ? Quelle doit être la taille d'un service ? Au niveau purement technique nous avons comme problème : le niveau de couplage entre les services, le problème de doubles emplois, la réalisation des tests, la gestion de la communication entre services, l'augmentation du temps d'exécution dû à la latence du réseau, l'augmentation de la consommation des ressources matérielles, etc.

En connaissant les atouts et les limites de ce style architectural, la responsabilité est donnée à l'équipe de développement, dans la phase trois du cycle de développement logiciel, d'effectuer les choix technologiques adéquats pour que l'implémentation soit la plus conforme possible à la conception. Cela nécessite d'être capable de vérifier qu'une technologie conserve les points forts du style, qu'elle apporte des solutions optimales aux problèmes sous-jacents au style et qu'elle respecte les standards de développement de ce style.

La réflexion que nous menons dans ce travail s'inscrit dans cette même problématique, qui est celle de savoir comment évaluer l'apport d'une technologie pour l'implémentation d'une architecture orientée micro-service.

Objectif

L'objectif de ce travail de mémoire est de proposer un modèle pour l'évaluation des plateformes devant servir à l'implémentation des logiciels conçus suivant le style micro-service. Cet objectif global, se décline en trois objectifs spécifiques :

- + Ressortir les atouts, les points faibles et les standards de conception et d'implémentation du style micro-service ;
- + Construire le modèle d'évaluation sous forme d'une grille de critères à satisfaire ;
- + Apprécier l'utilisation des technologies Spring Boot et JAVA EE pour l'implémentation de ce style architectural ;

Organisation du travail

Le corps de ce manuscrit est organisé comme suit :

- + Le chapitre 1 : propose un état du domaine de l'architecture logicielle mais surtout du style architectural micro-service, notamment dans ses aspects nécessaires à notre travail. Le chapitre débute par une brève présentation des notions d'architecture logicielle (rétrospective et définition) et de style architectural. Il se poursuit avec les travaux menés dans le domaine du style micro-service notamment les tentatives de définition de celui-ci, son intérêt, ses problèmes et les standards pour son implémentation. Pour finir, nous faisons une présentation d'une méthode d'évaluation des applications développées dans le style micro-service.
- + Le chapitre 2 : dans ce chapitre nous abordons la construction du modèle d'évaluation de la compatibilité des technologies avec le style. Ce qui consiste à ressortir une série d'exigences, qui serviront de critères d'évaluation pour toute technologie candidate. Pour cela, nous nous appuyerons sur les atouts, les problèmes et principalement sur les standards (design patterns) qui ont émergé de la communauté du génie logiciel et qui sont spécifiques à ce style architectural. La deuxième étape sera de construire la fonction d'évaluation. Celle-ci prend en paramètre une technologie et les exigences de l'application développée pour attribuer une note ou niveau de compatibilité.

- ✚ Le chapitre 3 : dans ce chapitre nous appliquons le modèle d'évaluation proposé aux technologies Spring Boot et JAVA EE afin d'en déterminer le niveau de compatibilité avec le style micro-service.

CHAPITRE 1 : G V C V " F G " N \emptyset C T V

Introduction

Dans ce chapitre dédié à l'état de l'art, nous faisons une revue de la littérature dans les domaines essentiels pour notre travail. Nous commençons par présenter les concepts fondamentaux liés à la notion d'architecture logicielle : historique, définition, représentation et styles architecturaux. En seconde partie nous présentons les travaux menés dans le domaine du style micro-service notamment les tentatives de définition de celui-ci, son intérêt, ses problèmes et les standards pour son implémentation. Pour finir, nous faisons une présentation d'une méthode d'évaluation des applications développées dans le style micro-service.

1.1 Architectures logicielles

1.1.1 Historique et évolution

L'origine de la notion d'architecture remonte à la fin des années 60 et s'est enrichie au fur à mesure par la création de nouveaux concepts. Ainsi la période 60 à 70 a vu l'invention des notions de programmation structurée (qui représente un programme informatique comme une suite d'étapes formant un organigramme) ; C'est dans les années 70 à 80 qu'ont été élaborés les grands principes architecturaux modernes et les principaux styles architecturaux que sont [2] :

- **L'architecture système** qui décrit les relations et interactions de l'ensemble des composants logiciels ;
- **L'architecture détaillée** décrivant l'architecture interne de chacun des composants ;
- **L'architecture statique** représentant les interrelations temporellement invariables (dépendances fonctionnelles, flux de contrôle, flux de données) ;
- **L'architecture dynamique** : décrivant les interactions et l'évolution du système dans le temps ;
- **Les styles architecturaux** : architecture en appels et retours (hiérarchie de contrôle), architecture centrée sur les données, architecture en flot de données, architecture en couches, architecture orientée objets, architecture orientée service.

La décennie 80 - 90 s'est essentiellement concentrée sur le développement de l'architecture orientée objet, introduisant trois nouveaux types de composants logiciels : l'objet, la classe et la méta-classe ; ainsi que des relations ontologiques entre ces composants.

La décennie 1990 - 2000 fut marquée par l'apparition en 1995, du langage UML qui devint en 2007 une norme internationale (ISO/IEC 15930) utilisée pour la représentation de l'architecture logicielle.

Toutefois ce n'est qu'en été 2006 avec le numéro spécial de l'IEEE Software magazine dédié au domaine de l'architecture logicielle, que celle-ci est reconnue comme une discipline majeure du génie logiciel et a été intégrée dans le cycle de développement des logiciels.

1.1.2 Définition

Le terme d'architecture logicielle est sans aucun doute l'un des plus utilisé actuellement dans le génie logiciel mais aussi l'un des termes dont la définition ne fait pas consensus. En effet, une étude du SEI (Software Engineering Institute) a documenté et catalogué des centaines de définitions. Au moins deux raisons expliquent cela. La première est l'analogie qui est fait avec la construction physique de bâtiments, ce qui tend à transposer certaines hypothèses et concepts vers le logiciel. La deuxième raison qui rend difficile le consensus est que l'architecture logicielle n'est reconnue comme un produit à part entière du cycle de développement logiciel que depuis quelques années.

Les architectures logicielles sont considérées comme une sous-discipline du génie logiciel. Une architecture est considérée comme l'organisation nécessaire d'un système caractérisé par ses composants, leurs relations et avec l'environnement, et les principes qui guident leur conception et évolution. Autrement dit une architecture logicielle est une représentation abstraite d'un système exprimée essentiellement à l'aide de composants logiciels en interaction via des connecteurs. Elles forment la colonne vertébrale pour construire des logiciels complexes et de grande taille.

Partant, deux éléments principaux composent une architecture logicielle :

- Les composants : un composant logiciel est un élément constitutif d'un logiciel destiné à être incorporé en tant que pièce détachée dans des applications. Les paquets, les bibliothèques logicielles, les exécutable, les fichiers, les bases de données ou encore des éléments de configuration (paramètres, scripts, fichiers de commandes) sont des composants logiciels. [12].
- Les connecteurs : Ce sont des objets du premier ordre qui assurent les interactions entre composants. Ils peuvent être de complexité variable, du simple appel de méthode à l'ordonnanceur. Ils permettent la flexibilité et l'évolution de l'ensemble du système.



Figure 1 : Architecture par composant [12].

Une architecture faible ou absente peut entraîner de graves problèmes à toutes les phases du projet : dans la définition du plan de travail, dans la répartition du travail entre les équipes, dans l'allocation des ressources, dans le choix des technologies, dans la maintenance, etc. En effet, toute modification d'un logiciel mal architecturé peut déstabiliser la structure de celui-ci et entraîner, à la longue, une dégradation.

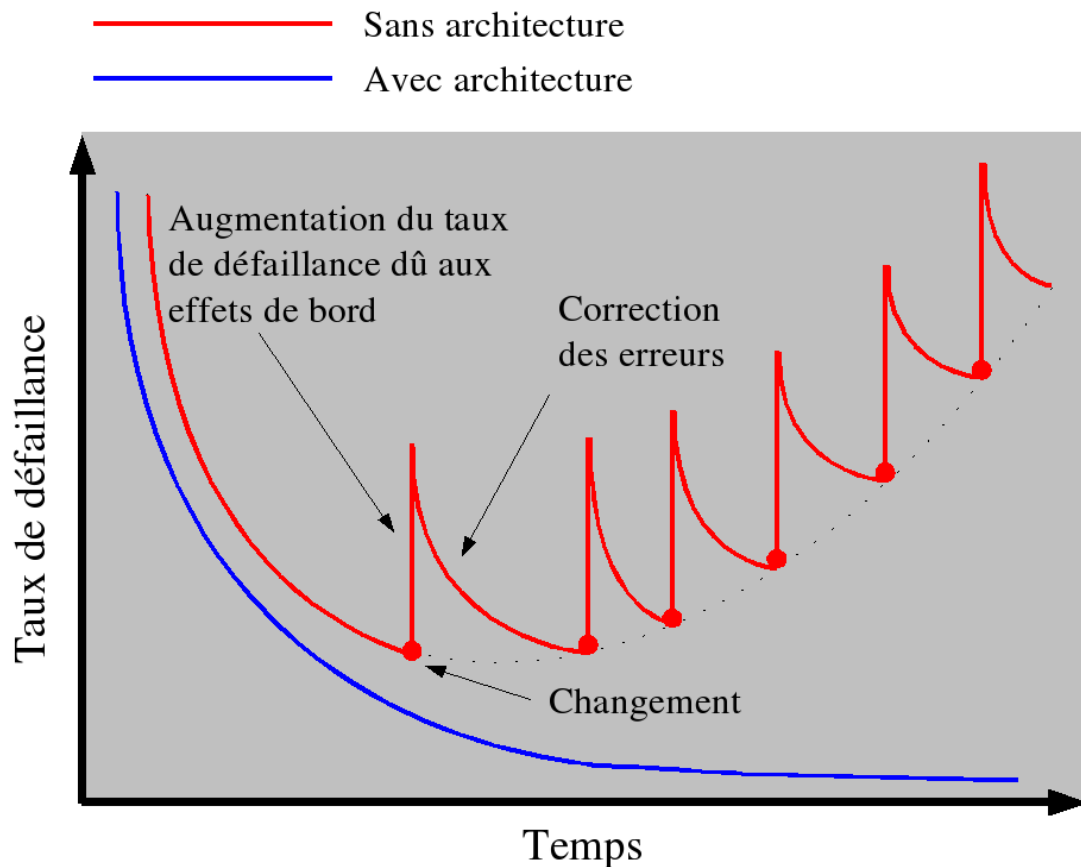


Figure 2 : Graphique de dégradation du logiciel en fonction de l'architecture [12]

De plus, avec l'adoption des méthodes de développement agile pour la gestion des projets logiciels comme SCRUM ou XP (Extreme Programming), la gestion des changements est primordiale. En effet, il est implicitement considéré que les besoins des utilisateurs du système peuvent changer et que l'environnement du système peut changer.

Dans les années 1990 il y a eu un effort commun pour tenter de définir et de codifier les aspects fondamentaux de la discipline. Des ensembles préliminaires de patrons de conception, de styles architecturaux, de meilleures pratiques, de langages de description et de logiques formelles ont été élaborés au cours de cette période.

A présent il n'existe pas un consensus sur une définition des architectures logicielles. Dans la littérature on pourrait retrouver quelques dizaines de définitions. Devant une telle perspective nous listerons par la suite quelques définitions tirées de la littérature avec le but de donner une idée plus claire des principaux intérêts portés par cette discipline.

Une définition assez récente est celle de Kruchten et al. (2006) : « L'architecture logicielle implique la structure et l'organisation par lesquelles des composants interagissent pour créer de nouvelles applications, possédant la propriété de pouvoir être les mieux conçues et analysées au niveau système » [3].

Dans le cadre de ce travail nous retiendrons la définition proposée par Garlan et Shaw (1993) : «un ensemble de composants de calcul - ou tout simplement de composants - accompagné d'une description des interactions entre ces composants - les connecteurs» [4]. Pour eux le niveau de conception des architectures logicielles va au-delà des algorithmes et des structures de données : « la conception et la spécification de la structure globale d'un système représente un nouveau type de problème ». Les tâches comprennent :

1. L'organisation et le contrôle structural ;
2. La définition de protocoles de communication, de synchronisation, et d'accès aux données ;
3. L'affectation des fonctionnalités aux éléments de conception ;
4. La distribution physique ;
5. La composition des éléments de conception ;
6. La mise à l'échelle et la performance ;
7. La sélection entre différentes alternatives de conception.

1.1.3 Langage de description des architectures (ADL)

Au courant des années 90, de nombreux universitaires se sont investis à établir des formalismes pour la représentation des architectures logicielles ce qui a donné naissance aux ADL ou langages de description d'architecture. Parmi les plus représentatifs correspondant à une première génération nous trouvons notamment : Darwin Magee et al. (1995) ; Magee et Kramer (1996), Rapide Luckham et al. (1995), Wright Allen (1997) ; Allen et Garlan (1997). L'étude de la première génération d'ADL a montré que, bien que différents, ces ADLs partagent une même base conceptuelle commune d'où l'idée de créer un ADL générique, qui combinerait les propriétés communes et les caractéristiques bénéfiques de la première génération d'ADL. On peut considérer comme faisant partie de cette deuxième génération les ADLs : xADL

Dashofy et al. (2001) et π -ADL Ouando (2004) et en acceptant l'idée de considérer UML 2.0 comme un ADL à part entière, celui-ci s'inscrit dans cette seconde génération.

Qu'il s'agisse de la première ou de la deuxième génération, les ADLs sont utilisés pour décrire la structure d'un système sous la forme d'un assemblage de briques logicielles. Cela est illustré par le diagramme de la Figure 3 où l'on voit un client et un serveur reliés par un appel de procédure à distance (Remote Procedure Call).

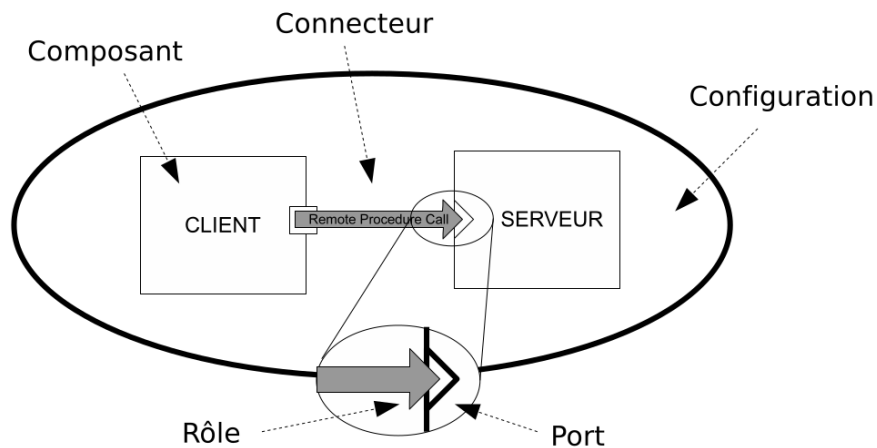


Figure 3 : Assemblage d'éléments architecturaux pouvant être décrit à l'aide d'un ADL [5].

Un ADL est, selon Taylor et Medvidovic [6], un langage qui offre des fonctionnalités pour la définition explicite et la modélisation de l'architecture conceptuelle d'un système logiciel, comprenant au minimum : des composants, des connecteurs, et des configurations architecturales :

1. **Le composant** : est une unité de calcul ou de stockage. Il peut être simple ou composite, et sa fonctionnalité peut aller de la simple procédure à une application complète. Le composant est considéré comme un couple spécification-code : la spécification donne les interfaces, les propriétés du composant ; le code correspond à la mise en œuvre de la spécification par le composant.
2. **Le connecteur** : modélise un ensemble d'interactions entre composants. Cette interaction peut aller du simple appel de procédure distante aux protocoles de communication. Tout comme le composant, le connecteur est un couple spécification-code : la spécification décrit les rôles des participants à une interaction ; le code correspond à l'implantation du connecteur. Cependant, la différence avec le composant est que le connecteur ne correspond pas à une unique, mais éventuellement à plusieurs unités de programmation.

3. **La configuration** : définit les propriétés topologiques de l'application : les connections entre composants et connecteurs, mais aussi, selon les ADL, les propriétés de concurrence, de répartition, de sécurité, etc. La topologie peut être dynamique, auquel cas la configuration décrit la topologie ainsi que son évolution.

1.1.4 Styles architecturaux

Un style architectural permet de décrire un ensemble de propriétés communes à une famille de systèmes comme, par exemple, les systèmes temps-réel ou les applications orientées service. Chaque style possède des attributs de qualité déterminés (performance, disponibilité, sécurité, réutilisabilité, etc.).

Un style architectural est un moyen générique qui aide à l'expression des solutions structurelles des systèmes. Il comprend un vocabulaire d'éléments conceptuels (les composants et les connecteurs), impose des règles de configuration entre les éléments du vocabulaire (ensemble de contraintes) et véhicule une sémantique qui donne un sens (non ambiguë) à la description structurelle.

Les styles architecturaux constituent des solutions standardisées et réutilisables à des problèmes récurrents d'architecture, toutefois, un bon architecte doit choisir un style qui corresponde aux besoins du problème particulier à résoudre, ce qui exige une compréhension du domaine du problème ainsi qu'une prise de conscience de la variété des styles architecturaux et des préoccupations spécifiques qu'ils adressent [6]. Dans bien des cas, l'architecture final d'un système est la combinaison de plusieurs styles d'architecture, selon deux principaux modes de mise en relation :

- **La spécialisation** : Un style peut être un sous-style d'un autre en renforçant les contraintes, ou en fournissant des versions plus spécialisées de quelques-uns des types d'éléments [4]. Par exemple : un style client-serveur N-tiers pourrait spécialiser le style plus général client-serveur en limitant les interactions entre les tiers non-adjacents.
- **La composition**, que Garlan et Shaw décrivent sous le terme d'architectures hétérogènes [4]. On peut combiner deux styles en prenant l'union de leur vocabulaire de conception, et en joignant leurs contraintes.

Les principaux styles architecturaux rencontrés dans la littérature et dans l'industrie sont :

Modèle Vue Contrôleur / Model-View-Controller (MVC) :

Est un style qui répartie l'application en 3 composants :

- Modèle : Représente les données et les fonctionnalités du domaine de l'application ;
- Vues : Représentations visuelles des données qui forment l'application ;
- Contrôleurs : Composantes de contrôle qui traitent les entrées (événements) en provenance des vues et du système et les traduit en opérations effectuées sur le modèle et les vues.

Architecture en couche :

Organise l'application en couches de composantes offrant des groupes de services. Elle limite l'échange de messages aux composantes de la couche adjacente et impose que les composants d'une couche ne font appel qu'aux composants de la même couche ou de la couche inférieure.

Architecture orientée objets :

Les composants du système (objets) intègrent des données et les opérations de traitement de ces données. La communication et la coordination entre les objets sont réalisées par un mécanisme de passage de messages. L'architecture orientée objets est souvent décrite par les trois piliers : encapsulation, héritage et polymorphisme.

Architecture orientée service (SOA) :

Les architectures orientées service (SOA) sont des modèles qui définissent un système par un ensemble de services logiciels distribués, qui fonctionnent indépendamment les uns des autres afin de réaliser une fonctionnalité globale [7]. Le choix d'une architecture SOA entre dans la perspective de transformer le Web en une énorme plate-forme de composants faiblement couplés et automatiquement intégrables.

SOA est une approche architecturale permettant la création des systèmes basés sur une collection de services développés dans différents langages de programmation, hébergés sur différentes plates-formes avec divers modèles de sécurité et processus métier. Chaque service représente une unité autonome de traitements et de gestion de données, communiquant avec son environnement à l'aide de messages. Les échanges de messages sont organisés sous forme de contrats d'échange.

Ce terme est apparu au cours de la période 2000-2001 et concernait à l'origine essentiellement les problématiques d'interopérabilité syntaxique en relation avec les technologies d'informatique utilisées en entreprise. Cette conception a évolué pour désigner maintenant le sous-ensemble particulier d'architecture de médiation en fonction de la technologie disponible.

L'une des technologies la plus utilisée pour implanter ce type d'architectures est les services Web. La particularité de cette nouvelle technologie réside dans le fait qu'elle utilise la technologie Internet comme infrastructure pour la communication entre les composants logiciels et prend le pari d'employer des standards généralistes, fortement répandus et peu coûteux tels que XML ou HTTP.

Souvent, un service Web est vu comme une application accessible pour d'autres applications à travers le Web. C'est une définition tellement large dans ce que l'on peut dire que n'importe quel objet ayant un URL est un service Web. Par exemple, un programme accessible sur le Web avec une API. Une définition plus précise est proposée par le consortium UDDI qui caractérise les services Web :

« Un service Web est une application métier modulaire qui possède des interfaces orientées Internet basées sur des standards » [7].

Cette définition est plus détaillée et elle nécessite que le service soit ouvert, pour qu'il soit invoqué à travers Internet. Malgré cette classification, la définition reste imprécise. Par exemple, on ne comprend pas bien le sens d'une application métier modulaire. Une autre définition est proposée par le World Wide Web Consortium (W3C)

« Un service Web est un système logiciel identifié par un identificateur uniforme de ressources (URI), dont les interfaces publiques et les liens (binding) sont définis et décrits en XML. Sa définition peut être découverte dynamiquement par d'autres systèmes logiciels. Ces autres systèmes peuvent ensuite interagir avec le service Web d'une façon décrite par sa définition, en utilisant des messages XML transportés par des protocoles Internet ».

Les services Web possèdent trois acteurs principaux :

- ✚ **Le fournisseur de service (service provider)** : Il définit le service à publier et la description du service dans l'annuaire ;
- ✚ **L'annuaire (service broker)** : Il reçoit et enregistre les descriptions des services publiés par les fournisseurs, d'autre part il reçoit et répond aux recherches de services lancées par les clients ;
- ✚ **Le demandeur (service requestor)** : obtient la description du service grâce à l'annuaire utilisé par le service et appelle les services demandés.

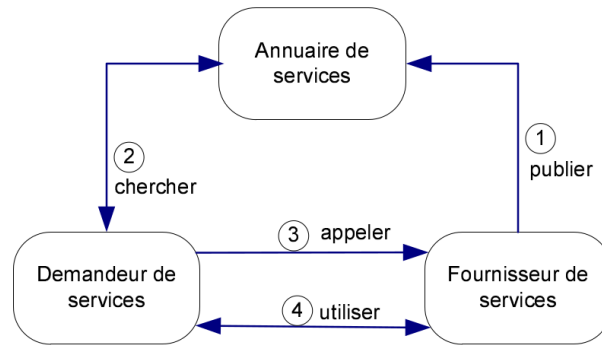


Figure 4 : Architecture orientée service [7]

1.1.5 Bilan

L'architecture logicielle permet de faire abstraction d'un système complexe, ce qui offre les avantages suivants :

- Elle fournit une base pour l'analyse du comportement des logiciels avant sa construction : La possibilité de vérifier qu'un futur logiciel répond aux besoins de ses parties prenantes sans avoir à le construire réellement. Ce qui représente une réduction substantielle des coûts et des risques.
- Elle fournit une base pour la réutilisation d'éléments et de connaissances. Une architecture logicielle complète ou une partie de celle-ci, peut être réutilisée sur plusieurs systèmes dont les parties prenantes exigent des attributs de qualité ou des fonctionnalités similaires, ce qui permet de réduire les coûts de conception et d'atténuer les risques liés à la conception.
- Elle facilite la communication avec les parties prenantes et contribue à l'obtention d'un produit correspondant à leurs besoins. L'architecture donne la possibilité de communiquer sur les décisions de conception avant la mise en œuvre du système, alors qu'elles sont encore relativement faciles à adapter.
- Elle aide dans la gestion des risques. L'architecture logicielle aide à réduire les risques et les risques d'échec.
- Elle permet de réduire les coûts. L'architecture logicielle est un moyen de gérer les risques et les coûts dans des projets informatiques complexes.

En somme, une architecture logicielle a deux objectifs principaux que sont la réduction des coûts et l'augmentation de la qualité du logiciel ; la réduction des coûts est essentiellement obtenue par la réutilisation de composants logiciels, la capitalisation des connaissances et par la diminution du temps de maintenance (correction d'erreurs et adaptation du logiciel). La qualité, quant à elle, se trouve distribuée à travers plusieurs critères dont : la maintenabilité, la

réutilisabilité, l'efficacité, l'extensibilité, etc. ; la norme ISO 9126 est un exemple d'un tel ensemble de critères.

1.2 L'architecture micro-service

1.2.1 Émergence du style micro-service

Comme tous les styles architecturaux, le style micro-service est une solution à un problème de structuration des applications auquel a été confronté l'industrie du logiciel. Pour ce style le problème rencontré est le suivant :

Soit une application serveur développée par une entreprise, qui doit prendre en charge divers clients, notamment les navigateurs de bureau, les navigateurs mobiles et les applications mobiles natives. L'application peut également exposer une API à utiliser par des tiers. Il peut également s'intégrer à d'autres applications via des services Web ou un courtier de messages (message broker). L'application doit recevoir et traiter les requêtes (requêtes HTTP et messages) en exécutant une logique métier ; accéder à une base de données ; échanger des messages avec d'autres systèmes ; et renvoyer une réponse dans différents formats de données notamment HTML / JSON / XML. De plus l'entreprise dispose de composants logiciels correspondant aux différents domaines fonctionnels de l'application.

À ces exigences fonctionnelles, sont venues s'ajouter des contraintes non-fonctionnelles d'ordre managériale, de rentabilité et d'efficacité telles que :

- Une seule équipe de développeurs travaillent sur l'application ;
- Les nouveaux membres de l'équipe doivent rapidement devenir productifs ;
- L'application doit être facile à comprendre et à modifier.
- L'utilisation des techniques de déploiement continu ;
- Plusieurs instances de l'application doivent s'exécuter au même moment sur plusieurs machines afin de satisfaire aux exigences de monter en charge et de disponibilité ;
- L'application doit pouvoir tirer parti des technologies émergentes (Framework, langages de programmation, etc.).

Dès lors la question qui se pose est : Quelle sera la meilleure architecture de déploiement de l'application ?

L'approche traditionnelle de développement est celle dite monolithique. Qui consiste à développer toute l'application comme un seul gros bloc logiciel dans lequel les fonctionnalités et les technologies utilisées sont fortement liées et difficilement modifiables. En interne, cette

application peut avoir plusieurs services, composants, etc. Mais elle est déployée en tant que solution unifiée, ce qui est l'un de ses principaux atouts. En outre, ce style correspond parfaitement aux applications de petite taille mais atteint très vite ses limites lorsque l'application devienne volumineuse. Ce style de développement a comme inconvénients :

- De rendre l'application difficile à comprendre et à modifier. C'est particulièrement vrai lorsque le volume de code devient énorme ;
- Il est difficile d'ajouter de nouveaux développeurs ou de remplacer les membres de l'équipe sortante ;
- Une baisse de la productivité, car le code source est désormais trop volumineux ;
- Empêche les développeurs de travailler de manière indépendante. Toute l'équipe doit coordonner tous les efforts de développement et de redéploiement.
- Le développement continu est rendu très difficile, car pour mettre à jour un petit composant, c'est l'ensemble de l'application qui doit être redéployé ;
- Il est très difficile (voir impossible) de changer la pile technologique de départ ;
- La montée en charge de l'application peut également s'avérer difficile. En effet, une architecture monolithique ne peut évoluer que dans une seule dimension (par la création de plusieurs copies de l'application et l'utilisateur d'un répartiteur de charge). Toutefois cela pose des problèmes notamment de gestion du cache, l'augmentation de la consommation de mémoire et le trafic d'entrée / sortie.

Le style micro-service, a donc été conçu comme solution candidate pour résoudre ce problème et il s'oppose en tout point à une application monolithique. Il propose une architecture qui structure l'application en tant qu'ensemble de services collaboratifs faiblement couplés et de petite taille (Cette approche correspond à l'axe des Y du Scale Cube). Chaque service :

- Doit être facilement maintenable et testable ;
- Doit pouvoir être développé et déployé rapidement et fréquemment ;
- Doit être faiblement couplé à d'autres services ;
- Doit être développé par une équipe de manière indépendante sur son (ses) service (s) sans être affectés par des modifications apportées à d'autres services et sans affecter d'autres services ;
- Déployer indépendamment des autres (une équipe doit pouvoir déployer son service sans avoir à se coordonner avec d'autres équipes) ;
- Doit être développé par une petite équipe (essentiel pour une productivité élevée).

1.2.2 Définitions notables

Un **micro-service** est un service léger et indépendant qui exécute des fonctions uniques et collabore avec d'autres services similaires à l'aide d'une interface bien définie [8].

Une **architecture de micro-service** est un style architectural qui structure une application en tant que collection de services : hautement maintenable et testable ; couplage lâche ; déployable indépendamment ; organisé autour des capacités de l'entreprise ; développé par une petite équipe [9].

Une autre définition du style architectural micro-service est celle donnée par James Lewis et Martin Fowler [10]. Pour eux il s'agit d'une approche permettant de développer une application unique sous la forme d'une suite de petits services, chacun s'exécutant dans son propre processus et communiquant avec des mécanismes légers, souvent une API de ressources HTTP. Ces services sont construits autour de capacités métier et peuvent être déployés indépendamment sur des hôtes. Il existe un minimum de gestion centralisée de ces services, qui peuvent être écrits dans différents langages de programmation et utiliser différentes technologies de stockage de données.

1.2.3 Caractéristiques du style micro-service

Les micro-services présente de nombreux avantages pour les équipes Agile : comme le souligne Martin Fowler, Netflix, eBay, Amazon, Twitter, PayPal et d'autres géant d'internet qui ont tous évolué d'une architecture monolithique à une architecture de micro-services [11]. Contrairement aux micro-services, une application monolithique est conçue comme une seule unité autonome. Cela ralentit les modifications apportées à l'application car cela affecte l'ensemble du système. Une modification apportée à une petite section de code peut nécessiter la création et le déploiement d'une version entièrement nouvelle du logiciel. La mise à l'échelle de fonctions spécifiques d'une application signifie également que vous devez mettre à l'échelle l'ensemble de l'application.

Les micro-services résolvent ces problèmes des systèmes monolithiques en étant aussi modulaires que possible. Sous leur forme la plus simple, ils permettent de créer une application sous la forme d'une suite de petits services, chacun s'exécutant dans son propre processus et pouvant être déployés de manière indépendante. Ces services peuvent être écrits dans différentes langues et utiliser différentes techniques de stockage de données. Bien que cela aboutisse à la mise au point de systèmes évolutifs et flexibles.

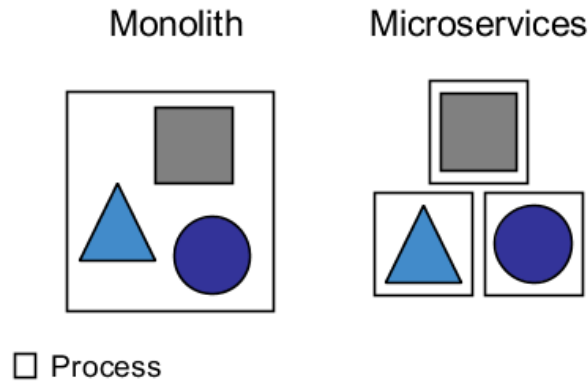


Figure 5 : Comparaison application micro-service et monolithique [12]

Pour qu'une architecture soit dite, micro-service elle doit respecter au moins 6 caractéristiques suivantes [12] :

1.2.3.1 Composants multiples

Les logiciels conçus sous forme de micro-services peuvent, par définition, être décomposés en plusieurs services. Pour que chacun de ces services puisse être déployé, peaufiné puis redéployé indépendamment, sans compromettre l'intégrité d'une application. Par conséquent, il n'est souvent nécessaire que de modifier un ou plusieurs services distincts au lieu de devoir redéployer des applications entières. Cependant, cette approche présente des inconvénients, notamment des appels distants coûteux (au lieu d'appels en cours de processus), des API distantes plus grossières et une complexité accrue lors de la redistribution des responsabilités entre les composants.

1.2.3.2 Orienté fonctionnalité

Le style des micro-services est généralement organisé en fonction des capacités et des priorités de l'entreprise. Contrairement à une approche de développement monolithique traditionnelle, dans laquelle différentes équipes est spécialisée soit dans la mise en œuvre des interfaces utilisateurs, soit des bases de données, soit des couches technologiques ou la logique côté serveur, l'architecture de micro-service utilise des équipes inter-fonctionnelles. Les responsabilités de chaque équipe sont de créer des produits spécifiques basés sur un ou plusieurs services individuels communiquant via un bus de messages. Dans les micro-services, une équipe possède le produit pendant toute sa durée de vie, comme le dit souvent la maxime d'Amazon : « Vous le construisez, vous l'exécutez ».

1.2.3.3 Routage simple

Les micro-services fonctionnent un peu comme le système UNIX classique : ils reçoivent des requêtes, les traitent et génèrent une réponse en conséquence. Ceci est opposé au nombre d'autres produits tels que les ESB (Enterprise Service Buses), qui utilisent des systèmes de haute technologie pour le routage de messages, l'orchestration et l'application de règles métiers. On peut dire que les micro-services ont des points de terminaison intelligents qui traitent les informations et appliquent la logique, ainsi que des canaux muets à travers lesquels les informations circulent.

1.2.3.4 Décentralisation

Comme les micro-services impliquent diverses technologies et plateformes, les méthodes traditionnelles de gouvernance centralisée ne sont pas optimales. La communauté privilégie la gouvernance décentralisée, car ses développeurs s'efforcent de produire des outils utiles pouvant ensuite être utilisés par d'autres personnes pour résoudre les mêmes problèmes. Tout comme la gouvernance décentralisée, l'architecture micro-service favorise également la gestion décentralisée des données. Les systèmes monolithiques utilisent une base de données logique unique pour différentes applications. Dans une architecture micro-service, chaque service gère généralement sa base de données unique.

1.2.3.5 Résistance aux pannes

Comme un enfant bien équilibré, les micro-services sont conçus pour faire face aux échecs. Étant donné que plusieurs services uniques et variés communiquent ensemble, il est tout à fait possible qu'un service échoue pour une raison ou une autre (par exemple, lorsque le fournisseur n'est pas disponible). Dans ces cas, le client doit permettre aux services voisins de fonctionner tout en se retirant de la manière la plus gracieuse possible. Cependant, la surveillance de micro-services peut aider à prévenir le risque d'échec. Pour des raisons évidentes, cette exigence ajoute davantage de complexité aux micro-services par rapport à une architecture de système monolithique.

1.2.3.6 Évolutivité

L'architecture de micro-services est une conception évolutive et, encore une fois, elle est idéale pour les systèmes évolutifs où il est difficile d'anticiper pleinement les types de périphériques pouvant accéder à votre application un jour.

1.2.4 Intérêts du style micro-service

Les architectures micro-service offrent de nombreux avantages, faisant de lui l'un des styles les plus utilisés actuellement :

- ✚ L'architecture micro-service offre aux développeurs la liberté de développer et de déployer de manière indépendante ;
- ✚ Un micro-service peut être développé par une équipe assez petite ;
- ✚ Le code pour différents services peut être écrit dans différents langages (bien que de nombreux praticiens le déconseille) ;
- ✚ Intégration facile et déploiement automatique (utilisation d'outils d'intégration continue open source tels que Jenkins, Hudson, etc.) ;
- ✚ Facile à comprendre et à modifier pour les développeurs, ce qui permet qu'un nouveau membre de l'équipe soit rapidement productif ;
- ✚ Les développeurs peuvent utiliser les dernières technologies ;
- ✚ Le code est organisé autour des capacités commerciales ;
- ✚ Lorsqu'une modification est requise dans une certaine partie de l'application, seul le service concerné peut être modifié et redéployé : il n'est pas nécessaire de modifier et de redéployer l'application entière ;
- ✚ Meilleure isolation des pannes : si un micro-service échoue, l'autre continuera à fonctionner ;
- ✚ Facile à dimensionner et à intégrer à des services tiers ;
- ✚ Pas d'engagement à long terme envers la pile de technologie.

1.2.5 Problèmes de conception et d'implémentation

La décomposition d'une application monolithique, en micro-service est source d'un certain nombre de problème :

- ✚ En raison du déploiement distribué, les tests peuvent devenir compliqués et fastidieux ;
- ✚ L'augmentation du nombre de services peut entraîner des barrières d'information ;
- ✚ L'architecture apporte une complexité supplémentaire car les développeurs doivent accroître la tolérance aux pannes, la latence du réseau et traiter différents formats de messages ainsi que l'équilibrage de charge ;
- ✚ Étant un système distribué, cela peut entraîner des doubles emplois ;
- ✚ Lorsque le nombre de services augmente, l'intégration et la gestion de produits entiers peuvent devenir compliquées ;

- ✚ En plus de plusieurs complexités de l'architecture monolithique, les développeurs doivent faire face à la complexité supplémentaire d'un système distribué ;
- ✚ Les développeurs doivent mettre des efforts supplémentaires dans la mise en œuvre du mécanisme de communication entre les services ;
- ✚ La gestion des cas d'utilisation qui couvrent plus d'un service sans utiliser de transactions distribuées n'est pas seulement difficile, mais nécessite également une communication et une coopération entre différentes équipes ;
- ✚ L'organisation en micro-service entraîne généralement une consommation de mémoire accrue ;
- ✚ Le fractionnement de l'application dans les micro-services est une opération hautement complexe.

La conception d'une application basée sur une architecture micro-service, nécessite de se poser au préalable un ensemble de question, visant à évaluer, si ce style architectural est adapté au problème posé.

1. Quand utiliser l'architecture de micro-service ?

L'un des défis de cette approche consiste à décider quand il est judicieux de l'utiliser. Lors du développement de la première version d'une application, les problèmes que cette approche résout ne se font souvent pas ressentir. De plus, l'utilisation d'une architecture distribuée élaborée ralentira le développement. Cela peut constituer un problème majeur pour les startups dont le principal défi consiste souvent à faire évoluer rapidement le modèle commercial et les applications qui les accompagnent. Cependant, plus tard, lorsque le défi consiste à savoir comment faire évoluer votre système et que vous devez utiliser une décomposition fonctionnelle, les dépendances enchevêtrées peuvent rendre difficile la décomposition de votre application monolithique en un ensemble de services.

2. Comment décomposer l'application en services ?

Un autre défi consiste à décider comment partitionner le système en micro-services. C'est un art, mais plusieurs stratégies peuvent aider :

- Décomposer par capacité métier et définir les services correspondant aux capacités métier ;
- Décomposer par sous-domaine de conception piloté par le domaine ;

- Décomposer par verbe ou cas d'utilisation et définissez les services responsables d'actions particulières. Par exemple, un service d'expédition responsable de l'expédition des commandes complètes ;
- Décomposez par noms ou ressources en définissant un service responsable de toutes les opérations sur les entités / ressources d'un type donné. Par exemple, un service de compte responsable de la gestion des comptes d'utilisateurs.

Idéalement, chaque service ne devrait avoir qu'un petit ensemble de responsabilités semblable au principe de responsabilité unique (SRP) utilisé pour les classes. Le SRP définit la responsabilité d'une classe comme une raison de changer et indique qu'une classe ne devrait avoir qu'une seule raison de changer.

Une autre analogie qui aide à la conception des services est la conception des utilitaires Unix. Unix fournit un grand nombre d'utilitaires tels que `grep`, `cat` et `find`. Chaque utilitaire fait exactement une chose, souvent exceptionnellement bien, et peut être combiné avec d'autres utilitaires à l'aide d'un script shell pour effectuer des tâches complexes.

3. Comment maintenir la cohérence des données ?


Afin de garantir un couplage lâche, chaque service dispose de sa propre base de données. Le maintien de la cohérence des données entre les services constitue un défi, car de nombreuses applications à validation de phase / distribuées ne sont pas une option. Une application doit plutôt utiliser le modèle SAGA. Un service publie un événement lorsque ses données changent. D'autres services utilisent cet événement et mettent à jour leurs données. Il existe plusieurs moyens de mettre à jour de manière fiable les données et de publier des événements, notamment la détermination de l'événement et la rédaction du journal des transactions.

4. Comment implémenter des requêtes ?

Un autre défi consiste à mettre en œuvre des requêtes qui doivent extraire des données appartenant à plusieurs services.

1.2.6 Patrons de conception

Pour aborder les problèmes posés par les applications, de nombreuses solutions standardisées ont été développées et organisées sous forme de patrons de conception :

 Patron de décomposition :

- Décomposer par capacité commerciale
- Décomposer par sous-domaine

- ✚ Patron de base de données par service : impose que chaque service dispose de sa propre base de données afin d'assurer un couplage souple ;
- ✚ Patron de passerelle API : définit la manière dont les clients accèdent aux services dans une architecture de micro-service ;
- ✚ Patron de découverte côté client et découverte côté serveur : ils sont utilisés pour router les demandes d'un client vers une instance de service disponible dans une architecture de micro-service ;
- ✚ Patron d'appel de messagerie et d'appel de procédure distante sont deux méthodes différentes permettant aux services de communiquer ;
- ✚ Patron service unique par hôte et services multiples par hôte sont deux stratégies de déploiement différentes ;
- ✚ Patron de problèmes transversaux : modèle de châssis micro-service et configuration externalisée ;
- ✚ Patron de test : test de composant de service et test de contrat d'intégration de service ;
- ✚ Patron d'interface utilisateur :
 - Composition de fragments de page côté serveur ;
 - Composition de l'interface utilisateur côté client.

1.3 Évaluation des applications micro-services

Le choix d'une architecture pour un projet logiciel est une opération vitale, en effet, une architecture inappropriée précipitera un projet à sa perte. Les objectifs liés à la performance ne seront pas atteints et ceux liés à la sécurité ne seront même pas considérés. L'utilisateur sera mécontent car la fonctionnalité désirée n'est pas disponible et qu'il est trop difficile de modifier le système pour l'ajouter. Les plannings et budgets ne seront également pas respectés et des mois ou des années plus tard, les changements qui auront été anticipés seront finalement abandonnés car ils deviendront en réalité trop coûteux.

L'architecture détermine également la structure du projet : les bibliothèques de contrôle de configuration, les plannings et budgets, les objectifs de performance, la structure de l'équipe, l'organisation des documentations, et les activités de test ou de maintenance sont toutes organisées autour de l'architecture. S'il s'avère qu'elle doit être modifiée en raison d'une éventuelle défaillance découverte tardivement, le projet entier peut être remis en cause. Il est nettement préférable de modifier l'architecture avant que quoi que ce soit n'ait encore été commencé afin de limiter les confits d'une éventuelle reconstruction du projet.

L'évaluation d'architecture est une manière bon marché d'éviter la catastrophe et elle n'ajoute que quelques jours de plus au planning initial. Elle peut être réalisée durant deux phases du cycle de vie d'un logiciel :

- Avant son implémentation (évaluation hâtive) : L'évaluation précoce d'architecture logicielle peut être effectuée à partir des spécifications, de la description de l'architecture logicielle et d'autres sources d'information, telles que les entretiens avec des architectes.
- Après son implémentation (évaluation tardive) : L'évaluation tardive de l'architecture logicielle est effectuée à partir de métriques et peut aussi bien être utilisée pour évaluer les systèmes existants avant de futures améliorations ou opérations liées à la maintenance du système que pour identifier les points forts et faibles de l'architecture.

La qualité d'un logiciel est l'un des enjeux majeurs dans la conception des logiciels de nos jours. L'analyse et l'évaluation des logiciels sont deux phases ayant connu un essor très important dans la communauté des systèmes informatiques durant ces trois dernières décennies. L'effort de développement, le temps et la complexité ayant considérablement augmenté, il est important d'assurer une qualité optimale au logiciel pour limiter les frais de maintenance.

1.3.1 Exigences requises pour une évaluation

À mesure qu'émergent de nouvelles méthodes et paradigmes de développement logiciel, de nouveaux systèmes de référence sont nécessaires pour aider les développeurs à mener des études empiriques répétables impliquant ces nouvelles tendances en matière de développement logiciel. C'est le cas du style architectural micro-service, pour lequel il n'existe actuellement qu'une poignée de systèmes open source.

Toutefois pour qu'un système entre dans cette liste d'outil de référence, il est nécessaire qu'il passe certaines exigences. Pour ce qui est des micro-service, ces exigences reflètent la manière dont les applications de micro-services typiques sont actuellement développées et mises en production, comme indiqué par les praticiens et les experts du secteur [10], [9]. Ils reposent en particulier, sur le principe que la plupart des applications de micro-services adoptent des modèles architecturaux bien connus [9] et suivent des pratiques de développement logiciel modernes et agiles, telles que celles préconisées par les initiatives DevOps et Twelve-Factor App, auxquelles viennent s'ajouter quelques exigences générales qui, sont bénéfiques directement à la communauté de la recherche en génie logiciel.

1.3.1.1 Exigences architecturales

Ces exigences représentent les attributs attendus de la part d'une application micro-services du point de vue de sa conception architecturale.

Exigence 1 : vue topologique explicite : une application micro-services typique est composée de plusieurs petits services pouvant être déployés indépendamment qui peuvent interagir de manière asynchrone et indirecte au moment de l'exécution [4]. Ces caractéristiques font qu'il est difficile pour un développeur de bien comprendre tous les points d'intégration et les responsabilités des services au sein de l'architecture globale de l'application en se basant uniquement sur son code source. Une application micro-services bien documentée devrait fournir une vue explicite de ses principaux éléments de service et de leurs canaux de communication potentiels au moment de l'exécution. Une telle vision est importante pour aider les acteurs du génie logiciel à mieux comprendre, explorer et évaluer les décisions de conception architecturale.

Exigence 2 : Conception basée sur des modèles : Les avantages d'une architecture logicielle basée sur des modèles, tels que la facilité de maintenance et de réutilisation, sont reconnus depuis longtemps par la communauté des ingénieurs en logiciel. À cet égard, un certain nombre de modèles architecturaux testés par l'industrie ont déjà été proposés pour prendre en charge la création d'applications de micro-services robustes et évolutives. Circuit-breaker, API Gateway et Service Discovery sont parmi les modèles de micro-services les plus connus [17]. L'utilisation de tels modèles est donc attendue dans la conception d'une application micro-services, car correspondant à la manière dont les applications de micro-services sont actuellement développées et livrées dans des environnements de production.

1.3.1.2 Exigences DevOps

Ces exigences reflètent le souhait du secteur selon lequel une application de micro-services prête pour la production devrait suivre les principes clés du modèle de développement continu DevOps [8]. Ceux-ci incluent un accès facile au code source à partir d'un système de contrôle de version, ainsi que la prise en charge des pratiques DevOps classiques telles que l'intégration continue, les tests automatisés, la gestion des dépendances, le déploiement automatisé et l'orchestration de conteneurs [8].

Exigence 3 : Accès facile depuis un système de contrôle de version : L'utilisation d'un système de contrôle de version (VCS) est un aspect crucial du développement de tout logiciel moderne, et plus encore dans un environnement distribué.

Exigence 4 : Prise en charge de l'intégration continue : L'intégration continue est une pratique de développement logiciel dans laquelle le nouveau code créé sur la machine d'un développeur est automatiquement intégré à la base de codes du logiciel existant après chaque validation de code soumise au système de contrôle de version. Des outils d'intégration continue tels que Jenkins et TeamCity sont chargés de créer une nouvelle version d'une application et d'informer l'équipe de développeurs des résultats de la construction. Ces outils peuvent également déclencher l'exécution de tâches supplémentaires, telles que le contrôle de la qualité du code et les tests.

Exigence 5 : Prise en charge des tests automatisés : des outils de test automatisés tels que Cucumber et Selenium sont capables d'exécuter des tests, de rapporter leurs résultats et de les comparer avec des exécutions de tests antérieures. Les tests effectués avec ces outils peuvent être exécutés à plusieurs reprises, à tout moment.

Exigence 6 : Prise en charge de la gestion des dépendances : un outil de gestion des dépendances tel que Maven ou NPM est chargé de télécharger et d'installer automatiquement tous les artefacts logiciels externes (par exemple : composants, bibliothèques) nécessaires à la création d'un produit logiciel donné. Ces outils fournissent une notation spécifique pour décrire ces dépendances, appelée fichier manifeste. Les dépendances spécifiées dans un fichier manifeste sont généralement téléchargées à partir d'un dépôt logiciel central.

Exigence 7 : Prise en charge des images de conteneur réutilisables : les applications de micro-services sont généralement déployées dans une infrastructure virtualisée, comme celle fournie par les fournisseurs de cloud public. Pour accélérer le déploiement, les développeurs s'appuient généralement sur des outils de virtualisation conteneurisés légers, tels que Docker, pour créer des images de conteneur réutilisables avec l'ensemble de la pile de logiciels et l'environnement d'exécution nécessaires à l'exécution de chaque composant de l'application. Cela permet à l'application d'être facilement déployée dans le même environnement virtuel, indépendamment de l'infrastructure physique sous-jacente (par exemple, des machines de développement, des serveurs de production).

Exigence 8 : Prise en charge du déploiement automatisé : la configuration de déploiement d'une application micro-service typique peut varier considérablement d'un environnement d'exécution à un autre (par exemple : développement, transfert, production). Si ces variations étaient intégrées à la mise en œuvre de l'application, la modification de son environnement d'exécution nécessiterait également la modification de son code source. Bien entendu, cela ferait du déploiement d'applications une tâche très exigeante et sujette aux erreurs. Pour

résoudre ce problème, les développeurs spécifient généralement des informations de configuration dépendant de l'environnement dans des artefacts externes au code source, qui sont ensuite utilisés par les outils de déploiement automatisés, tels que Chef et Ansible. En général, ces outils offrent un moyen structuré et centralisé de spécifier les différentes manières de déployer une application de micro-services au moment de l'exécution.

Exigence 9 : Prise en charge de l'orchestration de conteneur : une fonctionnalité intéressante des conteneurs est qu'ils peuvent être automatiquement planifiés et orchestrés au-dessus de toute infrastructure informatique physique ou virtualisée [5]. Docker Swarm, Kubernetes et Mesos sont trois des outils d'orchestration de conteneurs les plus couramment utilisés. Ces outils fournissent un support automatisé permettant de relever certains défis majeurs liés au déploiement d'applications micro-services, tels que la découverte de services, l'équilibrage de la charge et les mises à niveau progressives [1].

1.3.2 Système de références pour l'évaluation

La communauté de la recherche en ingénierie logicielle s'appuie depuis longtemps sur les systèmes logiciels open source comme points de repère courants pour mener et reproduire une grande variété d'études empiriques. Parmi les exemples de tests de génie logiciel utilisés dans des études antérieures, citons le navigateur Web Mosaic, l'application Web Java Pet Store, l'outil de dessin graphique JHotDraw, l'outil de modélisation ArgoUML et le Systèmes d'exploitation Linux et Android.

Pour la réalisation d'une évaluation de l'implémentation d'une architecture micro-service, il est important de disposer d'applications de référence (Benchmark). À cet effet, la communauté scientifique dispose d'applications implémentées selon ce style et qui respectent certaines des exigences ci-dessus citées. Celles-ci serviront de références pour la réalisation de toutes études dans le domaine des micro-services.

Après avoir effectué une recherche sur le Web notamment sur les sites de partage de codes (GitHub) pour trouver des applications de micro-services open source existantes qui pourraient être de bons candidats conformément aux neuf exigences citées dans la section précédente. Il n'existe que quelques applications de micro-services accessibles au public qui pourraient répondre à nos besoins. Les six que nous avons trouvés les plus appropriés sont décrits ci-dessous. Leurs principales caractéristiques, notamment le domaine d'application, le nombre de services conteneurisés, le langage de programmation et l'état de développement, sont résumées dans le tableau 1.

Tableau 1 : Evaluation de quelques applications micro-services de référence [13].

Exigence de référence pour les micro-services		TeaStore	ACME Air	Spring Cloud Demo	Sock Shop	MusicStore
R1	Vue topologique explicite					
R2	Conception basée sur des modèles					
R3	Système de contrôle de version					
R4	Intégration continue					
R5	Tests automatisés					
R6	Gestion des dépendances					
R7	Images de conteneur réutilisables					
R8	Déploiement automatisé					
R9	Orchestration de conteneur					

NB : $R_x = \text{Exigence } x$.

Acme Air

Cette application de micro-services simule le site Web d'une compagnie aérienne fictive [14]. Il est composé de quatre projets distincts : deux infrastructures de serveur, développées respectivement avec Java EE et Node.js, et deux modèles architecturaux respectant une architecture monolithique et une architecture de micro-services.

Applications de démonstration Spring Cloud

Il s'agit de deux applications de micro-services développées pour illustrer certains des concepts fondamentaux de la création d'architectures basées sur des micro-services à l'aide de Spring Cloud et de Docker. La première application [15] implémente une plate-forme de traitement de graphes pour le classement des communautés d'utilisateurs sur Twitter. La deuxième application [16] implémente un catalogue de films en ligne avec un service de recommandation de film associé.

Socks Shop

Cette application de micro-services simule la partie d'un site Web de commerce électronique vendant des chaussettes [17]. Il a été développé à l'aide de Spring Boot, Go et

Node.js, dans le but spécifique d'aider à la démonstration et au test des technologies existantes de micro-service et dans le cloud.

MusicStore

MusicStore est une application d'évaluation bien connue, développée à l'origine par Microsoft pour démontrer les fonctionnalités ASP.NET [9]. L'équipe du framework Steeltoe a divisé cette application en plusieurs services indépendants pour illustrer comment utiliser ensemble leurs composants dans une application ASP.NET de base dans un contexte de micro-services [18].

TeaStore

TeaStore [13] est une application de micro-services distribuée comprenant cinq services distincts et un registre. Chaque service peut être répliqué sans limite et déployé sur des périphériques distincts, selon les besoins. Les services communiquent via REST et l'équilibreur de charge côté client Netflix Ribbon. Chaque service est également proposé dans une variante pré-instrumentée utilisant Kieker pour fournir des informations détaillées sur les actions et le comportement du TeaStore.

Conclusion

Nous avons présenté dans ce chapitre le contexte global dans lequel s'inscrit ce travail de mémoire : le domaine des architectures logicielles à base de composants, plus précisément sur le style micro-service, les prérequis pour l'évaluation de l'implémentation d'une architecture micro-service avec certaines technologies.

Nous avons montré comment l'architecture logicielle s'est progressivement retrouvée sur le devant de la scène, et quelles nouvelles pratiques d'ingénierie sont apparues avec elle. Pour mieux comprendre ce que recouvre l'architecture logicielle, nous avons synthétisé les définitions notables proposées dans la littérature ; abordé les travaux sur les langages de description d'architectures ; et terminé avec la notion de style architectural, spécialement le style orienté service dont découle directement le style micro-service objet de ce travail.

Le style micro-service naît de problèmes rencontrés dans le domaine du génie logiciel, notamment celui de la taille, du besoin d'évolutivité et de la complexité des applications. Bien qu'il n'ait pas une définition qui fasse consensus, nous avons proposé une définition que nous jugeons acceptable, car elle englobe les concepts clés du style. L'étape suivante a donc consisté à présenter de manière successive les caractéristiques puis l'intérêt et enfin les problèmes du

style. Nous clôturons cette section par quelques standards sur lesquels doit être bâti une application micro-service.

La section trois est celle qui aborde l'évaluation de l'implémentation d'une application micro-service. Il en est ressorti que pour mener une telle évaluation il est nécessaire de disposer d'un modèle d'évaluation qui est un ensemble d'exigence qu'une application doit remplir.

Au terme de cette revue de la littérature, le constat qui se dégage est que le style micro-service est un style architectural très récent. Lorsqu'il est bien choisi et implémenté, il permet une organisation exceptionnelle des applications. Toutefois, les outils devant servir de boussole aux architectes logiciels et aux développeurs, lors de la mise en œuvre du style font encore défaut. En effet, les outils (modèles d'évaluation) existants font une évaluation tardive de l'implémentation (lorsque celle-ci est terminée ou presque terminée) et ne vont pas en profondeur dans l'évaluation de la mise en œuvre. Pour cela nous proposons de construire un modèle d'évaluation, applicable à toutes les étapes du projet, spécialement à la phase du choix des technologies, pour évaluation hâtive et profonde de la mise œuvre.

*CHAPITRE 2 : O Q F ~ N G " F ø ; Ø NCDESVCVK
PLATEFORMES POUR N ø K O R N ; O G P V C V K
DU STYLE MICRO-SERVICE*

Introduction

Dans ce chapitre nous proposons un modèle permettant d'évaluer la compatibilité d'une technologie (langage de programmation ou framework) avec le style architectural micro-service. Dans le cycle de vie d'un logiciel, cette étape d'évaluation peut être réalisée avant (évaluation hâtive) ou après (évaluation tardive) l'implémentation. Notre modèle consistant en une série d'exigences d'ordre technique que doit vérifier une technologie, nous nous situons donc dans le cadre d'une évaluation dite hâtive, car cette étude est faite avant l'implémentation de l'application.

Evaluer la compatibilité d'un style architectural avec une certaine technologie, consiste à vérifier que cette technologie conserve les points forts du style, qu'elle apporte des solutions aux problèmes sous-jacents mais surtout qu'elle respecte les standards sur lesquels le style est bâti. Pour ce modèle, nous admettons comme points d'évaluation les standards existants pour le style micro-service. En effet, ceux-ci, s'ils sont respectés entraînent la conservation des atouts du style et sont solutions aux problèmes présentés au chapitre précédent.

Ces standards couvrent tous les aspects de l'implémentation d'une application micro-service : la gestion des données, l'externalisation des services, la composition de services, la sécurité, la résistance aux pannes, etc. Dans la pratique logiciel, ces standards prennent la forme de patrons de conception (design pattern). En informatique, et plus particulièrement en développement logiciel, un patron de conception est une solution générale et réutilisable à un problème courant dans un contexte donné de conception de logiciels. Il s'agit d'une description ou d'un modèle pour résoudre un problème qui peut être utilisé dans de nombreuses situations différentes.

2.1 Exigence d'implémentation par domaine

Le modèle d'évaluation que nous construisons repose sur une série d'exigence (critères) qui lorsqu'elles sont appliquées à une technologie permettent de lui attribuer un niveau de compatibilité avec le style micro-service. Bien que ces exigences varient selon les besoins de l'application, il est nécessaire pour notre modèle de couvrir le plus grand nombre de cas d'implémentation possible.

La première étape a consisté à diviser l'implémentation du style en plusieurs domaines de conception suivant la méthodologie de conception piloté par domaine (DDD). Il s'agit d'une approche de développement de logiciels pour des besoins complexes, en connectant profondément l'implémentation à un modèle évolutif des concepts métier de base. Au terme de

cette première phase, nous avons recensé onze principaux domaines, dans lesquels peuvent être classées un ou plusieurs atouts ou limites du style micro-service.

La deuxième étape a consisté en la recherche de patrons de conception qui font office de bonne pratique pour l'implémentation des atouts et problèmes recensés par domaine de conception lors de la première étape. Puisque ces patrons de conception sont universellement reconnus par la communauté du génie logiciel et qu'ils couvrent la majorité des cas d'implémentation possible, ce sont eux qui dans la suite de ce travail nous servirons d'exigence pour notre modèle d'évaluation.

2.1.1 Gestion des données

Pour une application développée en utilisant le style d'architecture micro-service et dans laquelle les services ont besoin de persister les données dans une base de données, la problématique de l'architecture de base de données qui correspondra le mieux se pose immédiatement.

Les patrons de conception, solutions à ces différentes questions sont :

1. Base de données par service (Database per service)

Problème : Quelle est l'architecture de la base de données dans une application de micro-services ?

Solution : Consiste à conserver les données de chaque micro-service dans un espace privée et les rendre accessibles uniquement via son API. Il n'est pas nécessaire de configurer un serveur de base de données pour chaque service. Par exemple, si le SGBD est relationnel, les options sont les suivantes :

- Tables privées par service : chaque service possède un ensemble de tables accessibles uniquement à ce service ;
- Schéma par service : chaque service dispose d'un schéma de base de données. C'est un service privé pour ce service ;
- Serveur de base de données par service - chaque service a son propre serveur de base de données.

2. Composition d'API (API Composition)

Problème : Comment implémenter des requêtes dans une architecture de micro-service ?

Solution : Lorsque le patron de gestion des données est modèle Base de données par service, il n'est plus simple d'implémenter des requêtes qui joignent des données provenant de plusieurs

services. Par conséquent implémenter une requête nécessite de définir un API Composer, qui appelle les services propriétaires des données et effectue une jointure en mémoire des résultats.

3. SAGA

Problème : Comment maintenir la cohérence des données entre les services ?

Solution : Ce patron est une solution à la difficile problématique du maintien de la cohérence des données entre service, qui possède sa propre base de données. Cependant, certaines transactions s'étendent sur plusieurs services, d'où le besoin d'un mécanisme pour garantir la cohérence des données entre les services. La solution : Implémentez chaque transaction couvrant plusieurs services comme une saga. Une saga est une séquence de transactions locales. Chaque transaction locale met à jour la base de données et publie un message ou un événement pour déclencher la transaction locale suivante dans la saga. Si une transaction locale échoue parce qu'elle enfreint une règle de gestion, la saga exécute une série de transactions compensatrices qui annulent les modifications apportées par les transactions locales précédentes.

Il existe deux façons de coordonner les sagas:

- **Chorégraphie** : chaque transaction locale publie des événements de domaine qui déclenchent des transactions locales dans d'autres services

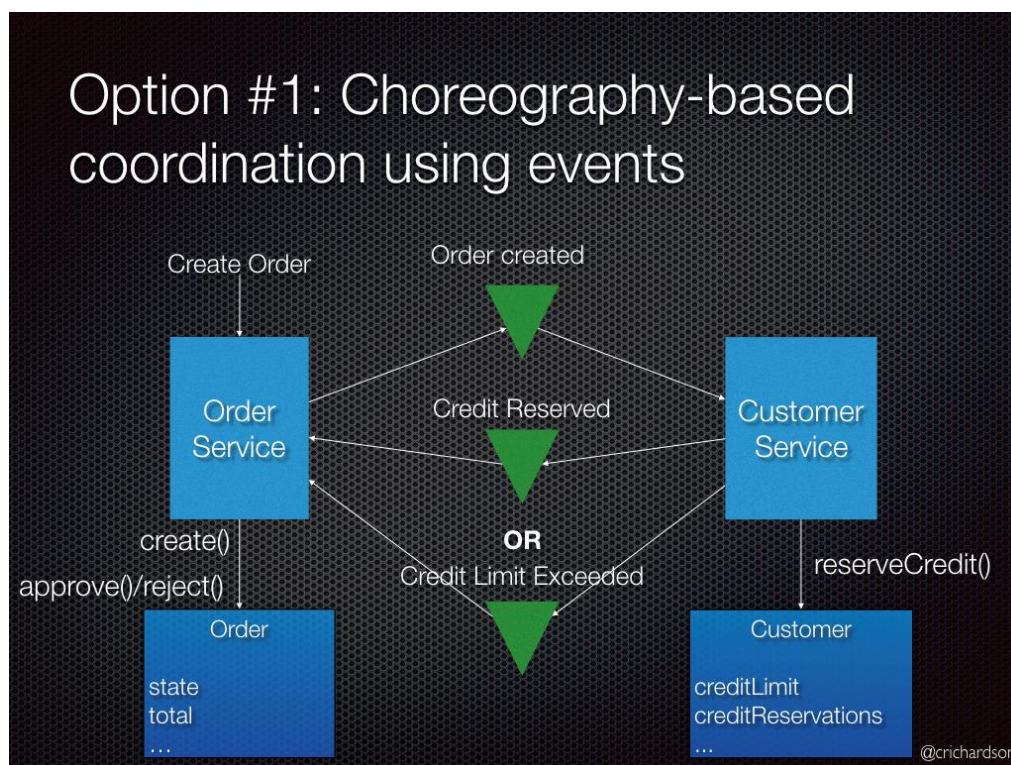


Figure 6 : Implémentation du pattern SAGA, par chorégraphie [9]

- Orchestration - un orchestrateur (objet) indique aux participants les transactions locales à exécuter

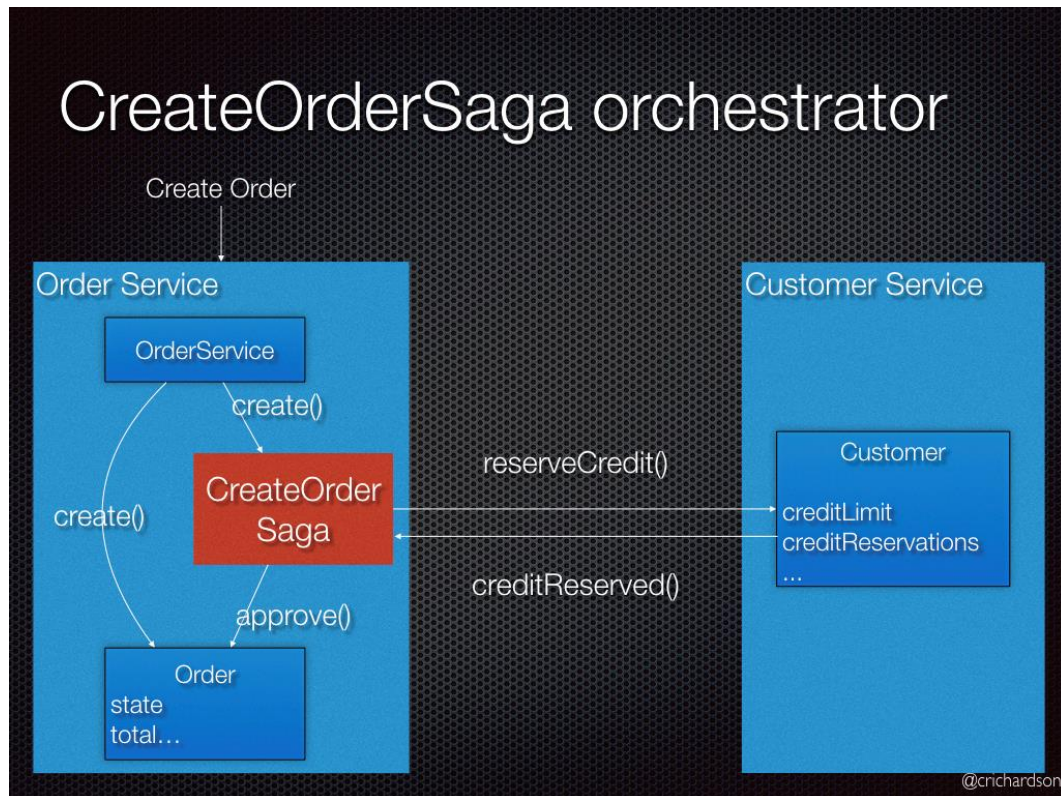


Figure 7 : Implémentation du pattern SAGA, par orchestration [9]

4. Événement de domaine (Domaine Event)

Problème : Comment un service publie-t-il un événement lorsqu'il met à jour ses données?

Solution : Il consiste à organiser la logique métier d'un service sous la forme d'une collection d'agrégats DDD (Domain-Driven Design) qui émettent des événements lors de leur création ou de leur mise à jour. Le service publie ces événements afin qu'ils puissent être consommés par d'autres services. Ce patron intervient notamment dans le patron SAGA.

5. Sourcing événementiel (Event sourcing)

Problème : Comment mettre à jour de manière fiable / atomique la base de données et publier des messages / événements ?

Solution : Le sourcing d'événements consiste à conserver l'état d'une entité sous la forme d'une séquence d'événements qui changent d'état. Chaque fois que l'état d'une entité change, un nouvel événement est ajouté à la liste des événements. Étant donné que l'enregistrement d'un événement est une opération unique, il est intrinsèquement atomique. L'application reconstruit l'état actuel d'une entité en rejouant les événements. Les applications conservent les événements

dans un magasin d'événements, qui est une base de données d'événements. Le magasin dispose d'une API pour ajouter et récupérer les événements d'une entité. Le magasin d'événements se comporte également comme un courtier de messages. Il fournit une API qui permet aux services de s'abonner à des événements. Lorsqu'un service enregistre un événement dans le magasin d'événements, il est distribué à tous les abonnés intéressés.

2.1.2 Gestion des tests (Testing)

Pour une application développée suivant le style d'architecture micro-service et comprenant de nombreux services, qui invoquent souvent d'autres services, il se pose rapidement le problème d'implémentation de tests automatisés qui vérifient qu'un service se comporte correctement.

1. Test des composants de service

Problème : Comment testez-vous facilement un service?

Solution : Une suite de tests qui teste un service de manière isolée à l'aide de tests doubles pour tous les services qu'il appelle.

2. Test de contrat d'intégration de services

Problème : Comment tester facilement qu'un service fournit une API que ses clients attendent?

Solution : Une suite de tests pour un service qui est écrit par les développeurs d'un autre service qui le consomme. La suite de tests vérifie que le service répond aux attentes du service consommateur.

2.1.3 Déploiement

Pour une application développée suivant le style d'architecture micro-service. Chaque service est déployé comme un ensemble d'instances de service pour garantir montée en charge et la haute disponibilité. De plus, les services sont écrits en utilisant une variété de langages, de frameworks et de versions de framework; chaque service se compose de plusieurs instances de service ; le service doit être déployable et évolutif de manière indépendante; les instances de service doivent être isolées les unes des autres; il faut être en mesure de limiter les ressources (CPU et mémoire) consommées par un service; il faut surveiller le comportement de chaque instance de service ; il faut s'assurer que le déploiement soit fiable et qu'il s'exécute de la manière la plus rentable possible.

1. Plusieurs instances de service par hôte(Multiple service instances per host)

Problème : Comment les services sont-ils regroupés et déployés?

Solution : Exécutez plusieurs instances de différents services sur un hôte (machine physique ou virtuelle). Il existe différentes manières de déployer une instance de service sur un hôte partagé, notamment:

- Déployer chaque instance de service en tant que processus JVM. Par exemple, une instance Tomcat ou Jetty par instance de service ;
- Déployez plusieurs instances de service dans la même JVM. Par exemple, sous forme d'applications Web ou de bundles OSGI.

2. Instance de service par conteneur (Service instance per container)

Problème : Comment les services sont-ils regroupés et déployés?

Solution : Empaqueter le service en tant qu'image de conteneur (Docker) et déployer chaque instance de service en tant que conteneur.

3. Déploiement sans serveur

Problème : Comment les services sont-ils regroupés et déployés?

Solution : Utiliser une infrastructure de déploiement qui cache tout concept de serveurs (c'est-à-dire des ressources réservées ou préallouées) - des hôtes physiques ou virtuels ou des conteneurs. L'infrastructure prend le code du service et l'exécute. La facturation se fait en fonction des ressources consommées. Pour déployer le service à l'aide de cette approche, il est nécessaire d'empaqueter le code (par exemple sous forme de fichier ZIP), le télécharger dans l'infrastructure de déploiement et décrire les caractéristiques de performances souhaitées. L'infrastructure de déploiement est un utilitaire exploité par un fournisseur de cloud public. Il utilise généralement des conteneurs ou des machines virtuelles pour isoler les services. Cependant, ces détails sont cachés du client. Ni vous ni personne d'autre dans votre organisation n'est responsable de la gestion d'une infrastructure de bas niveau telle que les systèmes d'exploitation, les machines virtuelles, etc.

2.1.4 Préoccupations transversales

Une application utilise généralement une ou plusieurs infrastructures et services tiers. Exemples de services d'infrastructure: un registre de services, un courtier de messages et un serveur de base de données. Exemples de services tiers: traitement des paiements, e-mail et messagerie, etc. En outre : un service doit être fourni avec des données de configuration qui lui

indiquent comment se connecter aux services externes / tiers; un service doit s'exécuter dans plusieurs environnements - développement, test, qa, transfert, production - sans modification et / ou recompilation; différents environnements ont différentes instances des services externes / tiers.

1. Configuration externalisée (Externalized configuration)

Problème : Comment permettre à un service de s'exécuter dans plusieurs environnements sans modification?

Solution : Externalisez toute la configuration de l'application, y compris les informations d'identification de la base de données et l'emplacement réseau. Au démarrage, un service lit la configuration à partir d'une source externe, par ex. Variables d'environnement du système d'exploitation, etc.

2.1.5 Style de communication

Pour répondre aux requêtes qu'elle reçoit des clients, une application micro-service doit parfois faire collaborer ses services pour traiter ces demandes. Ils doivent utiliser un protocole de communication inter-processus.

1. Appel de procédure à distance (Remote Procedure Invocation : RPI)

Problème : Comment faire communiquer deux services ?

Solution : Utilisez RPI pour la communication inter-services. Le client utilise un protocole basé sur les requêtes / réponses pour effectuer des requêtes auprès d'un service. Il existe de nombreux exemples de technologies RPI : REST, gRPC, Apache Thrift.

2. Echange de messages (Messaging)

Problème : Comment faire communiquer deux services ?

Solution : Utilisez la messagerie asynchrone pour la communication inter-services. Les services communiquent en échangeant des messages sur des canaux de messagerie. Il existe plusieurs styles différents de communication asynchrone :

- Demande / réponse : un service envoie un message de demande à un destinataire et s'attend à recevoir rapidement un message de réponse ;
- Notifications : un expéditeur envoie un message à un destinataire mais n'attend pas de réponse ;

- Demande / réponse asynchrone : un service envoie un message de demande à un destinataire et s'attend à recevoir un message de réponse éventuellement ;
- Publier / s'abonner : un service publie un message à zéro ou plusieurs destinataires ;
- Publier / réponse asynchrone : un service publie une demande à un ou plusieurs destinataires, dont certains renvoient une réponse

2.1.6 API externe

Pour une application développée suivant le style d'architecture micro-service et qui doit interagir avec des interfaces utilisateur hétérogènes :

- Interface utilisateur basée sur HTML5 / JavaScript pour les navigateurs de bureau et mobiles - le HTML est généré par une application Web côté serveur ;
- Clients Android et iPhone natifs - ces clients interagissent avec le serveur via des API REST

De plus, l'application en ligne doit exposer des données via une API REST pour une utilisation par des applications tierces. Une interface utilisateur peut afficher de nombreuses informations. Si on prend l'exemple, la page de détails d'un livre en vente sur le site de e-commerce Amazon.com, celle-ci regroupe :

- Les informations de base sur le livre telles que le titre, l'auteur, le prix, etc ;
- L'historique d'achat pour le livre ;
- Disponibilité ;
- Options d'achat ;
- Autres articles qui sont fréquemment achetés avec ce livre ;
- Autres articles acheté par les clients qui ont acheté ce livre ;
- Avis des clients ;
- Classement des vendeurs ;
- Etc.

Comme cette application utilise le modèle d'architecture micro-service, les données détaillées du produit sont réparties sur plusieurs services. Par exemple,

- Product Info Service informations de base sur le produit telles que titre, auteur ;
- Pricing Service : prix du produit ;
- Service de commande : historique des achats pour le produit ;
- Service d'inventaire : disponibilité du produit ;
- Service de révision : avis des clients.

Par conséquent, le code qui affiche les détails du produit a besoin pour récupérer des informations de tous ces services. En outre, nous avons comme contraintes :

- La granularité des API fournies par les microservices est souvent différente de celle dont a besoin un client. Les microservices fournissent généralement des API à granularité fine, ce qui signifie que les clients doivent interagir avec plusieurs services. Par exemple, comme décrit ci-dessus, un client ayant besoin des détails d'un produit doit extraire des données de nombreux services ;
- Différents clients ont besoin de données différentes. Par exemple, la version de navigateur de bureau présente une page de détails de produit généralement plus élaborée que la version mobile ;
- Les performances du réseau sont différentes pour chaque types de clients. Par exemple, un réseau mobile est généralement beaucoup plus lent et a une latence beaucoup plus élevée qu'un réseau non mobile. Et, bien sûr, tout WAN est beaucoup plus lent qu'un LAN. Cela signifie qu'un client mobile natif utilise un réseau qui présente des caractéristiques de performances très différentes d'un LAN utilisé par une application Web côté serveur ;
- L'application Web côté serveur peut faire plusieurs demandes de services backend sans impact sur l'expérience utilisateur alors qu'en tant que client mobile, il ne peut en faire que quelques-unes. Le nombre d'instances de service et leur emplacement (hôte + port) changent dynamiquement ;
- Le partitionnement en services peut changer au fil du temps et doit être caché aux clients ;
- Les services peuvent utiliser un ensemble divers de protocoles, dont certains peuvent ne pas être compatibles avec le Web.

1. Passerelle API

Problème : Comment les clients d'une application basée sur les micro-services accèdent-ils aux services individuels?

Solution : Implémentez une passerelle API qui est le point d'entrée unique pour tous les clients. La passerelle API gère les demandes de deux manières. Certaines demandes sont simplement mandatées/acheminées vers le service approprié. Il gère d'autres demandes en se déployant sur plusieurs services. Plutôt que de fournir une API de style unique, la passerelle API peut exposer une API différente pour chaque client. Par exemple, la passerelle API Netflix exécute un code d'adaptateur spécifique au client qui fournit à chaque client une API la mieux adaptée à ses

besoins. La passerelle API peut également implémenter la sécurité, par exemple vérifier que le client est autorisé à exécuter la demande.

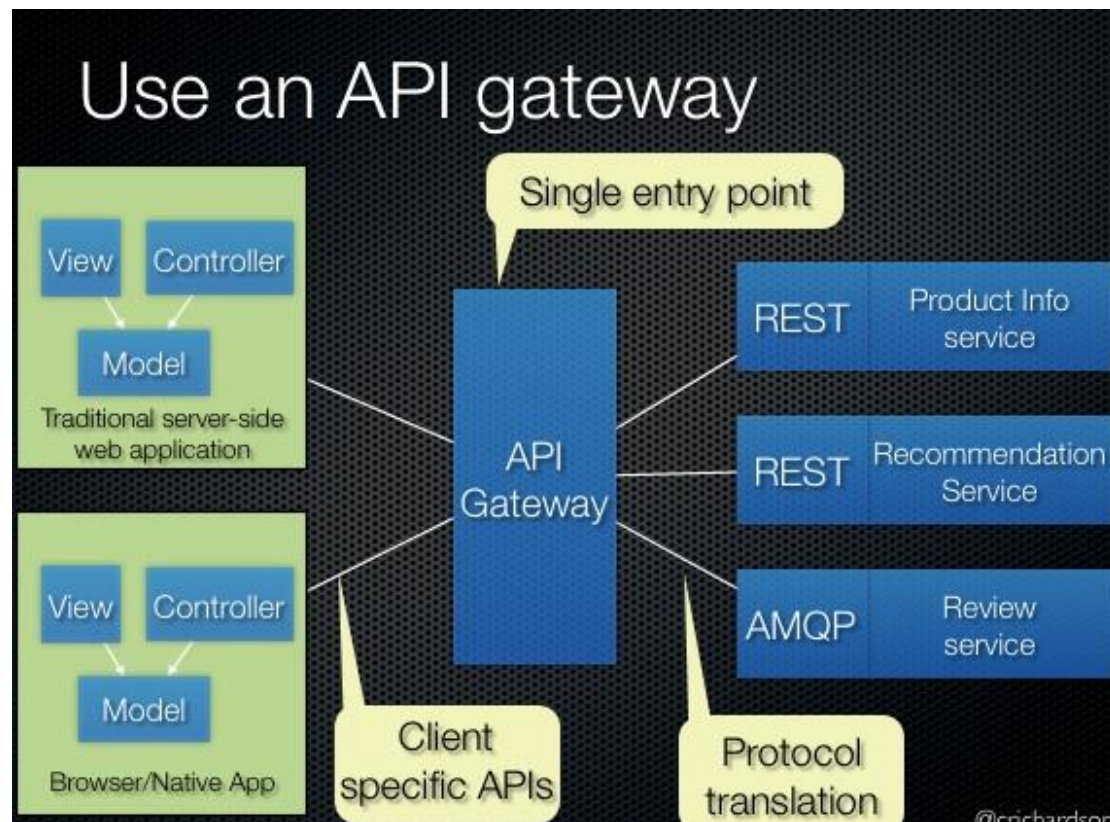


Figure 8 : Passerelle API – un seul point d'entrée [9]

2. Backends pour frontends

Problème : Comment les clients d'une application basée sur les micro-services accèdent-ils aux services individuels ?

Solution : Il s'agit d'une variation du modèle de passerelle API. Il définit une passerelle API distincte pour chaque type de client. Dans cet exemple, il existe trois types de clients : application Web, application mobile et application tierce externe. Il existe trois passerelles API différentes. Chacune fournissant une API pour son client.

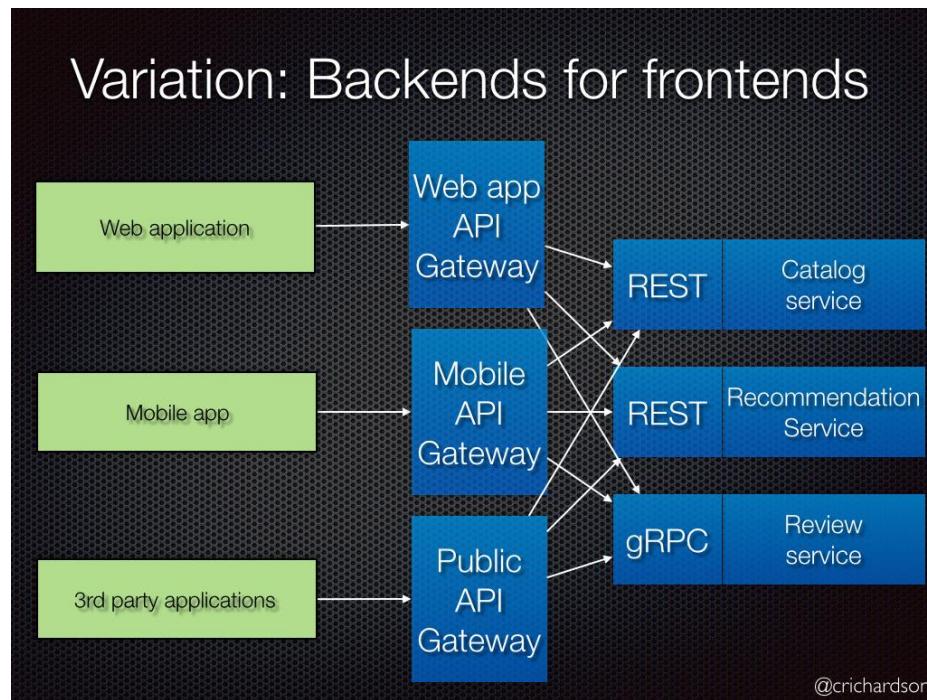


Figure 9 : Passerelle API - Point d'entrée selon le type de client [9]

2.1.7 Découverte de service

Dans une application monolithique, les services s'appellent mutuellement via des appels de méthode ou de procédure au niveau du langage. Dans un déploiement de système distribué traditionnel, les services s'exécutent à des emplacements fixes et bien connus (hôtes et ports) et peuvent donc facilement s'appeler à l'aide de HTTP / REST ou d'un mécanisme RPC. Cependant, une application moderne basée sur des micro-services s'exécute généralement dans des environnements virtualisés ou conteneurisés où le nombre d'instances d'un service et leurs emplacements changent de manière dynamique. Par conséquent, il est nécessaire d'implémenter un mécanisme permettant aux clients du service de faire des demandes à un ensemble changeant dynamiquement d'instances de service éphémères. En outre :

- Chaque instance d'un service expose une API distante telle que HTTP / REST, ou Thrift etc. à un emplacement particulier (hôte et port) ;
- Le nombre d'instances de services et leurs emplacements changent dynamiquement ;
- Les machines virtuelles et les conteneurs reçoivent généralement des adresses IP dynamiques ;
- Le nombre d'instances de services peut varier dynamiquement.

1. Découverte de service côté client

Problème : Comment le client d'un service - la passerelle API ou un autre service - découvre-t-il l'emplacement d'une instance de service ?

Solution : Lors d'une demande auprès d'un service, le client obtient l'emplacement d'une instance de service en interrogeant un registre de services, qui connaît les emplacements de toutes les instances de service. Le diagramme suivant montre la structure de ce modèle.

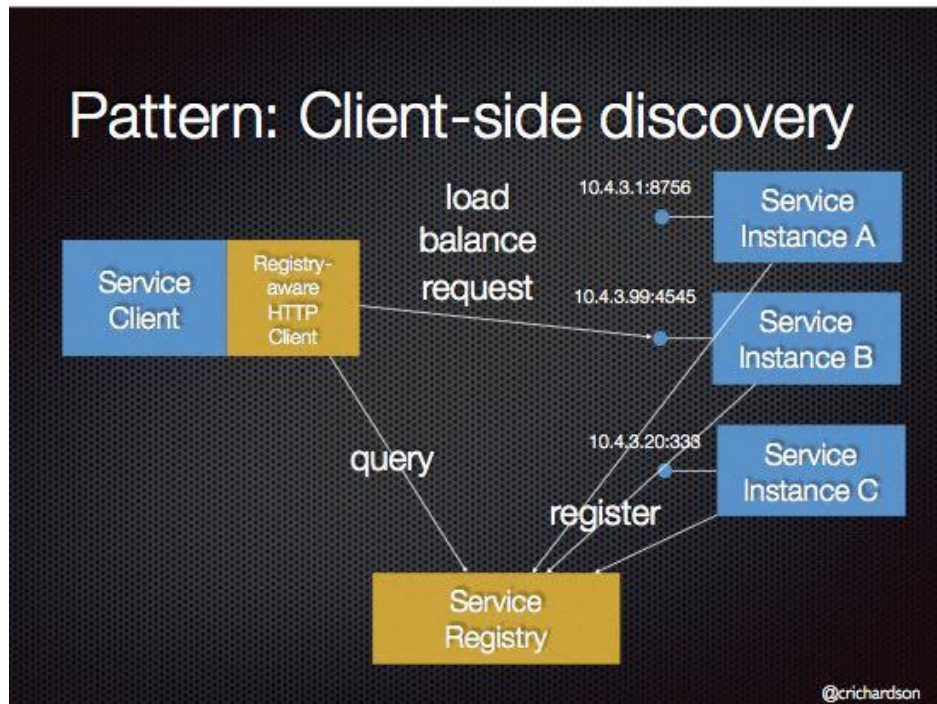


Figure 10 : Découverte de service côté client [9]

2. Découverte de service côté serveur

Problème : Comment le client d'un service - la passerelle API ou un autre service - découvre-t-il l'emplacement d'une instance de service ?

Solution : Lorsqu'il fait une demande à un service, le client fait une demande via un routeur (comme l'équilibreur de charge) qui s'exécute à un emplacement bien connu. Le routeur interroge un registre de services, qui peut être intégré au routeur, et transmet la demande à une instance de service disponible. Le diagramme suivant montre la structure de ce modèle.

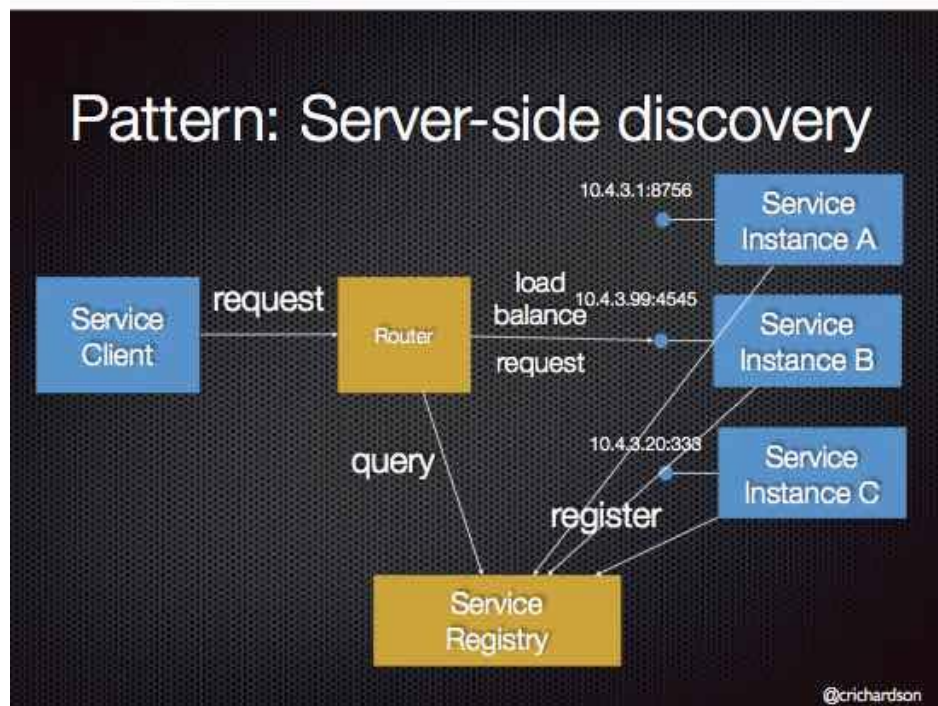


Figure 11 : Découverte de service côté serveur [9]

3. Registre des services

Problème : Comment les clients d'un service (dans le cas d'une découverte côté client) et / ou les routeurs (dans le cas d'une découverte côté serveur) connaissent-ils les instances disponibles d'un service ?

Solution : Implémentez un registre de services, qui est une base de données des services, de leurs instances et de leurs emplacements. Les instances de service sont enregistrées auprès du registre de service au démarrage et annulées à l'arrêt. Le client du service et / ou les routeurs interrogent le registre de services pour rechercher les instances disponibles d'un service. Un registre de services peut appeler l'API de vérification de l'état d'une instance de service pour vérifier qu'il est capable de gérer les demandes.

4. Auto-inscription

Problème : Comment les instances de service sont-elles enregistrées et désinscrites dans le registre de services ?

Solution : Une instance de service est chargée de s'inscrire auprès du registre de services. Au démarrage, l'instance de service s'enregistre (hôte et adresse IP) auprès du registre de service et se rend disponible pour la découverte. Le client doit généralement renouveler périodiquement

son enregistrement afin que le registre sache qu'il est toujours en vie. À l'arrêt, l'instance de service se désinscrit du registre de service.

2.1.8 Fiabilité

Dans une application micro-service, les services sont parfois appelés à collaborer lors du traitement des requêtes. Lorsqu'un service en appelle un autre de manière synchrone, il est toujours possible que l'autre service ne soit pas disponible ou présente une latence si élevée qu'il est essentiellement inutilisable. Des ressources précieuses telles que des threads peuvent être consommées dans l'appelant en attendant que l'autre service réponde. Cela pourrait conduire à l'épuisement des ressources, ce qui rendrait le service appelant incapable de traiter d'autres requêtes. La défaillance d'un service peut potentiellement se répercuter sur d'autres services dans l'application.

1. Disjoncteur

Problème : Comment empêcher une panne de réseau ou de service de se répercuter sur d'autres services ?

Solution : Un client de service doit invoquer un service distant via un proxy qui fonctionne de la même manière qu'un disjoncteur électrique. Lorsque le nombre de pannes consécutives dépasse un seuil, le disjoncteur se déclenche et, pendant la durée d'un délai d'expiration, toutes les tentatives d'appel du service distant échouent immédiatement. Une fois le délai expiré, le disjoncteur autorise le passage d'un nombre limité de demandes de test. Si ces demandes aboutissent, le disjoncteur reprend son fonctionnement normal. Sinon, en cas d'échec, le délai d'expiration recommence.

2.1.9 Sécurité

Pour une application micro-service implémentant les modèles de passerelle API. La passerelle API est le point d'entrée unique pour les demandes des clients. Il authentifie les demandes et les transmet à d'autres services, qui pourraient à leur tour appeler d'autres services. De plus, les services doivent pouvoir vérifier qu'un utilisateur est autorisé à effectuer une opération.

1. Jeton d'accès

Problème : Comment communiquer l'identité du demandeur aux services qui traitent la demande ?

Solution : La passerelle API authentifie la demande et transmet un jeton d'accès (par exemple, un jeton Web JSON) qui identifie en toute sécurité le demandeur dans chaque demande aux services. Un service peut inclure le jeton d'accès dans les demandes qu'il fait à d'autres services.

2.1.10 Observabilité

Pour une application micro-service, composée de plusieurs services et instances de service qui s'exécutent sur plusieurs machines. Chaque instance de service génère des traces d'exécution sur ce qu'elle fait dans un fichier journal dans un format normalisé. Le fichier journal contient des erreurs, des avertissements, des informations et des messages de débogage.

1. Agrégation de journaux

Problème : Comment comprendre le comportement d'une application et résoudre les problèmes ?

Solution : Utilisez un service de journalisation centralisé qui regroupe les journaux de chaque instance de service. Les utilisateurs peuvent rechercher et analyser les journaux. Ils peuvent configurer des alertes qui sont déclenchées lorsque certains messages apparaissent dans les journaux.

2. Mesures d'application

Problème : Comment comprendre le comportement d'une application et résoudre les problèmes ?

Solution : Utiliser un service pour recueillir des statistiques sur les opérations individuelles. Agréger les métriques dans le service de monitoring centralisé, qui fournit des rapports et des alertes.

3. Traçage distribué

Problème : Comment comprendre le comportement d'une application et résoudre les problèmes ?

Solution : Pour une requête nécessitant la collaboration de plusieurs services, un code est attribué à chaque requête. Celui-ci servira d'identifiant de la requête et il est transmis à tous les services impliqués dans le traitement de la requête pour être inclut dans toutes les traces d'exécution enregistrées dans le journal d'exécution (par exemple heure de début, heure de fin).

4. Vérification de la santé de l'API

Problème : Comment détecter qu'une instance de service en cours d'exécution ne peut pas gérer les demandes ?

Contraintes :

- Une alerte doit être générée en cas d'échec d'une instance de service ;
- Les demandes doivent être acheminées vers des instances de service opérationnelles

Solution : Un service possède un point de terminaison API de vérification de l'état de santé, qui renvoie l'état de santé du service. Le gestionnaire de point de terminaison API effectue diverses vérifications, telles que l'état des connexions aux services d'infrastructure utilisés par l'instance de service, l'état de l'hôte, etc. Un client de vérification d'intégrité ou un service de surveillance, un registre de services ou un équilibreur de charge, invoque périodiquement le point de terminaison pour vérifier l'intégrité de l'instance de service.

2.1.11 Modèles d'interface utilisateur

Pour une application développée suivant le style micro-service, où les services sont développés par des équipes orientées capacités/sous-domaines qui sont également responsables de l'expérience utilisateur. Certains écrans / pages d'interface utilisateur affichent des données provenant de plusieurs services. Si on considère par exemple, une page de détails de produit du site Amazon, qui affiche de nombreux éléments de données, notamment : des informations de base sur le livre telles que le titre, l'auteur, le prix, etc. ; votre historique d'achat pour le livre ; disponibilité ; options d'achat ; autres articles fréquemment achetés avec ce livre ; autres articles achetés par les clients qui ont acheté ce livre ; avis clients ; classement des vendeurs ; etc. Chaque donnée correspond à un service distinct et la manière dont elle est affichée est donc de la responsabilité d'une équipe différente.

1. Composition du fragment de page côté serveur

Problème : Comment implémenter un écran ou une page d'interface utilisateur qui affiche les données de plusieurs services ?

Solution : Chaque équipe développe une application Web qui génère le fragment HTML qui implémente la région de la page pour son service. Une équipe d'interface utilisateur est responsable du développement des modèles de page qui créent des pages en effectuant une agrégation côté serveur (par exemple, un mécanisme de style d'inclusion côté serveur) des fragments HTML spécifiques au service.

2. Composition de l'interface utilisateur côté client

Problème : Comment implémenter un écran ou une page d'interface utilisateur qui affiche les données de plusieurs services ?

Solution : Chaque équipe développe un composant d'interface utilisateur côté client, tel qu'une directive Angular, qui implémente la région de la page / de l'écran pour son service. Une équipe d'interface utilisateur est responsable de la mise en œuvre des squelettes de page qui créent des pages / écrans en composant plusieurs composants d'interface utilisateur spécifiques au service.

2.2 Modèle d'évaluation

Pour une application conçue suivant le style architectural micro-service, nous avons mis en évidence un ensemble de bonnes pratiques, qui lorsqu'elles sont implémentées dans une application, permettent d'obtenir une architecture capable de monter en charge, fiable, sécurisé, facile à surveiller, etc.

Puisque l'objectif de notre modèle d'évaluation est de permettre aux architectes logicielles et aux développeurs, de choisir la pile technologique qui leur permettra d'obtenir une application dont l'implémentation est la plus conforme possible au style architectural micro-service, le modèle reposera donc sur les patrons de conception présentés dans la section précédente.

2.2.1 Critères d'évaluation

Ce sont les patrons de conception, ci-dessus recensé. Le tableau suivant indique la liste de ces critères.

Tableau 2 : Critère d'évaluation

N°	Désignation
1	Base de données par service (Database per service)
2	Composition d'API (API Composition)
3	SAGA
4	Événement de domaine (Domaine Event)
5	Sourcing événementiel (Event sourcing)
6	Test des composants de service

MODÈLE D'ÉVALUATION DES PLATEFORMES POUR L'IMPLÉMENTATION DU STYLE MICRO-SERVICE

7	Test de contrat d'intégration de services
8	Plusieurs instances de service par hôte
9	Instance de service par conteneur
10	Déploiement sans serveur
11	Configuration externalisée
12	Appel de procédure à distance
13	Messagerie
14	Passerelle API
15	Backends pour frontends
16	Découverte de service côté client
17	Découverte de service côté serveur
18	Registre des services
19	Auto-inscription
20	Disjoncteur
21	Jeton d'accès
22	Agrégation de journaux
23	Mesures d'application
24	Traçage distribué
25	Vérification de la santé de l'API
25	Composition du fragment de page côté serveur
27	Composition de l'interface utilisateur côté client

2.2.2 Fonction d'évaluation

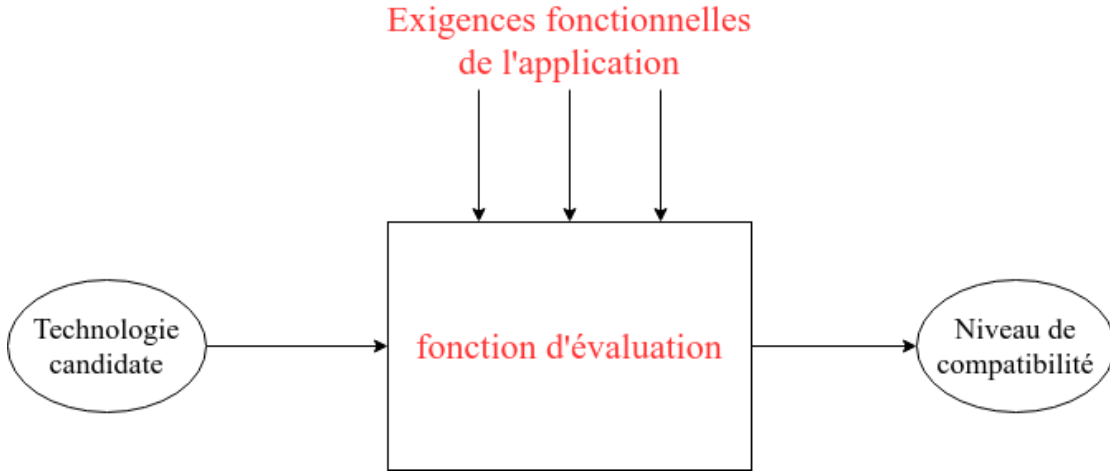


Figure 12 : Modèle d'évaluation

Notre modèle se présente dès lors comme une fonction paramétrique f , ayant pour inconnue une technologie t candidate à l'implémentation du style micro-service et un ensemble de paramètres e_1, e_2, \dots, e_q , qui sont des exigences fonctionnelles de l'application. À chaque exigence nous décidons d'attribuer une valeur dans l'intervalle $[1 - 5]$, selon que l'exigence est plus ou moins importante. Ainsi les exigences les plus faibles auront la valeur 1 et les plus fortes la valeur 5.

Soit P , l'ensemble des critères mesurables pour une étude de la compatibilité d'une technologie. Les éléments de cet ensemble sont les patrons de conception déterminés à la section précédente, nous avons donc : p_1, p_2, \dots, p_n .

NB : Pour notre modèle nous posons l'hypothèse que $q \leq n$ avec $n = 27$.

Soit h la fonction dont le rôle est d'indiquer si un patron de conception est implémenté ou pas dans une technologie. Elle reçoit en entrée deux paramètres : la technologie t et un patron p_i . Si ce patron de conception est implémenté dans cette technologie alors $h(t, p_i) = 1$ sinon 0.

La sortie de cette fonction est une note ($v \in \mathbb{N}$), qui indique le niveau de compatibilité de la technologie avec le style micro-service selon les exigences reçues en input. Celle-ci varie entre 0 et $q \times n = 27q$.

Notre fonction d'évaluation est donc la suivante :

$$f(t) = e_1 \times h(t, p_1) + e_2 \times h(t, p_2) + \dots + e_q \times h(t, p_q) = \sum_{i=1}^q e_i \times h(t, p_i)$$

Encadrement :

on a : $0 \leq h(t, p_i) \leq 1$ et $1 \leq e_i \leq 5$

Puis : $0 \leq e_i \times h(t, p_i) \leq 5$

$$\sum_{i=1}^q 0 \leq \sum_{i=1}^q e_i \times h(t, p_i) \leq \sum_{i=1}^q 5$$

$$0 \leq \sum_{i=1}^q e_i \times h(t, p_i) \leq 5 \times q \text{ or } q \leq n \text{ et } n = 27 \text{ donc } q \leq 27$$

en définitive, l'encadrement de la fonction f est : $0 \leq f(t) \leq 135$

De là, nous constatons que le degré de comptabilité d'une technologie dans notre modèle d'évaluation varie entre 0 et 135.

La représentation de la fonction f d'évaluation est donc la suivante :

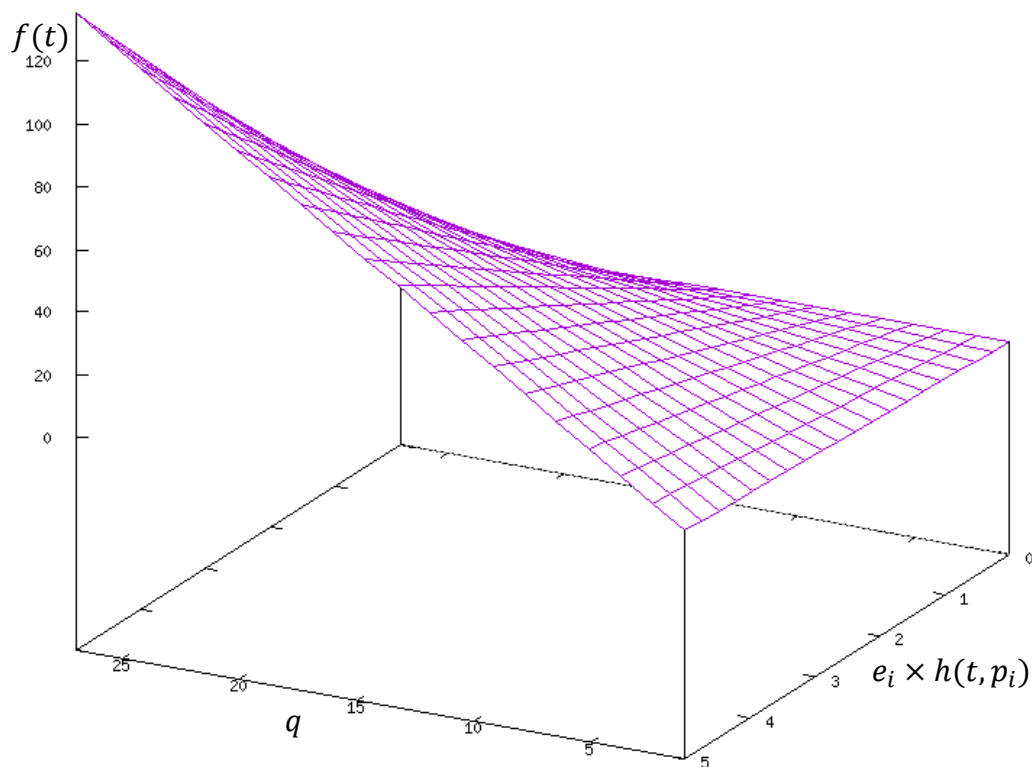


Figure 13 : Représentation de l'espace de la fonction évaluation

Conclusion

Ce chapitre qui est l'objet principal de ce travail de mémoire a consisté en l'élaboration d'un modèle d'évaluation pour l'évaluation de l'implémentation du style architectural micro-service.

Puisque notre modèle doit tenir compte de toutes les spécificités d'implémentation de chaque application, la première étape de ce chapitre a consisté à recenser tous les domaines entrant dans la conception d'une application. Ainsi en utilisant la méthodologie de conception pilotée par les domaines (Domain-Driven Design), nous avons divisé une application en onze principaux domaines.

La division en domaine étant faite, l'étape suivante a consisté à établir une liaison entre les domaines et les problèmes/atouts du style architectural micro-service. De là, nous avons exhibé de bonnes pratiques sous la forme de patron de conception (Design Pattern). Ce qui nous a permis de construire le modèle d'évaluation.

C'est de l'ensemble des patrons recueillis plus haut, que nous avons constitué nos critères d'évaluation de la technologie. Ainsi si une technologie t , implémente un patron p_i , cette technologie reçoit une note v_i , pondérée du coefficient e_i qui indique le degré d'importance de ce patron pour cette application. Dès lors nous avons pu construire une fonction d'évaluation paramétrique $f(t)$, dont les paramètres sont les coefficients e_i . Les valeurs de cette fonction vont de 0 (pas compatible) à 135 (totalement compatible).

CHAPITRE 3 : ÉVALUATION DES TECHNOLOGIES SPRING BOOT ET JAVA EE

Introduction

Dans ce chapitre, nous proposons une évaluation de deux technologies candidates, pour l'implémentation du style micro-service : Spring Boot 2.2.2 et JAVA EE 7. Cette évaluation est faite sur le modèle d'évaluation conçu au chapitre précédent.

Grâce à la fonction d'évaluation qui constitue le cœur du modèle, nous sommes capables d'affecter une note à chaque technologie. Cette note représente le degré de compatibilité de cette technologie avec le style. Cependant il est nécessaire de poser certaines hypothèses avant le début de l'évaluation.

Hypothèse 1 : Puisque nous faisons une étude large, le vecteur d'exigences qui permet le paramétrage de la fonction d'évaluation du modèle, correspondra aux 27 critères formant le modèle d'évaluation.

Hypothèse 2 : Toutes les exigences e_i ont le même niveau d'importance égale à 1.

Hypothèse 3 : La valeur de la fonction h , est obtenu en vérifiant si dans l'univers des packages officielles de la technologie étudiée, il existe un package qui implémente le critère passé en paramètre.

Des hypothèses 1 et 2, il ressort que le degré de compatibilité de ces technologies variera entre 0 et 27.

Toute évaluation se fera en deux étapes :

- Recherche de la valeur de la fonction h , pour chaque critère ;
- Calcul de la valeur de la fonction d'évaluation $f(t)$.

3.1 Évaluation de Spring Boot 2.2.2

Cette évaluation de Spring Boot, se fera sur la version 2.2.2 de ce framework JAVA, très populaire, en développement web back-end.

3.1.1 Présentation

Spring est un framework bien connu des développeurs Java pour les nombreuses fonctionnalités qu'il apporte sur les aspects web, sécurité, batch ou encore accès aux données dans le cadre du développement d'une application. Mais il est aussi malheureusement connu pour sa configuration qui peut s'avérer complexe et fastidieuse. Il n'était pas rare de passer plusieurs jours sur la configuration d'un projet Spring notamment pour des personnes novices dans l'utilisation du framework. Fort de ce constat, les équipes de Spring décidèrent de travailler

sur un projet permettant de faciliter le développement d'application Spring pour les développeurs. C'est ainsi qu'est né Spring Boot.

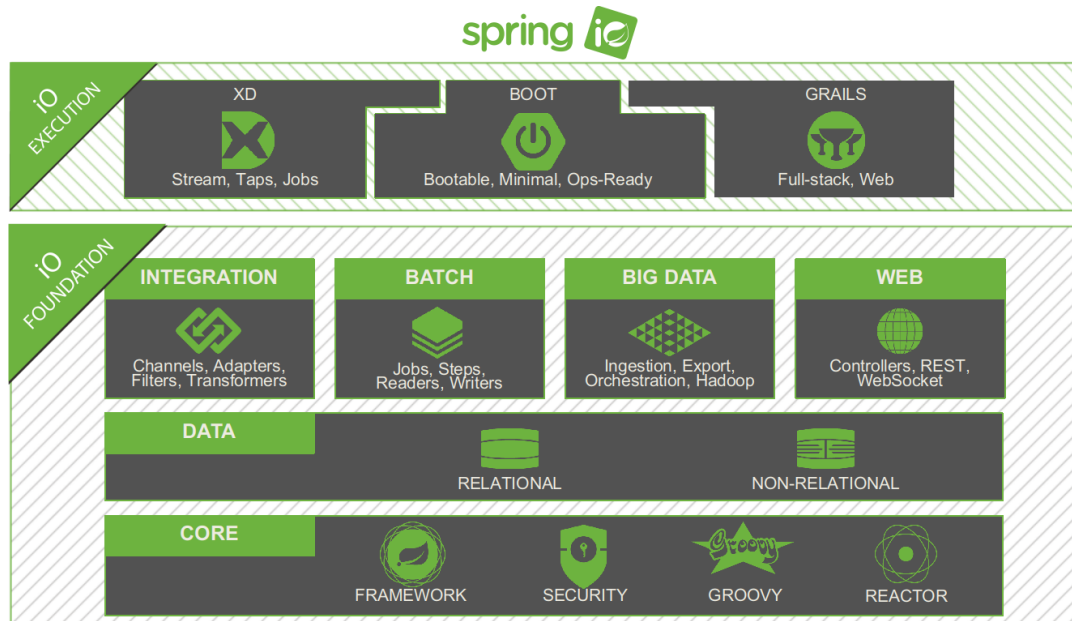


Figure 14 : Architecture du framwork Spring [1]

Spring Boot est un projet ou un micro framework qui a notamment pour but de faciliter la configuration d'un projet Spring et de réduire le temps alloué au démarrage d'un projet. Pour arriver à remplir cet objectif, Spring Boot se base sur plusieurs éléments :

Un site web (<https://start.spring.io/>) qui permet de générer rapidement la structure du projet en y incluant toutes les dépendances Maven nécessaires à application. Cette génération est aussi disponible via le plugin Eclipse STS.

L'utilisation de « Starters » pour gérer les dépendances. Spring a regroupé les dépendances Maven de Spring dans des « méga dépendances » afin de faciliter la gestion de celles-ci. Par exemple pour ajouter toutes les dépendances nécessaires pour gérer la sécurité il suffit d'ajouter le starter « spring-boot-starter-security ».

L'auto-configuration, qui applique une configuration par défaut au démarrage de l'application pour toutes dépendances présentes dans celle-ci. Cette configuration s'active à partir du moment où l'application comporte l'annotation « @EnableAutoConfiguration » ou « @SpringBootApplication ». Bien entendu cette configuration peut être surchargée via des propriétés Spring prédéfinie ou via une configuration Java. L'auto-configuration simplifie la configuration sans pour autant restreindre les fonctionnalités de Spring. Par exemple, si le starter « spring-boot-starter-security » est utilisé, Spring Boot configurera la sécurité dans

l'application avec notamment un utilisateur par défaut et un mot de passe généré aléatoirement au démarrage de votre application.

En plus de ces premiers éléments qui facilitent la configuration d'un projet, Spring Boot offre d'autres avantages notamment en termes de déploiement applicatif. Habituellement, le déploiement d'une application Spring nécessite la génération d'un fichier `.war` qui doit être déployé sur un serveur comme un Apache Tomcat. Spring Boot simplifie ce mécanisme en offrant la possibilité d'intégrer directement un serveur Tomcat dans l'exécutable. Au lancement de celui-ci, un Tomcat embarqué sera démarré afin de faire tourner l'application.

Enfin, Spring Boot met à disposition des opérationnels, des métriques qu'ils peuvent suivre une fois l'application déployée en production. Pour cela Spring Boot utilise «Actuator» qui est un système qui permet de monitorer une application via des URLs spécifiques ou des commandes disponibles via SSH. Toutefois, il est possible de définir ses propres indicateurs très facilement.

Voici une liste non exhaustive des indicateurs disponibles par défaut :

- metrics : métriques de l'application (CPU, mémoire, ...)
- beans : liste des BEANs Spring
- Trace : liste des requêtes HTTP envoyées à l'application
- Dump : liste des threads en cours
- Heath : état de santé de l'application
- env. : liste des profils, des propriétés et des variables d'environnement.

Spring cloud est un projet basé sur Spring Boot pour fournir aux développeurs des outils permettant de créer rapidement certains modèles courants dans les systèmes distribués (configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus, one-time tokens, global locks, leadership election, distributed sessions, cluster state). La coordination des systèmes distribués conduit à des modèles de plaque de chaudière et, grâce à Spring Cloud, les développeurs peuvent rapidement mettre en service des services et des applications qui implémentent ces modèles. Ils fonctionneront bien dans n'importe quel environnement distribué, y compris l'ordinateur portable du développeur, les centres de données et les plateformes gérées telles que Cloud Foundry.

Spring Cloud adopte une approche très déclarative, et de nombreuses fonctionnalités sont obtenues souvent, juste avec un changement de chemin de classe et/ou une annotation.

La figure suivante présente l'architecture d'une application implémentée avec Spring Cloud.

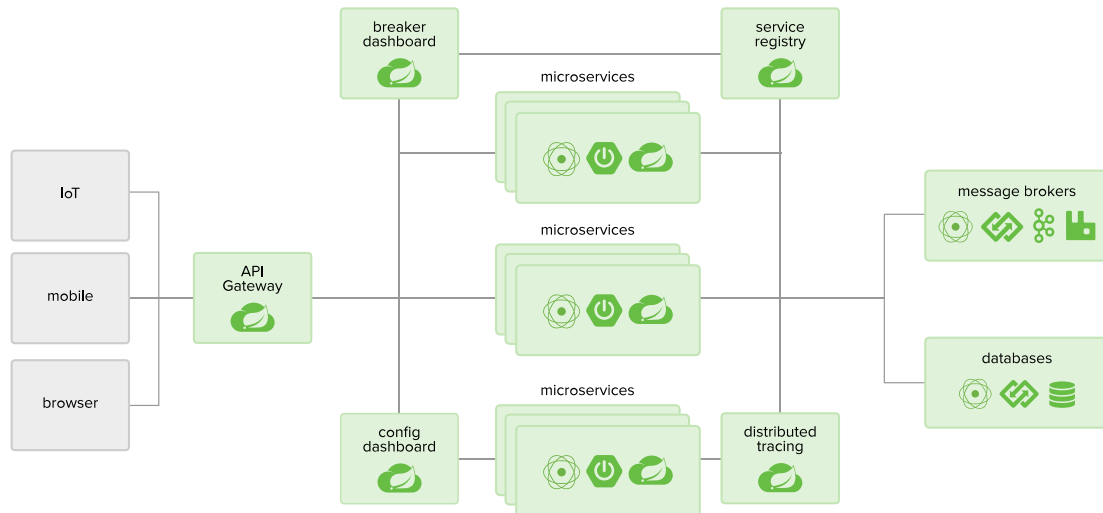


Figure 15 : Architecture d'un projet Spring Cloud [25]

Les principales fonctionnalités offertes par Spring Cloud sont :

- ✚ **Découverte de service (Service Discovery)** : répertoire dynamique qui permet l'équilibrage de la charge côté client et le routage intelligent ;
- ✚ **Disjoncteur (Circuit Breaker)** : tolérance aux pannes de micro-service avec un tableau de bord de surveillance ;
- ✚ **Serveur de configuration (Configuration Server)** : gestion de configuration dynamique et centralisée pour vos applications décentralisées ;
- ✚ **Passerelle API (API Gateway)** : point d'entrée unique pour les utilisateurs d'API (navigateurs, appareils, autres API, par exemple) ;
- ✚ **Journalisation distribué (Distributed Tracing)** : instrumentation d'application automatisée et visibilité opérationnelle pour les systèmes distribués ;
- ✚ **OAuth2** : prise en charge de l'authentification unique, du relais de jeton et de l'échange de jeton ;
- ✚ Etc.

3.1.2 Recherche des valeurs de la fonction h

Tableau 3 : Tableau des valeurs de la fonction h, pour la technologie Spring Boot 2.2.2

	Critères d'évaluation	h	Justification
p1	Base de données par service	1	Puisque Spring offre des packages pour la

			connexion et la manipulation de la plupart des SGBD existant, il est tout à fait compatible avec ce critère.
p2	Composition d'API	1	Grâce au starter data-flow de Spring, il est possible de composer les API, pour obtenir des données.
p3	SAGA	1	Grâce au starter JMS et ActiveMQ, une application Spring peut gérer les événements (émission et réception) nécessaires pour ce critère.
p4	Événement de domaine	1	Grâce au starter JMS et ActiveMQ, une application Spring peut gérer les événements (émission et réception) nécessaires pour ce critère.
p5	Sourcing événementiel	1	Grâce au starter JMS et ActiveMQ, une application Spring peut gérer les événements (émission, réception, abonnement) nécessaires pour ce critère.
p6	Test des composants de service	1	Grâce au starter de Spring notamment l'outil MOCK.
p7	Test de contrat d'intégration de services	1	Grâce au starter cloud-contract, on test l'intégration des services.
p8	Plusieurs instances de service par hôte	1	Dès qu'une JVM est installée sur un hôte, une application Spring peut être lancée et le port d'exécution est attribuée dynamiquement.
p9	Instance de service par conteneur	1	Grâce au starter web, Spring embarque son propre serveur web rendant ainsi déploiement dans un conteneur extrêmement simple.
p10	Déploiement sans serveur	1	Grâce aux gestionnaires de dépendances qui existent en JAVA, il est possible d'envoyer juste son code source pour un déploiement.
p11	Configuration externalisée	1	Par une simple modification du fichier de configuration de Spring il est possible de lui indiquer où aller chercher sa configuration, selon le l'exécution qui est faite.
p12	Appel de procédure à distance	1	Comme Spring utilise JAVA, il embarque toutes les techniques d'appel de procédure à distance.
p13	Messagerie	1	Grâce au starter JMS et ActiveMQ, les services peuvent échanger des messages et

			s'abonner.
p14	Passerelle API	1	Spring boot offre des starters pour retourner les données dans presque tous les formats notamment JSON, XML.
p15	Backends pour frontends	1	Grâce à différents starters permettant de créer des contrôleurs (MVC) selon la méthode d'appel, on peut avoir plusieurs API par client.
p16	Découverte de service côté client	1	Plusieurs implémentations sont disponibles dont la plus utilisée eureka-zuul de netflix.
p17	Découverte de service côté serveur	1	Plusieurs implémentations sont disponibles dont la plus utilisée eureka-zuul de netflix.
p18	Registre des services	1	Plusieurs implémentations sont disponibles dont la plus utilisée service-registry de netflix.
p19	Auto-inscription	1	Plusieurs implémentations sont disponibles dont la plus utilisée eureka-client de netflix.
p20	Disjoncteur	1	Plusieurs implémentations sont disponibles dont la plus utilisée Hystrix de netflix.
p21	Jeton d'accès	1	En combinant les starter security et la dépendance jjwt de maven, on obtient un système sécurisé par token.
p22	Agrégation de journaux	1	Grâce au starter sleuth et RabbitMQ, Spring permet une gestion centralisée des Log.
p23	Mesures d'application	1	Grâce au starter Actuator, il est possible d'avoir l'état de santé de l'application à tout moment ; grâce à des URL, REST disponibles.
p24	Traçage distribué	1	Grâce au starter sleuth et RabbitMQ, Spring permet une gestion distribuée des Log.
p25	Vérification de la santé de l'API	1	Grâce au starter Actuator, il est possible d'avoir l'état de santé de l'application à tout moment ; grâce à des URL, REST disponibles.
p26	Composition du fragment de page côté serveur	1	En utilisant le starter thymeleaf, on peut faire de la composition de fragments côté serveur et rendre une vue fonctionnelle.
p27	Composition de l'interface utilisateur côté client	0	Spring est uniquement serveur.
Total		h = 1, 26 fois & h = 0, 1 fois	

3.1.3 Calcul de degré de compatibilité

La fonction d'évaluation $f(t)$ d'une technologie est :

$$f(t) = e_1 \times h(t, p_1) + e_2 \times h(t, p_2) + \dots + e_q \times h(t, p_q)$$

Ainsi pour la technologie Spring Boot nous avons :

$$f(t) = 26, \text{ car } \forall i, e_i = 1 \text{ et } h \text{ a la } 1,26 \text{ fois dans le tableau ci-dessus.}$$

Ainsi Spring Boot est compatible avec l'architecture micro-service à **96,3 %**.

3.2 Évaluation de JAVA EE 7

Cette évaluation se fera sur la version 7, de la plateforme JAVA EE.

3.2.1 Présentation

JEE (Java Enterprise Edition) est une plateforme qui permet de faciliter le développement d'application d'entreprise en fournissant un environnement d'exécution et des composants sous forme d'API.

3.2.1.1 Architecture

La plateforme JAVA EE, propose une organisation du code, selon le modèle MVC. Le modèle MVC découpe littéralement l'application en couches distinctes et de ce fait impacte très fortement l'organisation du code. MVC signifie Modèle – Vue – Contrôleur, en gros quand on développe avec le MVC on segmente son code en trois parties ou couches, chaque couche ayant une fonction bien précise.

La couche Vue : C'est la partie de votre code qui s'occupera de la présentation des données à l'utilisateur, dans un format.

La couche Contrôleur : C'est la couche chargée de router les informations, elle va décider qui va récupérer l'information et la traiter. Elle gère les requêtes des utilisateurs et retourne une réponse avec l'aide de la couche Modèle et Vue.

La couche Modèle : C'est la partie de votre code qui exécute la logique métier de votre application.

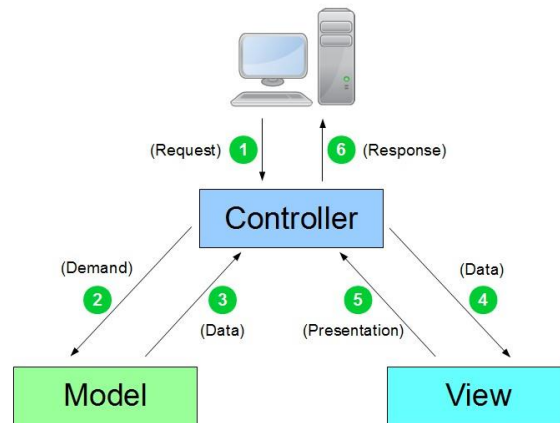


Figure 16 : Modèle architectural MVC [19]

Avec la plateforme JEE, chaque élément du modèle MVC porte en quelque sorte un nom. Le Contrôleur porte le nom de Servlet. Le modèle est en général géré par des objets Java ou des JavaBeans. Il peut être amené aussi à communiquer avec les bases de données pour stocker les informations, pour les persister et les garder en mémoire le plus longtemps possible. La Vue quant à elle est gérée par les pages JSP (Java Server Pages) qui sont en effet des pages qui vont utiliser du code HTML et du code spécifique en général en JAVA. Cette Vue est donc retournée au visiteur par le Contrôleur.

La servlet

En java EE une servlet est une classe Java ayant la capacité de permettre le traitement des requêtes et de personnaliser les réponses, c'est-à-dire qu'elle est capable de recevoir les requêtes HTTP envoyées depuis le navigateur (Client web) de l'utilisateur et de lui renvoyer une réponse http. Une servlet http doit toujours hériter de la classe HttpServlet qui est en effet une classe abstraite qui fournit des méthodes qui doivent être redéfinies dans la classe héritière. Parmi ces méthodes on y retrouve entre autres :

- ✚ doGet() : méthode qui permet de récupérer une ressource du serveur coté client (utilisé par le client) ;
- ✚ doPost() : c'est une méthode qui permet de soumettre au serveur des données de tailles variables, ou que l'on sait volumineuses.

La mise en place d'une Servlet se déroule en deux étapes : La définition de la servlet Il s'agit de déclarer notre servlet, c'est-à-dire lui donner un moyen d'être reconnu par le serveur. Ensuite faire le Mapping de la servlet, il s'agit ici de faire correspondre notre servlet déclaré précédemment à une URL afin qu'elle soit joignable par les clients.

Les JSP

Une page JSP est destinée à la vue (présentation des données). Elle est exécutée côté serveur et permet l'écriture des pages. C'est un document qui a première vue, ressemble beaucoup à une page HTML, mais qui en réalité diffère par plusieurs aspects :

- ✚ L'extension d'une telle page est .jsp et non .html ;
- ✚ Une page JSP contient des balises HTML mais aussi des balises JSP qui appellent de manière transparente du code JAVA ;
- ✚ Contrairement à une page HTML, une page JSP est exécutée côté serveur et génère alors une page renvoyée au client.

L'intérêt est de rendre possible la création des pages dynamiques. Avec une page JSP, on peut générer n'importe quel type de format, HTML, CSS, XML, du texte brut etc. Une page utilisant les JSP est exécutée au moment de la requête par un moteur de JSP fonctionnant généralement avec un serveur web ou un serveur d'application. Lorsqu'un utilisateur appelle une page JSP pour la première fois, le serveur web appelle le moteur JSP qui crée un programme Java à partir du script JSP, compile la classe afin de fournir un fichier compilé (extension .class). Au prochain appel, le moteur de JSP vérifie si la date du fichier .jsp correspond à celle du fichier .class, si elles sont identiques, le moteur de JSP n'effectuera aucune compilation et dans le cas contraire la compilation sera de nouveau faite. En d'autres termes le moteur JSP ne transforme et compile la classe que dans le cas où le script a été mis à jour. Ainsi le fait que la compilation ne se fasse que lors de mise à jour ou de la modification du script JSP fait de cette technologie une des plus rapides pour créer des pages dynamiques.

Les JavaBeans

Les Javabeans sont un modèle de composants du langage Java. Ce sont des “pièces logicielles” qui peuvent être réutilisées pour créer des programmes dans un environnement visuel de développement d'applications. Elles représentent généralement les données du monde réel. Les principaux concepts mis en jeu pour un Bean sont les suivants :

✚ **Propriétés** : Un JavaBean doit pouvoir être paramétrable. Les propriétés c'est-à-dire les champs non publics présent dans un bean. Qu'elles soient de type primitif ou objets, les propriétés permettent de paramétrer le bean, en y stockant des données.

✚ **La sérialisation** : Un bean est construit pour pouvoir être persistant. La sérialisation est un processus qui permet de sauvegarder l'état d'un bean et donne ainsi la possibilité de le

restaurer par la suite. Ce mécanisme permet une persistance de données voir de l'application elle-même.

✚ **La réutilisation** : Un bean est un composant conçu pour être réutilisable. Ne contenant que des données du code métier, un tel composant n'a en effet pas de lien direct avec la couche de présentation, et peut également être distant de la couche d'accès aux données. C'est cette indépendance qui lui donne ce caractère réutilisable.

✚ **L'introspection** : un bean est conçu pour être paramétrable de manière dynamique. L'introspection est un processus qui permet de connaître le contenu d'un composant (attributs, méthodes et événements) de manière dynamique, sans disposer de son code source. C'est ce processus, couplé à certaines règles de normalisation, qui rend possible une découverte et un paramétrage dynamique du bean.

Ainsi tout objet conforme à ces quelques règles peut être appelés Bean. Il est également important de noter qu'un JavaBean n'est pas un EJB (Entreprise Java Bean). Un bean doit également avoir la structure suivante :

- ✚ Doit être une classe publique ;
- ✚ Doit avoir au moins un constructeur par défaut, public et sans paramètres. Java l'ajoutera de lui-même si aucun constructeur n'est explicité ;
- ✚ Peut implémenter l'interface Serializable, il devient ainsi persistant et son état peut être sauvegardé ;
- ✚ Ne doit pas avoir de champs publics ;
- ✚ Peut définir des propriétés (des champs non publics), qui doivent être accessibles via des méthodes publiques getter et setter, suivant des règles de nommage.

3.2.1.2 Composants

Les applications Java EE sont constituées des composants. Un composant Java EE est une entité logicielle fonctionnelle autonome qui est assemblée dans une application Java EE avec ses classes et fichiers associés et qui communique avec d'autres composants. Le développement des applications JEE reposent sur un découpage en couche ou tiers (MVC), on parle alors d'applications multi-tiers. Trois grands tiers sont ainsi représentés : Modèle, Vue et Contrôleur dans lesquelles sont repartis les différents composants. La spécification Java EE définit les composants Java EE suivants :

Les composants Clients ou tiers Client qui sont exécutés côté client ;

Les composants Web ou tiers Web exécutés côté serveur (Moteur web) ;

Les composants métier ou tiers Métier exécutés côté serveur (Moteur d'application)

Composants clients

Les composants clients peuvent être des clients web ou des applications clientes.

Clients web

Les clients web sont composés de deux parties, des pages web dynamiques qui contenant différents types de langage de balise (HTML, XML, etc.), générées par des composants web s'exécutant dans la couche tiers Web et un navigateur web qui présente ou affiche les pages envoyées par le serveur.

Applets

Une page web reçu de la couche web tiers peut inclure un applet embarqué. Un applet est une petite application cliente écrit dans le langage de programmation Java qui s'exécute dans la machine virtuelle Java (JVM) installée dans le navigateur web. Toutefois, les systèmes clients auront probablement besoin du plug-in Java et éventuellement un fichier de règles de sécurité pour que l'applet puisse s'exécuter avec succès dans le navigateur web.

Applications Clientes

Une application cliente s'exécute sur un ordinateur client et permet aux utilisateurs de gérer des tâches nécessitant une interface utilisateur plus riche que celle fournie par un langage de balisage. Elle possède généralement une interface utilisateur graphique (GUI) créée à partir de l'API AWT (Swing ou Abstract Window Toolkit), mais une interface de ligne de commande est certainement possible. Les Applications clientes accèdent directement aux beans d'entreprise s'exécutant dans le niveau métier. Toutefois, si les exigences de l'application le justifient, un client d'application peut ouvrir une connexion HTTP pour établir une communication avec une servlet s'exécutant dans la couche Web tiers.

Les composants Web

Les composants Web Java EE sont des servlets ou des pages créés à l'aide de la technologie JSP (pages JSP) et / ou de la technologie JavaServer Faces. Les servlets sont des classes de langage de programmation Java qui traitent dynamiquement les requêtes et construisent des réponses. Les pages JSP sont des documents textuels qui s'exécutent en tant que servlets mais permettent une approche plus naturelle de la création de contenu statique. La technologie JavaServer Faces s'appuie sur les servlets et la technologie JSP et fournit une infrastructure de composants d'interface utilisateur pour les applications Web. Les pages HTML

statiques et les applets sont regroupés avec des composants Web lors de l'assemblage de l'application, mais ne sont pas considérés comme des composants Web par la spécification Java EE. Les classes d'utilitaires côté serveur peuvent également être regroupées avec des composants Web et, à l'instar des pages HTML, ne sont pas considérées comme des composants Web.

Les composants métiers

Le code métier, qui est la logique qui résout ou répond aux besoins d'un domaine d'activité particulier tel que la banque, le commerce de détail ou la finance, est géré par des beans entreprise s'exécutant dans côté métier. Un entreprise bean reçoit des données de programmes client, les traite (si nécessaire) et les envoie au niveau du système d'information d'entreprise pour le stockage, il récupère également les données du stockage, les traite (si nécessaire) et les renvoie au programme client.

On constate dans cette architecture qu'on dispose de tous les composants (Web tiers et Business Tiers) cités plus haut repartis sur les trois couches du modèle MVC. Cependant, les applications Java EE utilisent un serveur d'application situé entre le serveur Web et la couche base de données. Le Serveur d'application va servir à exécuter les composants métier (EJB) et constitue ce qu'on appelle un environnement d'exécution des composants métier (EJB) qu'on appelle Conteneur d'application qui fournira tout ce qui est nécessaire pour l'exécution des JavaBeans. Le conteneur Web quant à lui servira d'environnement d'exécution des JSP et servlet, etc. On obtient ainsi l'architecture suivante :

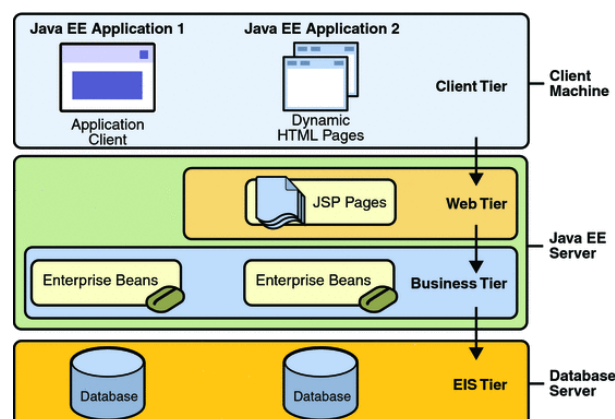


Figure 17 : Architecture d'application JAVA EE [19]

On a donc dans cette architecture un Serveur Java pour les composants Web ou Web Tiers et un Serveur Java EE pour les composants métier ou business Tiers.

3.2.1.3 Les conteneurs JAVA EE

L'architecture Java EE indépendante de ses composants et des plates-formes rend les applications Java EE facile à développer, en effet la logique métier est organisée en composants réutilisables. En outre, le serveur Java fournit des services sous-jacents sous la forme d'un conteneur pour chaque type de composant. Les conteneurs fournissent donc un support ou environnement d'exécution pour les composants des applications Java EE. Ils fournissent une vue fédérée des API Java EE sous-jacentes aux composants de l'application. Les composants d'application Java EE n'interagissent pas directement avec d'autres composants d'applications Java EE. Ils utilisent les protocoles et les méthodes du conteneur pour interagir entre eux avec les services de la plate-forme. Nous avons précédemment vu que les applications JEE utilisaient un serveur d'application et un serveur web, en réalité, il s'agit d'un seul serveur appelé Serveur JAVA (Java Server) qui propose plusieurs types de conteneurs (containers) ou moteur pour chaque type de composant. Le conteneur web qui constitue l'environnement d'exécution des servlets, des pages JSP (JavaServer Pages) et JSF (JavaServerFrame) et le conteneur EJB quant à lui constitue l'environnement d'exécution des Entreprises Java Bean (EJB). Chaque conteneur a une fonction bien définie et offre aux développeurs un ensemble de services tels que :

- L'annuaire de nommage d'accès aux ressources : Java Naming and Directory Interface (JNDI) qui est une interface unifiée de gestion de nommage pour les services et l'accès à ces derniers par les applications ;

- L'injection dynamique des ressources ;

- La gestion des accès aux bases de données ;

- Le modèle de gestion de la sécurité ;

- Le paramétrage des transactions ;

- Les connexions distantes.

L'exécution et le développement d'application JEE sont donc directement liés au conteneur utilisé. C'est-à-dire qu'une application JEE Web Tiers utilisera uniquement le conteneur Web pour son développement et de même une application JEE Business tiers utilisera uniquement le conteneur d'application pour son développement et son exécution.

3.2.1.4 Les API JAVA EE

Cette plateforme fournit essentiellement un environnement d'exécution et un ensemble de services accessibles via des APIs pour aider à concevoir nos applications. Ainsi donc, tout serveur JEE va fournir à nos applications un ensemble de services comme la connexion aux

bases de données, la messagerie, la gestion des transactions, etc. L'architecture JEE unifie l'accès à ces services au sein des API de services d'entreprise, mais plutôt que d'avoir accès à ces services à travers des interfaces propriétaires, les applications JEE peuvent accéder à ces API via le serveur.

La spécification de la plateforme J2EE prévoit un ensemble d'extensions Java standard que chaque plate-forme JEE doit prendre en charge :

JNDI : Java Naming and Directory Interface permet de localiser et d'utiliser les ressources. C'est une extension JAVA standard qui fournit un API uniforme permettant d'accéder à divers services de noms et de répertoires. Derrière JNDI il peut avoir un appel à des services tels que : CORBA, DNS, LDAP, etc.

JDBC : Java Database Connectivity est une API qui permet aux programmes JAVA de se connecter et d'interagir avec les bases de données SQL ;

JMS : Java Message Service est à la fois une ossature et une API permettant aux développeurs de construire des applications professionnelles qui se servent de messages pour transmettre des données ;

JTA : Java Transaction API définit des interfaces standards entre un gestionnaire de transactions et les éléments impliqués dans celle-ci : l'application, le gestionnaire de ressource et le serveur. JSP : Java ServerPage (Encore les JSP) c'est une extension de la notion de servlet permettant de simplifier la génération de pages web dynamiques. Elle se sert de balises semblables au XML ainsi que de scriptlets Java afin d'incorporer la logique de fabrication directement dans le code HTML ;

JSP : est un concurrent direct de l'ASP et du PHP ;

Servlet : Un servlet est un composant coté serveur, écrit en Java, dont le rôle est de fournir une trame générale pour l'implémentation de paradigmes " requête-réponse ". Ils remplacent les scripts CGI tout en apportant des performances bien supérieures ;

EJB : Chaque instance d'un EJB se construit dans un conteneur EJB, un environnement d'exécution fournissant des services (Sécurité, Communications, Cycle de vie...). Un client n'accède JAMAIS directement à un composant. Il doit pour cela passer par une interface locale et une interface distance. L'interface locale décrit le cycle d'existence du composant en définissant des méthodes permettant de le trouver, de le créer, de le détruire. Et L'interface distante spécifie les méthodes que ce composant présente au monde extérieur ;

Authentication : J2EE fournit des services d'authentification en se basant sur les concepts d'utilisateur, de domaines et de groupes.

3.2.2 Recherche des valeurs de la fonction h

Tableau 4 : Tableau des valeurs de la fonction h, pour la technologie JAVA EE 7

	Critères d'évaluation	h	Justification
p1	Base de données par service	1	Les développeurs de SGBD, fournissent les driver nécessaire pour la connexion à leurs serveurs. Le JAVA étant très populaire, ces drivers sont disponibles.
p2	Composition d'API	0	Pas d'implémentation disponible
p3	SAGA	0	Pas d'implémentation disponible
p4	Événement de domaine	1	Les événements sont gérés grâce au service ActiveMQ dont le driver est disponible.
p5	Sourcing événementiel	1	Les événements sont gérés grâce au service ActiveMQ dont le driver est disponible.
p6	Test des composants de service	0	Pas d'implémentation disponible
p7	Test de contrat d'intégration de services	0	Pas d'implémentation disponible
p8	Plusieurs instances de service par hôte	0	Le déploiement se fait dans un serveur d'application et une seule instance du serveur peut être lancée. De plus l'application tourne sur un port.
p9	Instance de service par conteneur	1	Une application peut être lancée dans un conteneur.
p10	Déploiement sans serveur	1	Une application JAVA EE, peut être déployée sans serveur. Car toutes les dépendances peuvent être chargées sur un dépôt.
p11	Configuration externalisée	0	Pas d'implémentation disponible
p12	Appel de procédure à distance	1	Les techniques d'appel de procédure à distance native de JAVA sont disponibles.
p13	Messagerie	1	JAVA EE, dispose d'une API, JMS pour la communication par message.
p14	Passerelle API	1	JAVA EE, permet l'implémentation d'API REST.
p15	Backends pour frontends	0	Pas d'implémentation disponible
p16	Découverte de service côté client	0	Pas d'implémentation disponible
p17	Découverte de service côté serveur	0	Pas d'implémentation disponible
p18	Registre des services	0	Pas d'implémentation disponible

p19	Auto-inscription	0	Pas d'implémentation disponible
p20	Disjoncteur	0	Pas d'implémentation disponible
p21	Jeton d'accès	1	La sécurisation d'API par token est disponible.
p22	Agrégation de journaux	0	Pas d'implémentation disponible
p23	Mesures d'application	1	Le contrôle de l'application est effectué au travers du serveur d'application.
p24	Traçage distribué	0	Pas d'implémentation disponible
p25	Vérification de la santé de l'API	1	Le contrôle de l'application est effectué au travers du serveur d'application.
p26	Composition du fragment de page côté serveur	1	Grâce aux JSP, on peut faire de la composition de fragments côté serveur et rendre une vue fonctionnelle.
p27	Composition de l'interface utilisateur côté client	0	JAVA EE, est uniquement serveur.
Total		h = 1, 12 fois & h = 0, 15 fois	

3.2.3 Calcul de degré de compatibilité

La fonction d'évaluation $f(t)$ d'une technologie est :

$$f(t) = e_1 \times h(t, p_1) + e_2 \times h(t, p_2) + \dots + e_q \times h(t, p_q)$$

Ainsi pour la technologie JAVA EE 7 nous avons :

$$f(t) = 12, \text{ car } \forall i, e_i = 1 \text{ et } h \text{ a la } 1, 12 \text{ fois dans le tableau ci-dessus.}$$

Ainsi JAVA EE est compatible avec l'architecture micro-service à **44,4 %**.

3.3 Interprétation

Pour la plateforme Spring Boot, le score obtenu est de 96,3% de compatibilité. Il s'explique d'une part par le fait qu'il s'agit d'un framework basé sur un langage très populaire, très riche, ayant une grande maturité et une importante communauté, d'autre part la conception même de ce framework le rend capable d'évoluer rapidement et d'intégrer de nouveaux packages (starter) qui se configurent automatiquement mais qu'on peut aussi configurer à souhait. Ceux-ci apportent une très grande simplicité dans le développement et couvrent une large gamme de besoins.

Pour la plateforme JAVA EE, le score obtenu est de 44,4% de compatibilité. Toutefois ce chiffre peut légèrement varier selon le serveur d'application utilisé, qui peut offrir des services supplémentaires. Ce score dénote une incompatibilité quasi-totale avec le style micro-

service. Cette incompatibilité s'explique par la conception même de la plateforme. En effet, la plateforme est conçue dans un style purement SOA et donc n'aborde aucune problématique introduite par le style micro-service notamment les problématiques majeurs telles que la gestion distribuée des données, la découverte des services, la composition d'API, etc.

Au terme de cette évaluation, un constat s'impose : nous avons évalué deux frameworks tous deux basés sur le langage JAVA mais fort est de constater l'important écart de score obtenu par les deux. Tandis que l'un est très compatible, l'autre ne l'est quasiment pas. Cet écart, peut s'expliquer de plusieurs manières :

La conception des deux frameworks est très différente : en effet, JAVA EE est conçu dans une logique purement SOA, figée, nécessitant beaucoup de configurations. Spring Boot par contre, permet de créer l'application voulue (SOA, micro-service, REST API, ligne de commande) juste en intégrant le starter correspondant ; celui-ci ajoute toutes les dépendances nécessaires et la configuration pour immédiatement démarrer ;

La communauté de Spring Boot est plus importante que celle de JAVA EE : tandis que les spécifications de JAVA EE viennent d'Oracle, les starters développés par la communauté Spring Boot peuvent être intégrés au projet officiel, ce qui permet d'avoir des starters abordant presque toutes les problématiques. C'est le cas de Netflix, qui est l'un des pionniers dans le domaine des architectures micro-services et qui a produit de nombreux starters dédiés à style ;

La facilité de prise en main : En effet grâce à son système d'auto-configuration, le développement et le déploiement d'une application Spring Boot ne nécessite presque aucune configuration, ni un quelconque serveur, tous les éléments se trouvent dans le fichier `.jar` issu de la compilation ; tandis que pour JAVA EE, la configuration est manuelle, fastidieuse et le déploiement oblige la présence d'un serveur d'application préalablement installé.

Conclusion

Dans ce chapitre, il a été question de réaliser une évaluation de la mise en œuvre du style architectural micro-service avec les technologies Spring Boot et JAVA EE. Pour accomplir cette évaluation, nous nous appuyer sur le modèle d'évaluation conçu au chapitre 2. Pour cela nous posons trois hypothèses préalables que sont : l'hypothèse 1 sur la taille du vecteur d'exigences ($q = 27$) qui permet le paramétrage de la fonction d'évaluation ;

l'hypothèse 2 sur l'égale importance des exigences e_i ($e_i = 1$) ; l'hypothèse 3 sur le calcul de la valeur de la fonction h .

Ces hypothèses ayant fixées le cadre et le déroulement des évaluations, les deux parties suivantes ont été dédiées à l'évaluation des technologies. En premier Spring Boot version 2.2.2, qui au terme de l'évaluation a obtenu une note de 26 / 27, soit une compatibilité à 96,3 %. Ce qui permet de dire de Spring Boot, qu'il est l'outil idéal pour l'implémentation d'une architecture micro-service côté serveur. Quant à JAVA EE, il a obtenu une note de 12 / 27 soit une compatibilité à 44,4 %, soit une incompatibilité totale avec le style micro-service, car le cœur même du style n'est pas abordé.

CONCLUSION GENERALE ET PERSPECTIVES

Bilan

La réflexion menée dans ce mémoire portait sur l'évaluation de la mise en œuvre d'une architecture micro-service avec les plateformes Spring Boot et JAVA EE, qui sont des plateformes opensource et très populaire. L'approche adoptée est basée sur la construction d'un modèle d'évaluation.

Dans un premier temps, il a fallu s'appropriier les spécificités du style architectural micro-service et des technologies Spring Boot et JAVA EE. Ensuite les connaissances issues de cette revue de la littérature, nous ont permis d'engager la construction de notre modèle d'évaluation, qui est fait d'une part d'une liste de critères à satisfaire : il s'agit d'un ensemble de patrons conceptions qui couvrent tous les domaines possibles d'une application micro-service à vérifier dans une technologie et d'autre part d'une fonction d'évaluation qui attribue une note à une technologie suivant les exigences de l'application développée.

En appliquant le modèle à ces deux technologies sous les hypothèses que : (1) le vecteur d'exigences qui permet le paramétrage de la fonction d'évaluation du modèle, correspondra aux 27 critères formant le modèle d'évaluation ; (2) toutes les exigences e_i ont le même niveau d'importance égale à 1 ; (3) la valeur de la fonction h , est obtenu en vérifiant si dans l'univers des packages officielles de la technologie étudiée, il existe un package qui implémente le critère passé en paramètre. celles-ci ont obtenu des scores très différents exprimant leur niveau de compatibilité avec le style micro-service 96,3% pour Spring Boot et 44,4% pour JAVA EE. L'intérêt de ce travail est donc d'offrir à la communauté scientifique un outil d'aide à la décision dans le choix des technologies pour implémenter application style micro-service.

Perspectives

Les orientations futures de ce travail vont vers deux axes principaux.

- ✚ Le premier axe est celui de la construction d'un benchmark, qui ferai une classification des technologies existantes selon la valeur retournée par la fonction d'évaluation ;
- ✚ Le deuxième axe de réflexion est celui de la conception et de l'implémentation d'un outil capable de faire l'évaluation d'une application existante.

REFERENCES BIBLIOGRAPHIQUES

- [1] NÉO-SOFT Solutions. « SPRING BOOT, késako ? », 7 avril 2016. <http://www.neo-soft-solutions.fr/spring-boot-kesako/>.
- [2] « Software Architecture ». In *Wikipedia*, 10 janvier 2019. https://en.wikipedia.org/w/index.php?title=Software_architecture&oldid=877799959.
- [3] Kruchten P., Obbink H., Stafford J. : The past, present, and future for software architecture. *IEEE Softw.* 23, 2 (2006), 22–30.
- [4] Garlan D., Shaw M. : An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering* (Singapore, 1993), Ambriola V., Tortora G., (Eds.), World Scientific Publishing Company, pp. 1–39.
- [5] Le Goaer, Olivier. « Styles d'évolution dans les architectures logicielles ». Thèse, Université de Nantes, 2009. <https://tel.archives-ouvertes.fr/tel-00459925>.
- [6] Medvidovic N., Taylor R. N. : «A classification and comparison framework for software architecture description languages». *IEEE Trans. Softw. Eng.* 26, 1 (2000), 70–93.
- [7] MOO MENA, Francisco José. « MODÉLISATION DES ARCHITECTURES LOGICIELLES DYNAMIQUES ». Thèse, Institut National Polytechnique de Toulouse, 2007. <https://tel.archives-ouvertes.fr/tel-00142298/document>.
- [8] Namiot, Dmitry, et Manfred Sneps-Snepp. « On Micro-services Architecture ». *International Journal of Open Information Technologies* 2, n° 9 (2014): 4.
- [9] microservices.io. « Microservices Pattern: Microservice Architecture pattern ». Consulté le 13 janvier 2019. <http://microservices.io/patterns/microservices.html>.
- [10] J. Lewis and M. Fowler, « Microservices » [En ligne]. Disponible sur: <https://martinfowler.com/articles/microservices.html>, 2014
- [11] « Spring Cloud ». Consulté le 9 décembre 2019. <https://spring.io/projects/spring-cloud>.
- [12] « Qu'est-ce que l'architecture Microservices ? | SUPINFO, École Supérieure d'Informatique ». Consulté le 13 février 2019. <https://www.supinfo.com/articles/single/5676-qu-est-ce-que-architecture-microservices>.
- [13] Kistowski, Jóakim v., Simon Eismann, Norbert Schmitt, André Bauer, Johannes Grohmann, et Samuel Kounev. « TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research ». *University of Würzburg*, s. d., 14.
- [14] « Patron de conception ». In *Wikipédia*, 28 octobre 2019. https://fr.wikipedia.org/w/index.php?title=Patron_de_conception&oldid=163947295.
- [15] Gampa Shreelekhy, Yazhini, et Senthilkumaran U. « METHODS FOR EVALUATING SOFTWARE ARCHITECTURE-A SURVEY ». *International Journal of Pharmacy & Technology*, 2 novembre 2016. <https://www.researchgate.net/publication/316887447>.
- [16] « Moving to Microservices: Top 5 Languages to Choose From ». Consulté le 24 septembre 2019. <https://rubygarage.org/blog/top-languages-for-microservices>.
- [17] « how-to-build-and-scale-with-microservices-french.pdf ». Consulté le 6 mars 2019. <https://cloud.kapostcontent.net/pub/93f35b9e-ca51-4fc9-8958-ad7cc5ccc289/how-to-build-and-scale-with-microservices-french.pdf>.

- [18] « GitHub - hantsy/spring-microservice-sample: Spring Boot based Microservice sample ». Consulté le 24 février 2019. <https://github.com/hantsy/spring-microservice-sample>.
- [19] « Model–View–Controller ». In Wikipedia, 15 décembre 2019. <https://en.wikipedia.org/w/index.php?title=Model%E2%80%93view%E2%80%93controller&oldid=930816340>.
- [20] Akentev Evgenii, Alexander Tchitchigin, Larisa Safina, et Manuel Mazzara. « Verified type checker for Jolie programming language ». *Innopolis University*, 23 mai 2017. <http://arxiv.org/abs/1703.05186v2>.
- [21] « Architectural pattern ». In Wikipedia, 14 janvier 2019. https://en.wikipedia.org/w/index.php?title=Architectural_pattern&oldid=878329033.
- [22] Belkhatir, Riad. « Contribution à l’automatisation et à l’évaluation des architectures logicielles ouvertes ». Thèse, Université de Nantes, 2014. <https://hal.archives-ouvertes.fr/tel-01146336>.
- [23] Belkhatir, Riad, Mourad Chabane Oussalah, et Arnaud Viguier. « SOAQE - Service Oriented Architecture Quality Evaluation ». *International Conference on Evaluation of Novel Approaches to Software Engineering*. juin 2012. <https://hal.archives-ouvertes.fr/hal-00991591>.
- [24] « Benchmark Requirements for Microservices Architecture Research ». *Conference Paper*, ResearchGate, mai 2017. <https://doi.org/10.1109/ECASE.2017.4>.
- [25] Dragoni, Nicola, Saverio Giallorenzo, Alberto Lafuente, et Manuel Mazzara. « Microservices: Yesterday, Today, and Tomorrow », Springer, 9 novembre 2017. <https://hal.inria.fr/hal-01631455/document>.
- [26] Guidi, Claudio, Ivan Lanese, Manuel Mazzara, et Fabrizio Montesi. « Microservices: a Language-based Approach », 26 avril 2017. <http://arxiv.org/abs/1704.08073v1>.
- [27] « Ingénierie logicielle à base de composants - Wikipedia ». Consulté le 7 février 2019. https://en.wikipedia.org/wiki/Component-based_software_engineering.
- [28] « Architecture des applications JAVA EE | SUPINFO, École Supérieure d’Informatique ». Consulté le 16 décembre 2019. <https://www.supinfo.com/articles/single/6628-architecture-applications-java-ee>.
- [29] M. Parizi, Reza. « Microservices as an Evolutionary Architecture of Component-Based Development: A Think-aloud Study ». *Department of Software Engineering and Game Development Kennesaw State University, Marietta, GA 30060, USA*, 30 mai 2018, 12.
- [30] « Software Architecture ». In Wikipedia, 10 janvier 2019. https://en.wikipedia.org/w/index.php?title=Software_architecture&oldid=877799959.
- [31] « Spring Cloud Netflix ». Consulté le 9 décembre 2019. <https://spring.io/projects/spring-cloud-netflix>.
- [32] « What is Microservices Architecture? » Consulté le 13 janvier 2019. <https://smartbear.com/learn/api-design/what-are-microservices/>.
- [33] « What’s a (micro)service - part 1? » Consulté le 4 octobre 2019. <http://chrisrichardson.net/post/microservices/general/2019/02/16/whats-a-service-part-1.html>.

- [34] Xiang, Zhou, Peng Xin, Xie Tao, et Sun Jun. « Benchmarking Microservice Systems for Software Engineering Research », mai 2017.
<https://www.researchgate.net/publication/314114327>.
- [35] Pressman R. S., « Software Engineering: A Practitioner's Approach », Fifth Edition. McGraw-Hill. Chapitre 1, p. 8, 2001.
- [36] «SEI's definitions of software architecture.» [En ligne]. Disponible sur:
<http://www.sei.cmu.edu/architecture/definitions.html>
- [37] Shaw M. : «Comparing architectural design styles». IEEE Softw. 12, 6 (1995), 27–41.
- [38] Gonclaves, Antonio. Beginning JAVA EE 7. Apress. The expert's voice in JAVA, 2013. Livre.
- [39] Richardson, Chris. Microservices Patterns. Manning Publications Co. S HELTER I SLAND, 2019. Livre.