CONSERVATOIRE NATIONAL DES ARTS ET MÉTIERS

PARIS

MÉMOIRE DE RECHERCHE

Spécialité : Informatique

Option : Systèmes d'information

Par

MONDELICE Romain

Transformation numérique des organisations : Les APIs et leurs méthodes d'utilisation

---

Soutenu le 21 juin 2021

Jury :

Président :

Monsieur DU MOUZA Cédric (CNAM)

Membres :

Monsieur ABOUCHAKRA Éric (CNAM)

Monsieur DETAMPLE Didier (Crédit Agricole Group Infrastructure Platform)

CFA AFIA

Transformation numérique des organisations : Les APIs et leurs méthodes d'utilisation.

Les technologies numériques et les solutions connexes changent toutes les dimensions de notre vie. Nous vivons une grande transformation numérique.

Parce que la transformation numérique aura un aspect différent pour chaque entreprise, il peut être difficile de mettre le doigt sur une définition qui s'applique à tous. Toutefois, de manière générale, nous définissons la transformation numérique comme l'intégration de la technologie numérique dans tous les domaines d'une entreprise, ce qui entraîne des changements fondamentaux dans le mode de fonctionnement et dans la manière dont elles apportent de la valeur à leurs collaborateurs ou à leurs clients. Au-delà de cela, il s'agit d'un changement culturel qui exige qu'elles remettent continuellement en question la conjoncture. Cela signifie parfois qu'il faut abandonner les processus de longue date sur lesquels les entreprises se sont construites en faveur de pratiques relativement nouvelles qui sont encore en cours de définition.

L'utilisation des *APIs* fait partie de ces nouvelles pratiques. Dans ce mémoire de recherche, nous allons donc retrouver toutes les clés qui permettent de prendre des décisions cruciales sur leurs architectures et leurs méthodes d'utilisation au sein d'une entreprise.

Les *APIs* sont une approche flexible et légère qui peut être utilisée par une entreprise pour fournir des fonctionnalités de programmation de logiciels à des applications internes et tierces.

Le but de ce mémoire de recherche est d'apporter les clés qui permettent de déterminer quelle méthode d'utilisation de l'*API* convient le mieux à une situation donnée.

Pour ce faire, un rappel sur la portée du mémoire sera effectué. Ensuite, les termes principaux qui seront utilisés tout au long de celui-ci seront définis. Une fois que la portée et les définitions sont acquises, il sera possible de mettre en exergue les spécifications et les fonctionnements techniques des différentes méthodes d'utilisation d'une *API.* À la suite de cette partie, le lecteur sera déjà en mesure de savoir quelle méthode d'utilisation est la plus appropriée pour sa situation. Il sera ensuite capable de définir dans quel niveau de maturité se trouvera l'*API* qu'il développera grâce à l'aide de la partie sur les modèles de maturité.

La suite du mémoire apporte une vision plus globale. On y trouvera une partie sur l'adoption des *APIs*. Ensuite, une partie concernant *CA-GIP* (*Crédit Agricole Group Infrastructure Platform*) sera disponible. Le lecteur prendra connaissance de l'impact de la transition numérique sur l'organisation et les méthodes d'utilisations d'*API* qui y sont mises en place.

À la fin de ce document, on pourra trouver un bilan et une conclusion qui comprend des conseils d'utilisation pour *CA-GIP* (*Crédit Agricole Group Infrastructure Platform*) résultant de l'analyse effectuée lors de l'élaboration de ce mémoire de recherche.

APIs are a flexible and lightweight approach that can be used by an enterprise to provide software programming functionality to internal and third-party applications.

The goal of this research paper is to provide the keys to determine which *API* usage method is best suited for a given situation.

To do this, a reminder of the scope of the dissertation will be provided. Then, the main terms that will be used throughout the brief will be defined. Once the scope and definitions are acquired, it will be possible to highlight the specifications and technical workings of the different methods of using an *API*. After this part, the reader will already know which method of use is the most appropriate for his situation. He will then be able to define the maturity level of the *API* he will develop with the help of the section on maturity models.

The rest of the thesis brings a more global vision. There is a section on the adoption of *APIs*. Then, a part concerning *CA-GIP* (*Crédit Agricole Group Infrastructure Platform*) will be available. The reader will learn about the impact of the digital transition on the organization and the methods of using APIs that are put in place.

At the end of this document, one will find a summary and a conclusion that includes usage tips for *CA-GIP* (*Crédit Agricole Group Infrastructure Platform*) resulting from the analysis performed during the elaboration of this research paper.

Liste de 10 mots clés:

API

RFC

RPC

XML-RPC

JSON-RPC

REST

GraphQL

Modèles de maturité

Amundsen

Richardson

1. Introduction 6
1.1. Portée du mémoire de recherche 6
1.2. Définitions 6
1.2.1. Application Programming Interfaces 6
1.2.2. Service Web 6
2. Méthodes d'utilisation 7
2.1. Remote procedure call – RPC 7
2.1.1. Encodage XML-RPC 7
2.1.1.1. XML 7
2.1.1.3. XML-RPC 10
2.1.2. Encodage JSON-RPC 11
2.1.2.1. JSON 11
2.1.2.2. JSON-RPC 14
2.2. Representational state transfer – REST 18
2.2.1. Client–server architecture 19
2.2.2. Statelessness 20
2.2.3. Cacheability 20
2.2.4. Layered system 20
2.2.5. Code on demand 21
2.2.6. Uniform interface 21
2.3. GraphQL 21
2.4. Etude comparative 24
3. Modèles de maturité 26
3.1. Le modèle de maturité d'Amundsen 26
3.1.1. AMM0: Amundsen niveau 0 27
3.1.2. AMM1: Amundsen niveau 1 27
3.1.3. AMM2: Amundsen niveau 2 27
3.1.4. AMM3: Amundsen niveau 3 27
3.1.5. Niveaux du modèle de maturité d'Amundsen en exemples 27
3.2. Le modèle de maturité de Richardson 28
3.2.1. RMM0: Richardson niveau 0 29
3.2.2. RMM1: Richardson niveau 1 29
3.2.3. RMM2: Richardson niveau 2 30
3.2.4. RMM3: Richardson niveau 3 31
4. Adoption des APIs 32
5. APIs et CA-GIP 33
5.1. Impact sur l'organisation : conduite du changement 34
5.2. Utilisation des APIs 36
6. Bilan et conclusion 36
7. Table des figures 38

8. Sources 39
8.1. Sitographie 39
8.2. Vidéographie 40
8.3. Bibliographie 40

1. Introduction

1.1. Portée du mémoire de recherche

En raison de l'extrême variabilité et de l'évolution rapide des *TIC (Technologies de l'information et de la communication)* et des technologies web, notamment dans le paysage des *API*, toute liste d'*API* existantes dans les secteurs concernés ou en relation avec les thèmes pertinents pour un quelconque domaine serait très probablement rapidement obsolète. Ainsi, dans ce mémoire, plutôt que de se concentrer sur les tendances technologiques actuelles, telles que celles décrites par l'*OGC* (*Open Geospatial Consortium,* https://www.ogc.org), l'objectif est de proposer des normes générales et des spécifications techniques pertinentes afin de fournir des connaissances permettant de prendre des décisions importantes concernant les méthodes d'utilisation d'une *API*.

1.2. Définitions

Cette section illustre les principaux concepts qui doivent être pris en compte lors de l'évocation des *APIs*. Une définition additionnelle d'un *service web* se trouvera à la fin de cette section pour permettre d'éclaircir ce qui le diffère d'une *API* et ainsi d'améliorer la compréhension globale du sujet.

1.2.1. Application Programming Interfaces

Le concept d'*API* n'est pas nouveau. Il est probablement apparu pour la première fois en 1968, défini comme « une collection de routines de code qui fournissent aux utilisateurs externes des données et des fonctionnalités de données » (Cotton et Greatorex, 1968). Parce que les *API* sont des solutions technologiques générales, elles peuvent être utilisées à de nombreuses fins. Nous pouvons donc adopter une explication plus récente et plus étendue des *API*, qui les définit comme « les appels, les sous-programmes ou les interruptions logicielles qui constituent une interface documentée permettant à un programme d'application d'utiliser les services et les fonctions d'une autre application, d'un système d'exploitation, de réseau, d'un pilote ou d'un autre programme logiciel de niveau inférieur » (Shnier, 1996). Du point de vue de l'ingénierie logicielle, les *APIs* constituent les interfaces des différents blocs de construction qu'un développeur peut assembler pour créer une application.

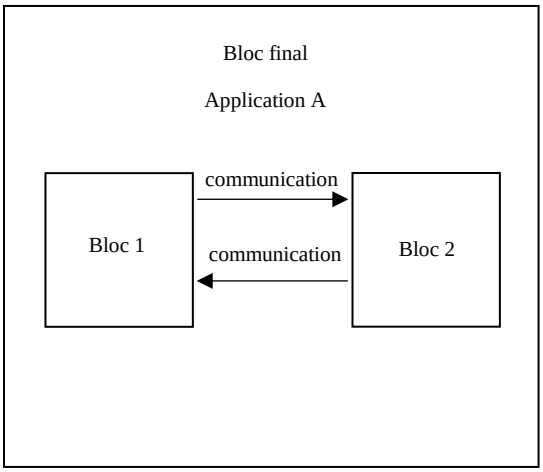

Figure 1 : Démonstration conceptuelle de l'architecture d'une application utilisant une API

Un développeur d'applications utilise les *API* pour créer une application en combinant diverses bibliothèques logicielles disponibles pour atteindre un objectif spécifique. Si la notion d'interfaces programmatiques en tant que collection de méthodes exportées par une certaine bibliothèque de code n'est pas nouvelle, avec l'avènement du web, et en particulier du web 2.0, la notion d'*API* web a été introduite pour désigner ces *API* fonctionnant sur le web. Les *API* Web sont utilisées pour fournir aux développeurs les blocs de construction nécessaires à la création d'applications logicielles basées sur le Web.

1.2.2. Service Web

Il existe plusieurs définitions d'un service Web. Le *W3C* définit un service Web comme : « un système logiciel conçu pour prendre en charge une interaction interopérable de machine à machine sur un réseau. Il possède une interface décrite dans un format exploitable par une machine (notamment *WSDL - Web Service Description Language*). » (*W3C*, 2004). D'autres fournissent des définitions plus génériques, par exemple (*IBM*, 2014) indique qu'un « Service Web est un terme générique pour une fonction logicielle interopérable de machine à machine qui est hébergée à un emplacement adressable par le réseau ». Le modèle de référence *OASIS* pour l'architecture orientée service définit un service Web comme « un mécanisme permettant d'accéder à une ou plusieurs capacités, dans lequel l'accès est fourni à l'aide d'une interface prescrite et d'un système de gestion de la qualité ».

Si les définitions génériques présentées ci-dessus généralisent la définition restrictive et axée sur la technologie du *W3C*, elles ne clarifient pas la différence entre une interface de service et une interface de programmation : la première est fournie par un service Web, la seconde est une caractéristique distincte d'une *API*. Les interfaces de services Web sont conçues pour offrir un accès à des fonctionnalités de « haut niveau » aux utilisateurs finaux, qu'il s'agisse d'humains ou de machines. En revanche, les *API* sont conçues pour fournir des fonctionnalités, même de « bas niveau », en tant que blocs de construction pouvant être utilisés et combinés par les développeurs de logiciels pour fournir un service de haut niveau.

2. Méthodes d'utilisation

Dans cette section nous allons expliquer les différentes méthodes d'utilisation d'une *API*.

2.1. Remote procedure call – RPC

L'appel de procédure à distance *RPC* (*Remote procedure call*) est un protocole défini dans le *RFC 1057* (*Request for comments 1057*, https://datatracker.ietf.org/doc/html/rfc1057) qu'un programme peut utiliser pour demander un service à un programme situé sur un autre ordinateur du réseau dont il n'a pas besoin de connaître les détails. Il est parfois appelé appel de fonction ou appel de sous-routine. *RPC* est basé sur le modèle client-serveur. Le programme demandeur est le client et le programme qui fournit le service est le serveur. Comme un appel de procédure locale ou habituelle, un appel de procédure distante est une opération synchrone : le programme demandeur attend la fin du traitement de la procédure distante pour reprendre. Cependant, l'utilisation de threads ou de processus légers avec un espace d'adressage commun permet l'exécution simultanée de plusieurs appels de procédures distantes. L'idée de *RPC* date d'au moins 1976, quand il a été décrit dans le *RFC 707* (*Request for comments 707*, https://datatracker.ietf.org/doc/html/rfc707). L'une des premières utilisations commerciales de *RPC* a été faite par la société Xerox sous le nom de Courier en 1981. La première implémentation populaire de *RPC* sur Unix était le *RPC* de Sun (maintenant appelée *ONC RPC*), qui a été utilisé comme la base pour *NFS* (*Network File System*).

Dans le cas du RPC, notre API serrait le serveur. Notre client enverrait une procédure avec un argument, et notre API (serveur) renverrait le résultat.

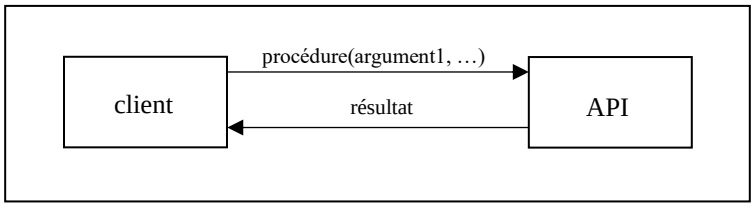

Figure 2 : Démonstration conceptuelle de l'utilisation du protocole RPC

Un RPC est initié par le client qui envoie un message de demande à un serveur distant connu pour exécuter une procédure spécifique avec des paramètres spécifiques. Le serveur distant envoie une réponse au client, puis l'application continue de s'exécuter. Pendant que le serveur traite l'appel, le client est bloqué (il attend que le serveur ait fini de traiter les données). Une différence importante entre les appels de procédure locaux et distants est que les appels distants peuvent échouer en raison de problèmes de réseau imprévisibles. De plus, les appelants doivent généralement gérer ces échecs sans savoir si la procédure distante a effectivement été invoquée. Les procédures idempotentes (c'est-à-dire celles qui n'ont pas d'effet supplémentaire si elles sont appelées plus d'une fois) sont faciles à gérer, mais il reste de nombreuses difficultés, qui font que le code d'appel de procédure à distance est souvent confiné à des sous-systèmes de bas niveau soigneusement écrits.

Il existe de nombreuses variations et subtilités dans les diverses implémentations, ce qui donne lieu à une variété de protocoles *RPC* (incompatibles). Les plus utilisés sont *XML-RPC* et *JSON-RPC*.

2.1.1. Encodage XML-RPC

XML-RPC est donc un protocole d'appel de procédure à distance (*RPC*) qui utilise *XML* pour coder ses appels et *HTTP* comme mécanisme de transport.

2.1.1.1. XML

XML est un profil d'application de *SGML* (*Standard Generalized Markup Language*, ISO 8879).

La polyvalence de *SGML* pour l'affichage dynamique d'informations a été comprise par les premiers éditeurs de médias numériques à la fin des années 1980, avant l'essor d'Internet. Au milieu des années 1990, certains praticiens de *SGML* avaient acquis de l'expérience avec le World Wide Web de l'époque et pensaient que *SGML*

offrait des solutions à certains des problèmes que le Web était susceptible de rencontrer au fur et à mesure de son développement. Dan Connolly a ajouté *SGML* à la liste des activités du *W3C* lorsqu'il a rejoint le personnel en 1995 ; le travail a commencé à la mi-1996 lorsque Jon Bosak, ingénieur chez Sun Microsystems, a élaboré une charte et recruté des collaborateurs. Bosak avait de bonnes relations dans la petite communauté des personnes ayant une expérience à la fois du *SGML* et du Web.

XML a été compilé par un groupe de travail de onze membres, soutenu par un groupe d'intérêt (d'environ) 150 membres. Le débat technique avait lieu sur la liste de diffusion du groupe d'intérêt et les problèmes étaient résolus par consensus ou, en cas d'échec, par un vote majoritaire du groupe de travail. Un registre des décisions de conception et de leurs justifications a été compilé par Michael Sperberg-McQueen le 4 décembre 1997. James Clark a été le responsable technique du groupe de travail, contribuant notamment à la syntaxe empty-element <empty /> et au nom *XML*. D'autres noms avaient été proposés pour examen, notamment *MAGMA* (*Minimal Architecture for Generalized Markup Applications*), *SLIM* (*Structured Language for Internet Markup*) et *MGML* (*Minimal Generalized Markup Language*). Les co-rédacteurs de la spécification étaient à l'origine Tim Bray et Michael Sperberg-McQueen. À mi-chemin du projet, Bray a accepté un contrat de consultant avec Netscape, ce qui a provoqué de vives protestations de la part de Microsoft. Il a été demandé à Bray de démissionner temporairement du poste de rédacteur en chef. Cela a conduit à une intense dispute au sein du groupe de travail, finalement résolue par la nomination de Jean Paoli de Microsoft comme troisième co-rédacteur. Le groupe de travail *XML* ne s'est jamais réuni en personne ; la conception s'est faite par le biais d'une combinaison de courriers électroniques et de téléconférences hebdomadaires. Les principales décisions de conception ont été prises au cours d'une courte période de travail intense entre août et novembre 1996, lorsque le premier projet de spécification *XML* a été publié. À la suite du travail de conception de 1997 *XML* 1.0 est devenu une recommandation du *W3C* le 10 février 1998.

L'essentiel de la raison pour laquelle les langages de balisage extensibles sont nécessaires est expliqué à Langage de balisage (par exemple, voir Langage de balisage § *XML*) et à Langage de balisage généralisé standard. Une centaine de format de document utilisant la syntaxe *XML* ont été mis en place, notamment *RSS*, Atom, *SOAP*, *SVG* et *XHTML*. Les formats basés sur *XML* sont devenus le langage par défaut de nombreux outils de productivité bureautique, notamment Microsoft Office (Office Open *XML*), OpenOffice.org et LibreOffice (OpenDocument), et iWork d'Apple. *XML* a également fourni le langage de base pour des protocoles de communication tels que *XMPP*. De nombreuses normes de données industrielles, telles que *Health Level 7*, *OpenTravel Alliance*, *FpML*, *MISMO* et *National Information Exchange Model*, sont basées sur *XML* et les riches fonctionnalités de la spécification du schéma *XML*. Nombre de ces normes sont assez complexes et il n'est pas rare qu'une spécification comprenne plusieurs milliers de pages. Dans le domaine de l'édition, la *Darwin Information Typing Architecture* est une norme de données industrielles *XML*. Le *XML* est largement utilisé pour prendre en charge divers formats de publication. Le *XML* est largement utilisé dans une architecture orientée services (*SOA*). Des systèmes distincts communiquent entre eux en échangeant des messages *XML*. Le format d'échange des messages est normalisé sous la forme d'un schéma *XML* (*XSD*). Ce schéma est également appelé schéma canonique. Le *XML* est devenu d'usage courant pour l'échange de données sur l'Internet. La *RFC 3023* (*Request for comments 3023*, https://datatracker.ietf.org/doc/html/rfc3023) de l'*IETF* (*Internet Engineering Task Force*), aujourd'hui remplacée par la *RFC 7303* (*Request for comments 7303*, https://datatracker.ietf.org/doc/html/rfc7303), donnait des règles pour la construction des types de données Internet à utiliser lors de l'envoi de *XML*. Elle définit aussi les types de données application/xml et text/xml, qui indiquent uniquement que les données sont en *XML*, et rien sur leur sémantique. La *RFC 7303* recommande également d'attribuer aux langages basés sur *XML* des types de médias se terminant par +xml, par exemple image/svg+xml pour *SVG*. D'autres directives pour l'utilisation et l'application de *XML* dans un contexte de réseau apparaissent dans la *RFC 3470* (*Request for comments 3470*, https://datatracker.ietf.org/doc/html/rfc3470), un document qui couvre de nombreux aspects de la conception et de la mise en œuvre d'un langage basé sur *XML*.

Les objectifs de conception de *XML* incluent le fait qu'il soit facile d'écrire des programmes qui traitent les documents *XML*. Malgré cela, la spécification de *XML* ne contient presque aucune information sur la manière dont les programmeurs peuvent procéder à un tel traitement. La spécification *XML Infoset* fournit un vocabulaire permettant de se référer aux constructions d'un document *XML*, mais ne donne aucune indication sur la manière d'accéder à ces informations. Diverses *API* d'accès au *XML* ont été développées et utilisées.

Les *API* existantes pour le traitement *XML* tendent à entrer dans ces catégories :

- *API* orientées flux accessibles depuis un langage de programmation, par exemple *SAX* et *StAX*.

- *APIs* de traversée d'arbre accessibles depuis un langage de programmation, par exemple *DOM*.

- La liaison de données *XML*, qui fournit une traduction automatisée entre un document *XML* et des objets en langage de programmation.

- Les langages de transformation déclaratifs tels que *XSLT* et *XQuery*.

- Extension de la syntaxe des langages de programmation à usage général, par exemple *LINQ* et *Scala*.

Les installations orientées flux nécessitent moins de mémoire et, pour certaines tâches basées sur une traversée linéaire d'un document *XML*, sont plus rapides et plus simples que d'autres alternatives. Les *API* de traversée d'arbre et de liaison de données nécessitent généralement beaucoup plus de mémoire, mais sont souvent jugées plus pratiques à utiliser par les programmeurs ; certaines incluent la récupération déclarative des composants du document via l'utilisation d'expressions *XPath*.

XSLT est conçu pour la description déclarative des transformations de documents *XML*, et a été largement mis en œuvre à la fois dans les paquets côté serveur et les navigateurs Web. *XQuery* recouvre *XSLT* dans sa fonctionnalité, mais est davantage conçu pour la recherche dans les grandes bases de données *XML*.

API simple pour XML : Simple *API* pour *XML* (*SAX*) est une *API* lexicale et événementielle dans laquelle un document est lu en série et son contenu est signalé sous forme de rappels à diverses méthodes d'un objet de traitement conçu par l'utilisateur. *SAX* est rapide et efficace à mettre en œuvre, mais difficile à utiliser pour extraire des informations au hasard du *XML*, car il tend à imposer à l'auteur de l'application de garder la trace de la partie du document en cours de traitement. Il est mieux adapté aux situations dans lesquelles certains types d'informations sont toujours traités de la même manière, quel que soit l'endroit où ils se trouvent dans le document.

Pull parsing : L'analyse syntaxique Pull traite le document comme une série d'éléments lus en séquence en utilisant le modèle de conception Iterator. Cela permet d'écrire des analyseurs descendants récursifs dans lesquels la structure du code effectuant l'analyse reflète la structure du XML analysé, et les résultats intermédiaires analysés peuvent être utilisés et accessibles en tant que variables locales dans les fonctions effectuant l'analyse, ou transmis dans des fonctions de niveau inférieur, ou retournés à des fonctions de niveau supérieur. Parmi les exemples d'analyseurs pull, citons Data::Edit::Xml en Perl, StAX dans le langage de programmation Java, XMLPullParser en Smalltalk, XMLReader en PHP, ElementTree.iterparse en Python, System.Xml.XmlReader dans le .NET Framework et l'API de traversée du DOM (NodeIterator et TreeWalker).

Un analyseur pull crée un itérateur qui visite séquentiellement les différents éléments, attributs et données d'un document *XML*. Le code qui utilise cet itérateur peut tester l'élément actuel (pour savoir, par exemple, s'il s'agit d'une balise de début ou de fin, ou d'un texte), et inspecter ses attributs (nom local, espace de noms, valeurs des attributs *XML*, valeur du texte, etc. Le code peut ainsi extraire des informations du document. L'approche par descente récursive tend à se prêter à la conservation des données sous forme de variables locales typées dans le code effectuant l'analyse syntaxique, alors que *SAX*, par exemple, exige généralement qu'un analyseur syntaxique conserve manuellement les données intermédiaires dans une pile d'éléments qui sont des éléments parents de l'élément analysé. Le code d'analyse syntaxique "pull" peut être plus simple à comprendre et à maintenir que le code d'analyse syntaxique *SAX*.

Modèle d'objet de document : Le *DOM* (*Document Object Model*) est une *API* qui permet de naviguer dans l'ensemble d'un document comme s'il s'agissait d'un arbre composé de nœuds représentant le contenu du document. Un document *DOM* peut être créé par un analyseur syntaxique, ou peut être généré manuellement par les utilisateurs (avec des limitations). Les types de données dans les nœuds *DOM* sont abstraits, les implémentations fournissent leurs propres associations spécifiques au langage de programmation. Les implémentations *DOM* ont tendance à être gourmandes en mémoire, car elles exigent généralement que le document entier soit chargé en mémoire et construit comme un arbre d'objets avant que l'accès ne soit autorisé.

Data binding : La liaison de données *XML* est la liaison de documents *XML* à une hiérarchie d'objets personnalisés et fortement typés, par opposition aux objets génériques créés par un analyseur *DOM*. Cette approche simplifie le développement du code et, dans de nombreux cas, permet d'identifier les problèmes au moment de la compilation plutôt qu'au moment de l'exécution. Elle convient aux applications où la structure du document est connue et fixe au moment de l'écriture de l'application. Parmi les exemples de systèmes de liaison de données, citons l'architecture Java pour la liaison *XML* (*JAXB*), la sérialisation *XML* dans .NET Framework et la sérialisation *XML* dans *gSOAP*.

XML comme type de données : *XML* est apparu comme un type de données de première classe dans d'autres langages. L'extension E4X (*ECMAScript for XML*) du langage ECMAScript/JavaScript définit explicitement deux objets spécifiques (*XML* et *XMLList*) pour JavaScript, qui prennent en charge les nœuds de document *XML* et les listes de nœuds *XML* en tant qu'objets distincts et utilisent une notation par points spécifiant les relations parent-enfant. *E4X* est pris en charge par les navigateurs Mozilla 2.5+ (bien qu'il soit désormais déprécié) et Adobe Actionscript, mais n'a pas été adopté de manière plus universelle. Des notations similaires sont utilisées dans l'implémentation *LINQ* de Microsoft pour Microsoft .NET 3.5 et plus, et dans Scala (qui utilise la VM Java). L'application open-source xmlsh, qui fournit un shell de type Linux avec des fonctionnalités spéciales pour la manipulation du *XML*, traite de manière similaire le *XML* comme un type de données, en utilisant la notation <[ ]>. Le Resource Description Framework définit un type de données rdf:XMLLiteral pour contenir le *XML* enveloppé et canonique. Facebook a produit des extensions aux langages *PHP* et *JavaScript* qui ajoutent le *XML* à la syntaxe de base de manière similaire à *E4X*, à savoir *XHP* et *JSX* respectivement.

2.1.1.3. XML-RPC

Le protocole XML-RPC a été créé en 1998 par Winer de UserLand Software et Microsoft, et Microsoft le considère comme un élément essentiel pour renforcer ses efforts dans le domaine du commerce électronique. Au fur et à mesure de l'introduction de nouvelles fonctionnalités, la norme a évolué vers ce qui est aujourd'hui *SOAP*. UserLand prend en charge *XML-RPC* depuis la version 5.1 du système de gestion de contenu Frontier, sorti en juin 1998. L'idée de *XML-RPC* d'une norme lisible et inscriptible par l'homme et analysable par script pour les demandes et les réponses basées sur *HTTP* a également été mise en oeuvre dans des spécifications concurrentes telles que le *WDDX* (*Web Distributed Data Exchange*) d'Allaire et le *WIDL* (*Web Interface Definition Language*) de webMethod. L'art antérieur d'envelopper les objets *COM*, *CORBA* et Java *RMI* dans la syntaxe *XML* et de les transporter via *HTTP* existait également dans la technologie WebBroker de DataChannel. L'utilisation générique de *XML* pour l'appel de procédure à distance a été brevetée par Phillip Merrick, Stewart Allen et Joseph Lapp en avril 2006, revendiquant le bénéfice d'une demande provisoire déposée en mars 1998. Le brevet a expiré le 23 mars 2019. Dans *XML-RPC*, un client effectue *RPC* en envoyant une requête *HTTP* au serveur implémentant *XML-RPC* et en répondant avec *HTTP*. Le protocole définit plusieurs types de données pour les paramètres et le résultat. Certains de ces types de données sont imbriqués. Par exemple, il est possible d'avoir un paramètre qui est un tableau de tableau de plusieurs entiers. La structure des paramètres/résultats et l'ensemble des types de données sont censés refléter ceux qui sont utilisés dans les langages de programmation courants. L'identification des clients à des fins d'autorisation peut être faite à l'aide de méthodes de sécurité *HTTP*. L'authentification d'accès de base peut servir pour l'identification et l'authentification. Par rapport aux protocoles *RESTful*, où les représentations de ressources sont transférées ; *XML-RPC* est conçu pour appeler des méthodes. La différence pratique réside dans le fait que *XML-RPC* est beaucoup plus structuré, ce qui signifie que du code de bibliothèque commun peut être appliqué pour mettre en œuvre les clients et les serveurs. Cela signifie aussi qu'il y a moins de travail de conception et de documentation pour un protocole d'application spécifique.

Les types de données courants sont convertis en leurs équivalents *XML*, avec des exemples de valeurs présentés ci-dessous :

Nom	**Example**	**Description**

array	`<array>`
	`  <data>`
	`  <value><i4>210</i4></value>`
	`  <value><string>Une valeur ici </string></value>`
	`  <value><i4>1</i4></value>`
	`  </data>`
	`</array>`
		Tableau de valeurs, ne contenant pas de clés

base64	`<base64>eW91IGNhbid0wxcjjhwqtyIHJlYWQgdGhpcyE=</base64>`	Données binaires codées en base64
boolean	`<boolean>1</boolean>`	Valeur logique booléenne (0 ou 1)

date/time	`<dateTime.iso8601>19981021T14:08:55Z</dateTime.iso8601>`	Date et heure au format *ISO 8601*
double	`<double>-3.14</double>`	Nombre à virgule flottante de double précision
integer	`<int>42</int>`	Nombre entier, nombre entier
string	`<string>Hello world!</string>`	String of characters. Must follow *XML* encoding.

struct	`<struct>`
	`  <member>`
	`  <name>foo name</name>`
	`  <value><i4>value</i4></value>`
	`  </member>`
	`  <member>`
	`  <name>bar name</name>`
	`  <value><i4>2</i4></value>`
	`  </member>`
	`</struct>`
		Tableau associatif

nil	`<nil/>`	Valeur nulle discriminée ; une extension *XML-RPC*

2.1.2. Encodage JSON-RPC

JSON-RPC est un protocole d'appel de procédure à distance codé en *JSON*. Il est similaire au protocole *XML-RPC*, ne définissant que quelques types de données et de commandes. *JSON-RPC* permet les notifications (données envoyées au serveur qui ne nécessitent pas de réponse) et l'envoi de plusieurs appels au serveur qui peuvent recevoir une réponse asynchrone.

2.1.2.1. JSON

JSON est un format de fichier standard ouvert et un format d'échange de données qui utilise du texte lisible par l'homme pour stocker et transmettre des objets de données composés de paires attribut-valeur et de tableaux. Il s'agit d'un format de données très courant, avec une gamme variée d'applications, un exemple étant les applications web qui communiquent avec un serveur. *JSON* est un format de données qui est indépendant du langage. Il est dérivé de JavaScript, mais de nombreux langages de programmation modernes incluent du code permettant de générer et d'analyser des données au format *JSON*. Les noms de fichiers *JSON* utilisent l'extension .json. *JSON* est né du besoin d'un protocole de communication serveur-navigateur stateless (« sans état ») et en temps réel, sans utiliser les méthodes dominantes du début des années 2000, c'est-à-dire, des plugins de navigateur tels que Flash ou des applets Java (un applet aussi orthographié applette ou aussi nommée appliquette, est un mini-logiciel qui s'exécute dans la fenêtre d'une autre application, en général un navigateur web). Un précurseur des bibliothèques *JSON* a été utilisé dans un projet de jeu d'échange d'actifs numériques pour enfants nommé « *Cartoon Orbit* » sur communities.com pour Cartoon Network, qui utilisait un plug-in côté navigateur avec un format de messagerie propriétaire pour manipuler les éléments Dynamic *HTML* (ce système est également la propriété de *3DO*). Après avoir découvert les premières capacités d'Ajax, digiGroups, Noosh et d'autres ont utilisé des cadres pour faire passer des informations dans le champ visuel des navigateurs des utilisateurs sans rafraîchir le contexte visuel d'une application Web, réalisant ainsi des applications Web riches en temps réel en utilisant uniquement les capacités *HTTP*, *HTML* et JavaScript standard de Netscape 4.0.5+ et IE 5+. Doglas Crockford a été le premier à spécifier et à populariser le format *JSON*. Les cofondateurs de State Software ont convenu de construire un système qui utilisait les capacités standard des navigateurs et fournissait une couche d'abstraction permettant aux développeurs Web de créer des applications Web à état qui disposaient d'une connexion duplex persistante avec un serveur Web en maintenant ouvertes deux connexions *HTTP* et en les recyclant avant les délais standard des navigateurs si aucune autre donnée n'était échangée. Lors d'un tour de table, les cofondateurs ont décidé d'appeler le format de données *JSML* ou *JSON*, et de décider sous quel type de licence le rendre disponible.

JSON est basé sur un sous-ensemble du langage de script JavaScript (plus précisément, la norme *ECMA-262 3e édition-décembre 1999*) et est couramment utilisé avec JavaScript, mais c'est un format de données indépendant du langage. Le code permettant d'analyser et de générer des données *JSON* est facilement disponible dans de nombreux langages de programmation. Le site Web de *JSON* répertorie les bibliothèques *JSON* par langage. Ecma International a publié la première édition de sa norme *JSON ECMA-404* en octobre 2013. La même année, le *RFC 7158* (*Request for comments 7158*, https://datatracker.ietf.org/doc/html/rfc7158) a utilisé *ECMA-404* comme référence. En 2014, le *RFC 7159* (*Request for comments 7159*, https://datatracker.ietf.org/doc/html/rfc7159) est devenu la référence principale pour les utilisations de *JSON* sur Internet, remplaçant le *RFC 4627* (*Request for comments 4627*, https://datatracker.ietf.org/doc/html/rfc4627) et le *RFC 7158* (*Request for comments 7158*, https://datatracker.ietf.org/doc/html/rfc7158) mais conservant l'*ECMA-262* et l'*ECMA-404* comme références principales. En novembre 2017, l'*ISO/IEC JTC 1/SC 22* a publié l'*ISO/IEC 21778:2017* en tant que norme internationale. Le 13 décembre 2017, l'Internet Engineering Task Force a rendu obsolète le *RFC 7159* (*Request for comments 7159*, https://datatracker.ietf.org/doc/html/rfc7159) en publiant le *RFC 8259* (*Request for comments 8259*, https://datatracker.ietf.org/doc/html/rfc8259).

Douglas Crockford a ajouté une clause à la licence *JSON* stipulant que « le logiciel doit être utilisé pour le bien et non pour le mal », afin d'ouvrir le code source des bibliothèques *JSON* tout en se moquant des avocats d'affaires et de ceux qui sont trop pédants. D'un autre côté, cette clause a entraîné des problèmes de compatibilité de la licence *JSON* avec d'autres licences open-source, car les logiciels open-source et les logiciels libres n'impliquent généralement aucune restriction quant au but de l'utilisation.

L'exemple suivant montre une représentation *JSON* possible décrivant une personne.

```
{
  "firstName": "John",
  "lastName": "Doe",
  "isAlive": true,
  "age": 25,
  "address": {
    "streetAddress": "2 Rue de la faisanderie",
    "city": "PARIS",
    "postalCode": "75116"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "0154859878"
    },
    {
      "type": "office",
      "number": "0121458963"
    }
  ],
  "children": [],
  "spouse": null
}
```

Bien que Crockford ait initialement affirmé et cru que *JSON* était un sous-ensemble strict de *JavaScript* et d'*ECMAScript*, sa spécification autorise en fait des documents *JSON* valides qui ne sont pas des *JavaScript* valides ; *JSON* permet aux terminaisons de ligne Unicode *U+2028 LINE SEPARATOR* et *U+2029 PARAGRAPH SEPARATOR* d'apparaître sans échappement dans les chaînes citées, alors que *ECMAScript* 2018 et les versions antérieures ne le permettent pas. Pour une portabilité maximale, ces caractères doivent être mis en évidence par un antislash. Cette subtilité est importante lors de la génération de *JSONP*.

Les échanges *JSON* dans un écosystème ouvert doivent être encodés en *UTF-8*. Cet encodage prend en charge l'ensemble du jeu de caractères Unicode, y compris les caractères situés en dehors du plan de base multilingue (U+10000 à U+10FFFF). Toutefois, s'ils sont échappés, ces caractères doivent être écrits à l'aide de paires de substituts *UTF-16*, un détail qui échappe à certains analyseurs *JSON*. Par exemple, pour inclure le caractère Emoji U+1F610 😐 NEUTRAL FACE dans *JSON* :

```
{ "face": "😐" }
// ou
```

```
{ "face": "\uD83D\uDE10" }
```

JSON est devenu un sous-ensemble strict d'*ECMAScript* à partir de la révision 2019 du langage.

Les types de données de *JSON* sont :

- **Nombre :** un nombre décimal signé qui peut contenir une partie fractionnaire et peut utiliser la notation exponentielle E, mais ne peut pas inclure des non-nombres tels que NaN. Le format ne fait aucune distinction entre les nombres entiers et les nombres à virgule flottante. JavaScript utilise un format à virgule flottante en double précision pour toutes ses valeurs numériques (jusqu'à plus tard, il prend également en charge BigInt), mais d'autres langages mettant en œuvre *JSON* peuvent coder les nombres différemment.
- **Chaîne :** une séquence de zéro ou plusieurs caractères Unicode. Les chaînes sont délimitées par des guillemets et supportent une syntaxe d'échappement de type backslash.
- **Booléen :** l'une ou l'autre des valeurs true ou false.
- **Tableau :** une liste ordonnée de zéro ou plusieurs éléments, chacun d'entre eux pouvant être de n'importe quel type. Les tableaux utilisent la notation entre crochets avec des éléments séparés par des virgules.
- **Objet :** une collection de paires nom-valeur où les noms (également appelés clés) sont des chaînes de caractères. Les objets sont destinés à représenter des tableaux associatifs, où chaque clé est unique dans un objet. Les objets sont délimités par des accolades et utilisent des virgules pour séparer chaque paire, tandis qu'à l'intérieur de chaque paire, le caractère deux-points « : » sépare la clé ou le nom de sa valeur.
- **Null** : une valeur vide, en utilisant le mot null.

Les espaces blancs sont ignorés autour ou entre les éléments syntaxiques. Quatre caractères spécifiques sont considérés comme des espaces à cette fin : l'espace, la tabulation horizontale, le saut de ligne et le retour chariot. À l'exception de beaucoup de langage, *JSON* ne fournit pas de syntaxe pour les commentaires.

Les premières versions de *JSON* (telles que spécifiées par la *RFC 4627*, https://datatracker.ietf.org/doc/html/rfc4627) exigeaient qu'un texte JSON valide soit constitué uniquement d'un type d'objet ou de tableau, qui pouvait contenir d'autres types. Cette restriction a été abandonnée dans la *RFC 7158* (https://datatracker.ietf.org/doc/html/rfc7158), où un texte *JSON* a été redéfini comme toute valeur sérialisée.

Les nombres en *JSON* sont agnostiques par rapport à leur représentation dans les langages de programmation. Bien que cela permette de sérialiser des nombres d'une précision arbitraire, cela peut entraîner des problèmes de portabilité. Par exemple, comme aucune différenciation n'est faite entre les valeurs entières et les valeurs à virgule flottante, certaines implémentations peuvent traiter 42, 42.0 et 4.2E+1 comme le même nombre, alors que d'autres ne le feront pas. La norme *JSON* n'impose aucune exigence concernant les détails de mise en œuvre tels que le débordement, le sous-débordement, la perte de précision, l'arrondi ou les zéros signés, mais elle recommande de ne pas s'attendre à une précision supérieure à celle de la norme *IEEE 754* binaire64 pour une « bonne interopérabilité ». Il n'y a pas de perte de précision inhérente à la sérialisation d'une représentation binaire au niveau machine d'un nombre à virgule flottante (comme binary64) en une représentation décimale lisible par l'homme (comme les nombres dans *JSON*), et inversement, puisqu'il existe des algorithmes publiés pour le faire de manière exacte et optimale. Les commentaires ont été délibérément exclus de *JSON*. En 2012, Douglas Crockford a décrit sa décision de conception ainsi : « J'ai supprimé les commentaires de *JSON* parce que j'ai vu que les gens les utilisaient pour contenir des directives d'analyse, une pratique qui aurait détruit l'interopérabilité ». *JSON* désapprouve les « virgules traînantes », une virgule après la dernière valeur à l'intérieur d'une structure de données. Les virgules traînantes sont une caractéristique commune des dérivés de *JSON* pour améliorer la facilité d'utilisation.

Le type *MIME* officiel pour le texte *JSON* est « application/json », et la plupart des implémentations modernes l'ont adopté. Le type *MIME* non officiel « text/json » ou le type de contenu « text/javascript » sont également pris en charge, pour des raisons historiques, par de nombreux fournisseurs de services, navigateurs, serveurs, applications web, bibliothèques, cadres et *API*. Parmi les exemples notables, citons l'*API* de recherche Google, Yahoo !, Flickr, l'*API* Facebook, le framework Lift, le Dojo Toolkit 0.4, etc.

JSON Schema spécifie un format basé sur *JSON* pour définir la structure des données *JSON* pour la validation, la documentation et le contrôle des interactions. Il fournit un contrat pour les données *JSON* requises par une application donnée, et fournit aussi une explication du comment ces données peuvent être modifiées. *JSON Schema* est basé sur les concepts de *XSD* (*XML Schema)*, mais est basé sur *JSON*. Les mêmes outils de

sérialisation/désérialisation que XSD peuvent être utilisés à la fois pour le schéma et les données, et il est autodescriptif. Il est spécifié dans un Internet Draft à l'*IETF*, actuellement dans le draft 2019-09, qui a été publié le 19 septembre 2019. Plusieurs validateurs sont disponibles pour différents langages de programmation, chacun avec des niveaux de conformité variables. Il n'existe pas d'extension de nom de fichier standard, mais certains ont suggéré schema json.

La norme *JSON* ne prend pas en charge les références d'objet, mais il existe un projet de norme *IETF* pour les références d'objet basées sur *JSON*. La boîte à outils Dojo prend en charge les références d'objet à l'aide de la norme *JSON* ; plus précisément, le module dojox.json.ref prend en charge plusieurs formes de référencement, notamment le référencement circulaire, multiple, intermessage et paresseux. En interne, les deux modules le font en assignant une clé "$ref" pour de telles références et en la résolvant au moment de l'analyse syntaxique ; le projet de l'IETF ne spécifie que la syntaxe *URL*, mais Dojo en permet davantage Il existe également des solutions non standard telles que l'utilisation des variables Sharp de Mozilla JavaScript. Toutefois, cette fonctionnalité est devenue obsolète avec JavaScript 1.8.5 et a été supprimée dans la version 12 de Firefox.

2.1.2.2. JSON-RPC

JSON-RPC fonctionne en envoyant une requête à un serveur mettant en œuvre ce protocole. Dans ce cas, le client est généralement un logiciel qui a l'intention d'appeler une seule méthode d'un système distant. Plusieurs paramètres d'entrée peuvent être transmis à la méthode distante sous forme de tableau ou d'objet, tandis que la méthode elle-même peut également renvoyer plusieurs données de sortie. Les mots clés « *MUST* », « *MUST NOT* », « *REQUIRED* », « *SHALL* », « *SHALL NOT* », « *SHOULD* », « *SHOULD NOT* », « *RECOMMENDED* », « *MAY* » et « *OPTIONAL* » dans cette partie doivent être interprétés comme décrit dans la *RFC 2119* (*Request for comments 2119*, https://datatracker.ietf.org/doc/html/rfc2119). Puisque *JSON-RPC* utilise *JSON*, il possède le même système de types (voir partie 2.1.2.1). *JSON* peut représenter quatre types primitifs (chaînes de caractères, nombres, booléens et nuls) et deux types structurés (objets et tableaux). Le terme "Primitif" dans cette spécification fait référence à l'un de ces quatre types primitifs *JSON*. Le terme « Structuré » fait référence à l'un ou l'autre des types *JSON* structurés. Chaque fois que ce document fait référence à un type *JSON*, la première lettre est toujours en majuscule : Object, Array, String, Number, Boolean, Null. Les termes True et False prennent également la majuscule. Tous les noms de membres échangés entre le client et le serveur qui sont pris en compte pour une correspondance quelconque doivent être considérés comme sensibles à la casse. Les termes « fonction », « méthode » et « procédure » peuvent être considérés comme interchangeables. Le client est défini comme l'origine des objets de demande et le gestionnaire des objets de réponse. Le serveur est défini comme étant l'origine des objets de réponse et le gestionnaire des objets de demande. Une implémentation de cette spécification pourrait facilement remplir ces deux rôles, même en même temps, pour d'autres clients différents ou pour le même client. Cette spécification n'aborde pas cette couche de complexité.

Les objets de requête et de réponse *JSON-RPC 2.0* peuvent ne pas fonctionner avec les clients ou serveurs *JSON-RPC 1.0* existants. Cependant, il est facile de distinguer les deux versions, car la version 2.0 possède toujours un membre nommé « jsonrpc » avec une valeur String de « 2.0 », ce qui n'est pas le cas de la version 1.0. La plupart des implémentations 2.0 devraient envisager d'essayer de gérer les objets 1.0, même si les aspects peer-to-peer et class hinting de la 1.0 ne sont pas pris en compte.

Un appel rpc est représenté par l'envoi d'un objet Request à un serveur. L'objet Request possède les membres suivants :

Jsonrpc : Une chaîne spécifiant la version du protocole *JSON-RPC* doit (« *MUST* ») être exactement « 2.0 ».

Méthode : Chaîne contenant le nom de la méthode à invoquer. Les noms de méthode qui commencent par le mot rpc suivi d'un point (*U+002E* ou *ASCII 46*) sont réservés aux méthodes et extensions internes à rpc et ne doivent pas (« *MUST NOT* ») être utilisés pour autre chose.

Params : Une valeur structurée qui contient les valeurs des paramètres à utiliser lors de l'invocation de la méthode. Ce membre peut (« *MAY* ») être omis.

Id :Un identifiant établi par le client qui doit (« *MUST* ») contenir une chaîne, un nombre ou une valeur NULL s'il est inclus. S'il n'est pas inclus, l'objet Request est supposé être une notification. La valeur ne doit (« *SHOULD* ») normalement pas être NULL et les nombres ne doivent pas (« *SHOULD NOT* ») contenir de parties fractionnaires. Le serveur doit (« *MUST* ») répondre avec la même valeur dans l'objet Réponse s'il est inclus. Ce

membre est utilisé pour corréler le contexte entre les deux objets. Il est déconseillé d'utiliser NULL comme valeur pour le membre id dans un objet de demande, car cette spécification utilise la valeur NULL pour les réponses dont l'id est inconnu. De plus, comme *JSON-RPC* 1.0 utilise une valeur d'id de NULL pour les notifications, cela pourrait entraîner une confusion dans le traitement. Les fractions peuvent poser problème, car de nombreuses fractions décimales ne peuvent pas être représentées exactement comme des fractions binaires.

Une Notification est un objet Request qui ne possède pas de membre « id ». Un objet de demande qui est une notification signifie que le client ne souhaite pas spécialement recevoir de réponse, et en tant que tel, aucun objet de réponse ne doit être renvoyé au client. Le serveur ne doit pas (« *MUST NOT* ») répondre à une notification, y compris celles qui font partie d'une demande par lot. Les notifications ne sont pas confirmables par définition, puisqu'elles n'ont pas d'objet de réponse à renvoyer. En tant que tel, le client n'aurait pas connaissance d'éventuelles erreurs (comme par exemple « Invalid params », « Internal error »).

S'ils sont présents, les paramètres de l'appel rpc doivent (« *MUST* ») être fournis sous forme de valeur structurée. Soit par position à travers un tableau, soit par nom à travers un objet.

- **By-position** : params doit (« *MUST* ») être un tableau, contenant les valeurs dans l'ordre attendu par le serveur.
- **By-name** : params doit (« *MUST* ») être un Object, avec des noms de membres qui correspondent aux noms de paramètres attendus par le serveur. L'absence de noms attendus peut (« *MAY* ») entraîner la génération d'une erreur. Les noms doivent (« *MUST* ») correspondre exactement, y compris la casse, aux paramètres attendus de la méthode.

Lorsqu'un appel rpc est effectué, le serveur doit (« *MUST* ») répondre par une réponse, sauf dans le cas des notifications. La réponse est exprimée sous la forme d'un objet *JSON* unique, avec les membres suivants :

Jsonrpc :

- Une chaîne spécifiant la version du protocole *JSON-RPC* doit (« *MUST* ») être exactement « 2.0 ».

Result :

- Ce membre est obligatoire (« *REQUIRED* ») en cas de succès.
- Ce membre ne doit pas (« *MUST NOT* ») exister si une erreur s'est produite lors de l'appel de la méthode.
- La valeur de ce membre est déterminée par la méthode invoquée sur le serveur.

Error :

- Ce membre est obligatoire (« *REQUIRED* ») en cas d'erreur.
- Ce membre ne doit pas (« *MUST NOT* ») exister si aucune erreur n'a été déclenchée pendant l'invocation.
- La valeur de ce membre doit (« *MUST* ») être un objet tel que défini dans la section 5.1.

Id :

- Ce membre est obligatoire (« *REQUIRED* »).
- Il doit (« *MUST* ») être identique à la valeur du membre id dans le Request Object.
- Si une erreur s'est produite lors de la détection de l'identifiant dans l'objet de demande (par exemple, erreur d'analyse ou demande invalide), il doit (« *MUST* ») être nul.

Le membre « *result* » ou le membre « *error* » doit (« *MUST* ») être inclus, mais les deux membres ne doivent pas (« *MUST NOT* ») l'être.

Lorsqu'un appel rpc rencontre une erreur, le Response Object doit (« *MUST* ») contenir le membre error avec une valeur qui est un Object avec les membres suivants :

Code :

- Un nombre qui indique le type d'erreur qui s'est produit.
- Il doit (« *MUST* ») s'agir d'un nombre entier.

Message :

- Une chaîne fournissant une brève description de l'erreur.
- Le message doit (« *SHOULD* ») être limité à une seule phrase concise.

Données :

- Une valeur primitive ou structurée qui contient des informations supplémentaires sur l'erreur.
- Elle peut être omise.
- La valeur de ce membre est définie par le serveur (par exemple, informations détaillées sur l'erreur, erreurs imbriquées, etc.)

Les codes d'erreur compris entre -32768 et -32000 sont réservés aux erreurs prédéfinies. Tout code compris dans cette plage, mais non défini explicitement ci-dessous, est réservé pour une utilisation future.

Code	Message	Description
-32700	Parse error	Le serveur a reçu un *JSON* non valide. Une erreur s'est produite sur le serveur lors de l'analyse du texte *JSON*.
-32700	Invalid Request	Le *JSON* envoyé n'est pas un objet de requête valide.
-32601	Method not found	La méthode n'existe pas / n'est pas disponible.
-32602	Invalid params	Paramètre(s) de méthode invalide(s).
-32603	Internal error	Internal *JSON-RPC* error.
-32000 to -32099	Server error	Réservé pour les erreurs de serveur définies par l'implémentation.

Pour envoyer plusieurs objets Request en même temps, le client peut (« *MAY* ») envoyer un Array rempli d'objets Request. Le serveur doit répondre par un tableau contenant les objets de réponse correspondants, une fois que tous les objets de demande du lot ont été traités. Un objet de réponse doit (« *SHOULD* ») exister pour chaque objet de demande, sauf qu'il ne doit pas (« *MUST NOT* ») y avoir d'objets de réponse pour les notifications. Le serveur peut (« *MAY* ») traiter un appel rpc par lot comme un ensemble de tâches simultanées, en les traitant dans n'importe quel ordre et avec n'importe quel degré de parallélisme. Les objets de réponse renvoyés par un appel de lot peuvent (« *MAY* ») être renvoyés dans n'importe quel ordre dans le tableau. Le client doit (« *SHOULD* ») faire correspondre les contextes entre l'ensemble d'objets de demande et l'ensemble d'objets de réponse résultant, sur la base du membre id de chaque objet. Si l'appel rpc par lot lui-même n'est pas reconnu comme un *JSON* valide ou comme un tableau comportant au moins une valeur, la réponse du serveur doit (« *MUST* ») être un objet de réponse unique. Si aucun objet de réponse n'est contenu dans le tableau de réponse tel qu'il doit être envoyé au client, le serveur ne doit pas (« *MUST NOT* ») renvoyer un tableau vide et ne doit rien renvoyer du tout.

Voici quelques exemples d'utilisation de *JSON-RPC*.

Syntaxe :

```
--> data sent to Server

<-- data sent to Client
```

Appel rpc avec des paramètres positionnels :

```
--> {"jsonrpc": "2.0", "method": "subtract", "params": [42, 23], "id": 1}

<-- {"jsonrpc": "2.0", "result": 19, "id": 1}

--> {"jsonrpc": "2.0", "method": "subtract", "params": [23, 42], "id": 2}

<-- {"jsonrpc": "2.0", "result": -19, "id": 2}
```

Appel rpc avec des paramètres nommés :

```
--> {"jsonrpc": "2.0", "method": "subtract", "params": {"subtrahend": 23, "minuend": 42}, "id:

3}

<-- {"jsonrpc": "2.0", "result": 19, "id": 3}
```

```
--> {"jsonrpc": "2.0", "method": "subtract", "params": {"minuend": 42, "subtrahend": 23}, "id"
: 4}

<-- {"jsonrpc": "2.0", "result": 19, "id": 4}
```

Une notification :

```
--> {"jsonrpc": "2.0", "method": "update", "params": [1,2,3,4,5]}

--> {"jsonrpc": "2.0", "method": "foobar"}
```

Appel rpc d'une méthode inexistante :

```
--> {"jsonrpc": "2.0", "method": "foobar", "id": "1"}

<-- {"jsonrpc": "2.0", "error": {"code": -32601, "message": "Method not found"}, "id": "1"}
```

Appel rpc avec JSON invalide :

```
--> {"jsonrpc": "2.0", "method": "foobar, "params": "bar", "baz]

<-- {"jsonrpc": "2.0", "error": {"code": -32700, "message": "Parse error"}, "id": null}
```

Appel rpc avec un objet Request invalide :

```
--> {"jsonrpc": "2.0", "method": 1, "params": "bar"}

<-- {"jsonrpc": "2.0", "error": {"code": -32600, "message": "Invalid Request"}, "id": null}
```

Appel rpc Batch, JSON invalide :

```
--> [
  {"jsonrpc": "2.0", "method": "sum", "params": [1,2,4], "id": "1"},
  {"jsonrpc": "2.0", "method"
]

<-- {"jsonrpc": "2.0", "error": {"code": -32700, "message": "Parse error"}, "id": null}
```

Appel rpc avec un Array vide :

```
--> []

<-- {"jsonrpc": "2.0", "error": {"code": -32600, "message": "Invalid Request"}, "id": null}
```

Appel rpc avec un lot invalide (mais non vide) :

```
--> [1]

<-- [
  {"jsonrpc": "2.0", "error": {"code": -32600, "message": "Invalid Request"}, "id": null}
]
```

Appel rpc avec Batch invalide :

```
--> [1,2,3]

<-- [
  {"jsonrpc": "2.0", "error": {"code": -32600, "message": "Invalid Request"}, "id": null},
```

```
  {"jsonrpc": "2.0", "error": {"code": -32600, "message": "Invalid Request"}, "id": null},
  {"jsonrpc": "2.0", "error": {"code": -32600, "message": "Invalid Request"}, "id": null}
]
```

Appel rpc Batch :

```
--> [
        {"jsonrpc":    "2.0",    "method":    "sum",    "params":    [1,2,4],    "id":    "1"},
        {"jsonrpc":      "2.0",      "method":      "notify_hello",      "params":      [7]},
        {"jsonrpc":   "2.0",   "method":   "subtract",   "params":   [42,23],   "id":   "2"},
        {"foo":                                                                          "boo"},
        {"jsonrpc":  "2.0",  "method":  "foo.get",  "params":  {"name":  "myself"},  "id":  "5"},
        {"jsonrpc":        "2.0",        "method":        "get_data",        "id":        "9"}
    ]
<--                                                                                      [
        {"jsonrpc":          "2.0",          "result":          7,          "id":          "1"},
        {"jsonrpc":          "2.0",          "result":          19,          "id":          "2"},
        {"jsonrpc": "2.0", "error": {"code": -32600, "message": "Invalid Request"}, "id": null
},
        {"jsonrpc": "2.0", "error": {"code": -32601, "message": "Method not found"}, "id": "5"
},
        {"jsonrpc":      "2.0",      "result":      ["hello",      5],      "id":      "9"}
    ]
```

Appel rpc Batch (toutes les notifications) :

```
--> [
        {"jsonrpc":      "2.0",      "method":      "notify_sum",      "params":      [1,2,4]},
        {"jsonrpc":      "2.0",      "method":      "notify_hello",      "params":      [7]}
    ]
<-- //Nothing is returned for all notification batches
```

2.2. Representational state transfer – REST

Dans cette section nous allons expliquer l'architecture *REST* tout en montrant les méthodes d'utilisation qui lui sont associées.

Le transfert d'état représentationnel (*REST*, *Representational state transfer*) est un style d'architecture logicielle qui a été créé pour guider la conception et le développement de l'architecture du World Wide Web. *REST* définit un ensemble de contraintes sur le comportement de l'architecture d'un système hypermédia distribué à l'échelle d'Internet, tel que le Web. *REST* a été utilisé dans l'ensemble de l'industrie logicielle et constitue un ensemble de directives largement acceptées pour la création de services Web fiables et sans état (stateless).

Tout service web qui obéit aux contraintes *REST* est décrit de manière informelle comme étant *RESTful*. Un tel service Web doit fournir ses ressources Web dans une représentation textuelle et permettre leur lecture et leur modification à l'aide d'un protocole sans état et d'un ensemble prédéfini d'opérations. Cette approche permet la plus grande interopérabilité entre les clients et les serveurs dans un environnement durable à l'échelle de l'internet.

Les « ressources web » ont d'abord été définies sur le World Wide Web comme des fichiers ou des documents identifiés par leur URL. Aujourd'hui, la définition est beaucoup plus générique et abstraite, et inclut toute chose, entité ou action qui peut être identifiée, nommée, adressée, manipulée ou exécutée de quelque manière que ce soit sur le web. Dans un service Web *RESTful*, les demandes adressées à l'*URI* d'une ressource génèrent une réponse avec une charge utile majoritairement formatée en *HTML*, *XML* et *JSON*.

Par exemple, la réponse peut confirmer que l'état de la ressource a été modifié. La réponse peut également inclure des liens hypertextes vers des ressources connexes. Le protocole le plus courant pour ces demandes et réponses est le *HTTP*. Il fournit des opérations telles que GET, POST, PUT et DELETE.

En utilisant un protocole sans état et des opérations standard, les systèmes *RESTful* visent des performances rapides, la fiabilité et la capacité de se développer en réutilisant des composants qui peuvent être gérées et mises à jour sans affecter le système dans son ensemble, même pendant son fonctionnement. Les différents objectifs de l'architecture *REST* sont d'accroître les performances, l'évolutivité, la simplicité, la visibilité, la portabilité et la fiabilité. Ces objectifs sont atteints en suivant des principes tels que l'architecture client-serveur, l'absence d'état, la mise en cache, l'utilisation d'un système en couches, le support du code à la demande et l'utilisation d'une interface uniforme. Ces principes doivent être respectés pour que le système soit classé comme *REST*.

Détails des objectifs :

- **La performance** des interactions entre les composants, qui peut être le facteur dominant de la performance perçue par l'utilisateur et de l'efficacité du réseau.
- **L'évolutivité** permettant la prise en charge d'un grand nombre de composants et d'interactions entre les composants.
- **Simplicité** d'une interface uniforme ;
- **Visibilité** de la communication entre les composants par les agents de service
- **Portabilité** des composants par le déplacement du code de programme avec les données ;
- **La fiabilité,** c'est-à-dire la résistance aux défaillances au niveau du système en présence de défaillances au sein des composants, des connecteurs ou des données.

Les contraintes formelles de l'architecture *REST* qui permettent de répondre à ces objectifs sont les suivantes : « *Client–server architecture* », « *Statelessness* », « *Cacheability* », « *Layered system* », « *Code on demand (optional)* », « *Uniform interface* ».

2.2.1. Client–server architecture

Le modèle client-serveur est une structure d'application distribuée qui répartit les tâches ou les charges de travail entre les fournisseurs d'une ressource ou d'un service, appelés serveurs, et les demandeurs de services, appelés clients. Les clients et les serveurs communiquent sur un réseau informatique sur du matériel distinct, mais le client et le serveur peuvent aussi résider dans le même système. Un hôte serveur exécute un ou plusieurs programmes serveurs, qui partagent leurs ressources avec les clients. Un client ne partage généralement aucune de ses ressources, mais il demande un contenu ou un service à un serveur. Ce sont donc les clients qui initient des sessions de communication avec les serveurs, qui attendent les demandes entrantes. Le courrier électronique, l'impression en réseau et le World Wide Web sont des exemples d'applications informatiques qui utilisent le modèle client-serveur.

La caractéristique « client-serveur » décrit la relation entre des programmes coopérants dans une application. Le composant serveur fournit une fonction ou un service à un ou plusieurs clients qui en font la demande. Une ressource partagée peut être n'importe lequel des composants logiciels et électroniques de l'ordinateur serveur, des programmes et des données aux processeurs et aux dispositifs de stockage. Le partage des ressources d'un serveur constitue un service.

En général, un service est une abstraction de ressources informatiques et un client n'a pas à se préoccuper de la façon dont le serveur s'acquitte de la demande et fournit la réponse. Le client doit seulement comprendre la réponse basée sur le protocole d'application bien connu, c'est-à-dire le contenu et le formatage des données pour le service demandé. Les clients et les serveurs échangent des messages selon un modèle de messagerie demande-réponse. Le client envoie une demande et le serveur renvoie une réponse. Cet échange de messages est un exemple de communication interprocessus. Pour communiquer, les ordinateurs doivent avoir un langage commun et suivre des règles afin que le client et le serveur sachent à quoi s'attendre. Le langage et les règles de communication sont définis dans un protocole de communication. Tous les protocoles fonctionnent dans la couche application. Le protocole de la couche application définit les modèles de base du dialogue. Pour formaliser encore davantage l'échange de données, le serveur peut mettre en œuvre une *API*.

L'*API* est une couche d'abstraction permettant d'accéder à un service. En limitant la communication à un format de contenu spécifique, elle facilite l'analyse syntaxique. En abstrayant l'accès, elle facilite l'échange de données entre plates-formes. Un serveur peut recevoir de nombreuses demandes des demandes de clients distincts dans un

court laps de temps. Un ordinateur ne peut effectuer qu'un nombre limité de tâches. Il s'appuie donc sur un système d'ordonnancement pour classer par ordre de priorité les demandes entrantes des clients afin de les satisfaire. Pour éviter les abus et maximiser la disponibilité, le logiciel du serveur peut limiter la disponibilité pour les clients.

2.2.2. Statelessness

Un protocole sans état (stateless) est un protocole de communication dans lequel le récepteur ne doit pas conserver l'état de session des demandes précédentes. L'émetteur transfère l'état de session pertinent au récepteur de manière que chaque demande puisse être comprise de manière isolée, c'est-à-dire sans référence à l'état de session des demandes précédentes conservé par le récepteur. En revanche, un protocole avec état est un protocole de communication dans lequel le récepteur peut conserver l'état de session des demandes précédentes. Dans les réseaux informatiques, les exemples de protocoles sans état comprennent le protocole Internet (*IP*), qui est à la base d'Internet, et le protocole de transfert hypertexte (*HTTP*), qui est à la base du World Wide Web. Le protocole de contrôle de transmission (*TCP*) et le protocole de transfert de fichiers (*FTP*) sont des exemples de protocoles avec état. Les protocoles sans état améliorent les propriétés de visibilité, de fiabilité et d'évolutivité. La visibilité est améliorée parce qu'un système de surveillance n'a pas besoin de regarder au-delà d'une seule demande pour en déterminer la nature complète. La fiabilité est améliorée parce qu'elle facilite la tâche de récupération après des défaillances partielles. L'évolutivité est améliorée, car le fait de ne pas avoir à stocker l'état de la session entre les demandes permet au serveur de libérer rapidement des ressources et simplifie encore la mise en œuvre. L'inconvénient des protocoles sans état est qu'ils peuvent diminuer les performances du réseau en augmentant les données répétitives envoyées dans une série de demandes, puisque ces données ne peuvent pas être laissées sur le serveur et réutilisées.

2.2.3. Cacheability

Un cache Web (ou cache *HTTP*) est un système permettant d'optimiser le World Wide Web. Il est mis en œuvre à la fois côté client et côté serveur. La mise en cache d'images et d'autres fichiers peut permettre de réduire le délai global de navigation sur le Web.

Un cache avant est un cache situé en dehors du réseau du serveur web, par exemple dans le navigateur web du client, dans un fournisseur d'accès à Internet ou dans un réseau d'entreprise. Un serveur proxy situé entre le client et le serveur web peut évaluer les en-têtes *HTTP* et choisir de stocker ou non le contenu web. Un cache inverse se trouve devant un ou plusieurs serveurs web, accélérant les demandes provenant d'Internet et réduisant la charge des serveurs. Il s'agit généralement d'un réseau de diffusion de contenu (*CDN, Content Delivery Network*) qui conserve des copies du contenu web en divers points du réseau.

Le protocole de *HTTP* définit trois mécanismes de base pour contrôler les caches : la date de création, la validation et l'invalidation, spécifiées dans l'en-tête des messages de réponse *HTTP* du serveur. La date de création permet d'utiliser une réponse sans la revérifier sur le serveur d'origine, et peut être contrôlée à la fois par le serveur et le client. Par exemple, l'en-tête de réponse Expires donne une date à laquelle le document devient périmé, et la directive Cache-Control : « *max-age* » indique au cache pendant combien de secondes la réponse est fraîche. La validation peut être utilisée pour vérifier si une réponse mise en cache est toujours valable après qu'elle soit devenue périmée. Par exemple, si la réponse comporte un en-tête « *last-modified* », un cache peut effectuer une requête conditionnelle en utilisant l'en-tête « *if-modified-since* » pour voir si elle a changé. Le mécanisme *ETag* (*entity tag*) permet également une validation forte et faible. L'invalidation est généralement un effet secondaire d'une autre requête qui passe par le cache. Par exemple, si une *URL* associée à une réponse mise en cache reçoit ultérieurement une demande *POST*, *PUT* ou *DELETE*, la réponse mise en cache sera invalidée. De nombreux *CDN* et fabricants d'équipements réseau ont remplacé ce contrôle standard du cache *HTTP* par une mise en cache dynamique.

2.2.4. Layered system

Dans le domaine des télécommunications, un système en couches est un système dans lequel les composants superposés selon une disposition hiérarchique, de sorte que les couches inférieures fournissent des fonctions et des services qui prennent en charge les fonctions et les services des couches supérieures. Des systèmes d'une complexité et d'une capacité toujours plus grandes peuvent être construits en ajoutant ou en modifiant les couches pour améliorer la capacité globale du système tout en utilisant les composants qui sont encore en place.

Un client ne peut généralement pas dire s'il est connecté directement au serveur final ou à un intermédiaire en cours de route. Si un proxy ou un équilibreur de charge est placé entre le client et le serveur, il n'affectera pas leurs communications et il ne sera pas nécessaire de mettre à jour le code du client ou du serveur. Les serveurs

intermédiaires peuvent améliorer l'évolutivité du système en permettant l'équilibrage des charges et en fournissant des caches partagés. De plus, la sécurité peut être ajoutée comme une couche au-dessus des services Web, séparant la logique commerciale de la logique de sécurité. Enfin, les serveurs intermédiaires peuvent appeler plusieurs autres serveurs pour générer une réponse au client.

2.2.5. Code on demand

Le code à la demande est une technologie qui envoie un code logiciel exécutable d'un ordinateur serveur à un ordinateur client à la demande du logiciel du client. Les applets Java, le langage ActionScript d'Adobe pour le lecteur Flash et JavaScript sont des exemples bien connus du paradigme du code à la demande sur le Web. Le code du programme reste inactif sur un serveur Web jusqu'à ce qu'un utilisateur (client) demande une page Web contenant un lien vers le code à l'aide du navigateur Web du client. Lors de cette demande, la page web et le programme sont transportés vers la machine de l'utilisateur à l'aide du protocole *HTTP*. Lorsque la page s'affiche, le code est lancé dans le navigateur et s'exécute localement, à l'intérieur de l'ordinateur de l'utilisateur, jusqu'à ce qu'il soit arrêté (par exemple, lorsque l'utilisateur quitte la page web). Le code à la demande est une utilisation spécifique du code mobile, dans le domaine de la mobilité du code.

Les serveurs peuvent étendre ou personnaliser temporairement la fonctionnalité d'un client en transférant du code exécutable : par exemple, des composants compilés tels que les applets Java, ou des scripts côté client tels que JavaScript.

2.2.6. Uniform interface

La contrainte d'interface uniforme est fondamentale pour la conception de tout système *RESTful*. Elle simplifie et découple l'architecture, ce qui permet à chaque partie d'évoluer indépendamment. Les quatre contraintes de cette interface uniforme sont :

- ***Identification des ressources dans les demandes*** - Les ressources individuelles sont identifiées dans les demandes, par exemple à l'aide des *URI* dans les services *Web RESTful*. Les ressources elles-mêmes peuvent-être conceptuellement distinctes des représentations qui sont renvoyées au client. Par exemple, le serveur peut envoyer les données de sa base de données en *HTML*, *XML* ou *JSON*, qui ne sont pas des représentations internes du serveur.
- ***Manipulation des ressources par le biais des représentations*** - Lorsqu'un client détient une représentation d'une ressource, y compris les métadonnées qui y sont attachées, il dispose de suffisamment d'informations pour modifier ou supprimer l'état de la ressource.
- ***Messages autodescriptifs*** - Chaque message contient suffisamment d'informations pour décrire la manière de le traiter.
- ***L'hypermédia comme moteur de l'état de l'application (HATEOAS, Hypermedia as the Engine of Application State)*** - Après avoir accédé à un *URI* initial pour l'application *REST* un client devrait ensuite être capable d'utiliser les liens fournis par le serveur de manière dynamique pour découvrir toutes les ressources disponibles dont il a besoin. Au fur et à mesure de l'accès, le serveur répond avec un texte qui comprend des hyperliens vers d'autres ressources actuellement disponibles. Il n'est pas nécessaire que le client soit codé en dur avec des informations concernant la structure ou la dynamique de l'application.

2.3. GraphQL

GraphQL est un langage open-source d'interrogation et de manipulation de données pour les *API*, ainsi qu'un runtime permettant de remplir des requêtes avec des données existantes. Il a été développé en interne par Facebook en 2012 sous le nom de *FacebookQL* avant d'être rendu public en 2015.

GraphQL fournit une approche du développement des *API* et a été comparé et contrasté avec l'architecture *REST* et d'autres architectures d'*API*. Elle permet aux clients de définir la structure des données requises, ce qui évite de renvoyer des quantités de données trop importantes.

La flexibilité et la richesse du langage d'interrogation ajoutent également une complexité qui peut ne pas être utile pour les *API* simples. Malgré son nom, *GraphQL* ne fournit pas la richesse des opérations sur les graphes que l'on pourrait trouver dans une base de données de graphes à part entière telle que *Neo4j*, ou même dans les langages de *SQL* qui supportent la fermeture transitive.

Par exemple, une interface *GraphQL* qui indique les parents d'un individu ne peut pas retourner, en une seule requête, l'ensemble de tous ses ancêtres.

GraphQL se compose d'un système de types, d'un langage de requêtes, d'une sémantique d'exécution, d'une validation statique et d'une introspection de types. Il prend en charge la lecture, l'écriture (mutation) et la souscription aux modifications de données (mises à jour en temps réel, le plus souvent implémentées à l'aide de Websockets). Les serveurs *GraphQL* sont disponibles pour de nombreux langages, notamment *Haskell*, *JavaScript*, *Perl*, *Python*, *Ruby*, *Java*, *C++*, *C#*, *Scala*, *Go*, *Rust*, *Elixir*, *Erlang*, *PHP*, *R*, *D* et *Clojure*. Le 9 février 2018, le langage de définition de schéma (*SDL*) *GraphQL* est devenu partie intégrante de la spécification.

Accéder à des graphes d'objets complexes

Les données du monde réel sont difficiles à appréhender, elles contiennent un grand nombre d'objets reliés par des relations complexes :

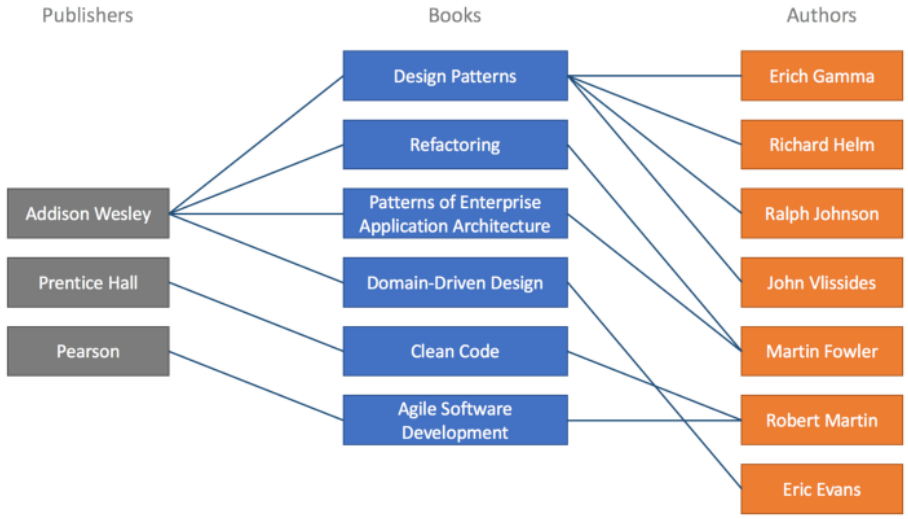

Figure 3 : Graphe d'objets possédant des relations complexes

Voici quelques questions que nous pouvons nous poser :

Quels livres ont été publiés par Addison Wesley ?

Quels livres Martin Fowler a-t-il écrits et qui les a publiés ?

Qui a écrit Clean Code et quels autres livres ont-ils écrits ?

Notez comment nous devons parcourir le graphe d'objets pour répondre à ces questions. Pour la première question, nous devons partir d'un éditeur et passer aux livres. Pour la deuxième question, nous devons partir d'un auteur, puis passer aux livres et enfin aux éditeurs. Pour la troisième question, nous devons commencer par un livre, puis passer aux auteurs et enfin aux livres à nouveau.

Dans le monde *REST* traditionnel, chaque entité est associée à un point de terminaison (ou ressource) distinct. Nous devrions atteindre plusieurs points de terminaison pour répondre à nos questions. Certains cas d'utilisation futurs nécessiteront davantage d'informations et nous devrons modifier nos interfaces pour les prendre en compte. Mais qu'en est-il d'un cas d'utilisation qui a besoin de moins de données ? Soit il obtient plus de données de notre *API* existante, soit nous créons une nouvelle *API* (presque identique) qui renvoie moins de données.

GraphQL change tout cela : il confie le contrôle au client. Le serveur n'est responsable que de la publication de la forme du graphe d'objets et laisse ensuite le client l'interroger comme il le souhaite. Le graphe résultant est renvoyé au client en une seule fois. Par exemple, la troisième question ci-dessus est satisfaite par la requête *GraphQL* suivante :

```
{
 book(id: "clean-code") {
  name
   authors {
    name
     books {
      name
```

```
      }
    }
  }
}
```

Cette requête dit littéralement :

```
Donnez-moi le livre avec id="clean-code"
Donnez-moi le nom du livre
Donnez-moi les auteurs du livre
Donnez-moi le nom de l'auteur
Donnez-moi tous les livres que l'auteur a écrits
Donnez-moi le nom du livre
```

Et le résultat de la requête est explicite :

```
{
 "data": {
  "book": {
   "name": "Clean Code",
    "authors": [
     {
      "name": "Robert C. Martin",
      "books": [
       {
        "name": "Clean Code"
       },
       {
        "name": "Agile Software Development"
       }
      ]
     }
    ]
   }
  }
}
```

Notez que ce qui est renvoyé est un sous-ensemble du graphe d'objets complet, mais il n'est pas renvoyé sous forme de graphe. Le serveur dénormalise le graphe en un arbre (avec éventuellement des nœuds en double) et l'envoie au client en *JSON*.

Précision des réponses

Dans la requête de livre ci-dessus, vous avez peut-être remarqué que nous demandons chaque champ d'entité de manière explicite. Cette syntaxe est obligatoire et permet au client de demander exactement ce dont il a besoin et de n'obtenir que cela. Imaginez que le livre comporte 100 champs, mais que notre vue de liste n'en nécessite que deux : le nom et la date de publication. Nous pourrions alors demander uniquement ces deux champs :

```
{
 books {
  name
   publishDate
 }
}
```

Cette approche améliore considérablement les performances, notamment sur les appareils mobiles, où la bande passante et la puissance de traitement sont limitées.

Maintenir les modèles clients et serveurs.

Un problème majeur lors du développement d'une application est la déconnexion conceptuelle entre le client et le serveur. Ces deux extrémités sont généralement développées par des équipes différentes à des vitesses différentes. Malgré une documentation soignée, il est fréquent que les structures de données du client et du serveur ne soient pas synchronisées. Il en résulte une intégration douloureuse et des bogues qui ne sont pas détectés, même en production. *GraphQL* aide cette situation en proposant une approche axée sur les *API*. La définition de l'*API* permet d'abord d'établir une compréhension commune entre le client et le serveur. Bien que cela puisse également être fait avec *REST*, ce qui est unique à *GraphQL*, c'est son système de type fort, exprimé à l'aide d'une syntaxe

simple appelée *GraphQL Schema Definition Language* (*SDL*). Le client et le serveur peuvent tous deux valider leurs messages pour qu'ils soient conformes au schéma, ce qui évite une catégorie importante de bogues. De plus, nous pouvons générer des parties du client et du serveur à partir du schéma, ce qui rend encore plus pratique la conformité à l'*API*.

Accéder à plusieurs sources de données à partir d'un seul point d'accès

Comme mentionné précédemment, *GraphQL* permet d'accéder à des données provenant d'une ou plusieurs sources. Le client ne se soucie pas de l'emplacement physique des données. Tout ce qu'il sait, c'est le modèle logique des données (le schéma). C'est au serveur d'aller chercher les données dans une ou plusieurs sources et de les assembler pour qu'elles soient conformes au schéma. Le diagramme ci-dessous montre quelques sources de données potentielles telles que des fichiers, des bases de données, des *API REST* et des systèmes de gestion de contenu.

Afficher les changements en temps réel

Il existe toute une série d'applications qui peuvent bénéficier de l'affichage des dernières données en temps réel. Par exemple, une application de trading doit afficher le cours des actions en temps réel. Les jeux et autres applications interactives doivent afficher les événements dès qu'ils se produisent. Les abonnements *GraphQL* sont un moyen de pousser les données du serveur vers les clients en temps réel. Comme défini par la spécification *GraphQL* : « l'abonnement est une requête à longue durée de vie qui récupère des données en réponse à des événements sources ». En fait, les abonnements sont très similaires aux requêtes dans la mesure où ils spécifient un ensemble de champs à renvoyer. Cependant, au lieu de les renvoyer immédiatement, le serveur les envoie au client demandeur chaque fois qu'un événement spécifié se produit. Par exemple, l'abonnement ci-dessous spécifie « bookCreated » comme événement source. Chaque fois que cet événement se produit, le serveur envoie le nom du livre au client.

```
subscription {
 bookCreated {
  book {
   name
  }
 }
}
```

2.4. Etude comparative

Dans cette section, nous resterons objectifs et aborderons les quatre principaux styles d'API dans l'ordre de leur apparition, nous comparerons leurs côtés forts et faibles et nous mettrons en évidence les scénarios dans lesquels chacun d'eux s'inscrit le mieux.

	RPC	**REST**	**GraphQL**
Format	JSON, XML	XML, JSON, HTML, plain text	JSON
Courbe d'apprentissage	Facile	Facile	Moyen
Communauté	Grande	Grande	En pleine croissance
Cas d'utilisation	API de commande, API spécifiques au client pour les microservices internes	API de gestion, Applications simples basées sur les ressources	API mobile, Systèmes complexes et microservices

Avantages de RPC

Interaction simple et directe : *RPC* utilise *GET* pour récupérer des informations et *POST* pour tout le reste. Les mécanismes d'interaction entre un serveur et un client sont aussi simples que d'appeler un point de terminaison et d'obtenir une réponse.

Facilité d'ajout de fonctions : Si nous obtenons une nouvelle exigence pour notre *API*, nous pouvons facilement ajouter un autre point de terminaison exécutant cette exigence : écrire une nouvelle fonction et la lancer derrière

un point de terminaison et un client peut maintenant atteindre ce point de terminaison et obtenir les informations répondant à l'exigence définie.

Haute performance : Les charges utiles légères sont faciles à utiliser sur le réseau et offrent des performances élevées, ce qui est important pour les serveurs partagés et pour les calculs parallèles exécutés sur des réseaux de stations de travail. *RPC* est capable d'optimiser la couche réseau et de la rendre très efficace pour l'envoi de beaucoup de messages par jour entre différents services.

Inconvénients RPC

Couplage étroit avec le système sous-jacent : Le niveau d'abstraction d'une *API* contribue à sa réutilisabilité. Plus elle est étroitement liée au système sous-jacent, moins elle sera réutilisable pour d'autres systèmes. Le couplage étroit du *RPC* au système sous-jacent ne permet pas de créer une couche d'abstraction entre les fonctions du système et l'*API* externe. Cela pose des problèmes de sécurité car il est assez facile de faire fuir des détails de mise en œuvre du système sous-jacent dans l'*API*. Le couplage étroit d'un *RPC* rend les exigences d'évolutivité et les équipes à couplage lâche difficiles à atteindre. Par conséquent, le client s'inquiète des effets secondaires possibles de l'appel d'un point de terminaison particulier ou essaie de déterminer quel point de terminaison appeler parce qu'il ne comprend pas comment le serveur nomme ses fonctions.

Faible capacité de découverte : En *RPC*, il n'y a aucun moyen d'introspecter l'*API* ou d'envoyer une requête et de commencer à comprendre quelle fonction appeler.

Explosion des fonctions : Avec la facilité de création de fonction il est simple d'aboutir à une liste importante de fonctions qui se chevauchent et sont difficiles à comprendre.

Avantages REST

Client et serveur découplés : L'architecture *REST* permet un fort découplage entre le client et le serveur, cela permet une meilleure abstraction que *RPC*. Un système avec des niveaux d'abstraction est capable d'encapsuler ses détails pour mieux identifier et maintenir ses propriétés. Cela rend une *API* avec une architecture *REST* suffisamment flexible pour évoluer dans le temps tout en restant un système stable.

Possibilité de découverte : La communication entre le client et le serveur décrit tout, de sorte qu'aucune documentation externe n'est nécessaire pour comprendre comment interagir avec l'*API* basée sur *REST*.

Prise en charge de la mise en cache : Réutilisant de nombreux outils *HTTP*, *REST* est le seul style qui permet la mise en cache des données au niveau *HTTP*. Toutefois, la mise en œuvre de la mise en cache sur toute autre *API* nécessitera la configuration d'un module de mise en cache supplémentaire.

Prise en charge de plusieurs formats : La possibilité de prendre en charge plusieurs formats pour le stockage et l'échange de données est l'une des raisons pour lesquelles *REST* est actuellement un choix prédominant pour la création d'*API*.

Inconvénients REST

Il n'existe pas de cadre REST unique : Il n'existe pas de méthode exacte pour créer une *API* dite *RESTful*. La manière de modéliser les ressources dépendent de chaque scénario. Cela rend l'architecture *REST* simple en théorie, mais difficile en pratique.

Des charges utiles importantes : *REST* renvoie beaucoup de métadonnées riches afin que le client puisse tout comprendre de l'état de l'application juste à partir de ses réponses. Et ces métadonnées ne sont pas un problème pour un réseau avec une grande capacité de bande passante. Mais ce n'est pas toujours le cas. C'était un des facteurs clés pour Facebook dans la description du style *GraphQL* en 2012.

Problèmes de récupération excessive ou insuffisante : Contenant trop ou trop peu de données, les réponses *REST* créent souvent le besoin d'une autre requête.

Avantages GraphQL

Schéma typé : En faisant pointer un client vers l'*API GraphQL*, on peut découvrir quelles requêtes sont disponibles.

Idéal pour les données graphiques : *GraphQL* est idéal pour les données qui vont profondément dans les relations connexes, grâce à la facilité syntaxique et un « payload » qui se contente que du strict nécessaire.

Pas de contrôle de version : La meilleure pratique en matière de versioning est de ne pas versionner l'*API* du tout. Alors que *REST* propose plusieurs versions de l'*API*, *GraphQL* utilise une version unique et évolutive qui donne un accès continu aux nouvelles fonctionnalités et contribue à un code serveur plus propre et plus facile à maintenir.

Messages d'erreur détaillés : Comme *SOAP*, *GraphQL* fournit des détails sur les erreurs qui se sont produites. Son message d'erreur inclut tous les résolveurs et fait référence à la partie exacte de la requête concernée.

Permissions flexibles : *GraphQL* permet l'exposition sélective de certaines fonctions tout en préservant les informations privées.

Inconvénients GraphQL

Complexité de la mise en cache : Comme *GraphQL* ne réutilise pas la sémantique de la mise en cache *HTTP*, un effort de mise en cache personnalisé est nécessaire.

Beaucoup d'éducation avant le développement : N'ayant pas assez de temps pour comprendre les opérations de niche de *GraphQL* et *SDL*, de nombreux projets décident de suivre la voie bien connue de l'architecture *REST*.

3. Modèles de maturité

La maturité est une mesure de la capacité d'une organisation à s'améliorer continuellement dans une discipline particulière. Plus la maturité est élevée, plus les chances que les incidents ou les erreurs conduisent à des améliorations soit de la qualité, soit de l'utilisation des ressources de la discipline telle qu'elle est mise en œuvre par l'organisation seront grandes. La plupart des modèles de maturité évaluent qualitativement les personnes/culture, les processus/structures et les objets/technologies. Il existe deux approches pour mettre en œuvre les modèles de maturité. Avec une approche descendante, comme celle proposée par Becker, un nombre fixe de stades ou de niveaux de maturité est d'abord spécifié, puis corroboré par des caractéristiques (généralement sous forme d'éléments d'évaluation spécifiques) qui soutiennent les hypothèses initiales sur la façon dont la maturité évolue. Lorsqu'on utilise une approche ascendante, comme celle suggérée par Lahrmann, des caractéristiques ou des éléments d'évaluation distincts sont déterminés dans un premier temps et regroupés dans un deuxième temps en niveaux de maturité afin d'induire une vision plus générale des différentes étapes de l'évolution de la maturité.

3.1. Le modèle de maturité d'Amundsen

Un modèle de maturité de conception bien connu est le modèle d'Amundsen (Amundsen, 2017). Ce modèle définit des niveaux de conformité basés sur le degré d'abstraction de l'API fournie par rapport à l'implémentation sous-jacente. Plus le niveau est élevé, plus l'*API* est découplée des modèles internes des implémentations et plus elle est axée sur les consommateurs de l'*API*. Les niveaux de conformité du modèle d'Amundsen sont les suivants :

- **Niveau 0** : centré sur les données - le modèle de mise en œuvre interne est directement exposé au niveau de l'*API*.
- **Niveau 1** : centré sur l'objet - l'*API* n'expose pas directement le modèle interne, mais elle fournit les moyens (méthodes) de manipuler les objets du modèle interne.
- **Niveau 2** : centré sur les ressources - l'*API* est décrite comme un ensemble de ressources qui peuvent être consommées par les applications clientes ; à ce niveau, les ressources sont indépendantes des objets du modèle interne.
- **Niveau 3** : centré sur l'affordance - l'*API* est décrite comme un ensemble de ressources utilisant des représentations hypermédia pour fournir des actions disponibles (opérations et liens) qui peuvent être exécutées sur la ressource décrite.

Les niveaux 0 et 1 sont considérés comme des modèles internes, car ils exposent les structures internes de l'implémentation au niveau de l'*API*. Les niveaux 2 et 3 sont considérés comme des modèles externes, car ils séparent le modèle externe exposé par l'*API* du modèle interne utilisé par l'implémentation.

3.1.1. AMM0: Amundsen niveau 0

L'abstraction des données est l'une des fonctionnalités principales des *API*. Cependant, au niveau zéro d'Amundsen, il n'y a aucune abstraction entre la couche de persistance du service (par exemple, la base de données) et le client. Cela signifie que, quel que soit le modèle de données contenu dans la base de données, c'est ce modèle de données que le client obtient. Vous ne pouvez pas changer votre base de données sans casser le client.

3.1.2. AMM1: Amundsen niveau 1

Le niveau 1 d'Amundsen fait passer l'exposition des éléments internes de la mise en œuvre de l'*API* de la couche de persistance à la couche d'exécution. Au lieu de traiter les données telles qu'elles sont stockées dans une base de données (niveau 0), les clients sont maintenant présentés avec les objets de mise en œuvre. Il est donc possible, en théorie, de modifier votre base de données, mais pas la mise en œuvre du service (« framework » / « middleware »). Les anciens services Web sont un excellent exemple du niveau un, mais il existe des variantes modernes. Par exemple, le middleware « Restify-Mongoose » expose au client la fonctionnalité d'interrogation du framework de modélisation des objets de la base de données. Pour identifier les *API* de niveau un d'Amundsen, la personne chargée de l'évaluation doit être capable de distinguer si les données, telles qu'acceptées et exposées par l'*API*, sont liées à la mise en œuvre de l'*API* ou non. Si c'est le cas, c'est un mauvais signe : l'*API* laisse échapper des données internes à la mise en œuvre et empêche ainsi toute évolution indépendante de la mise en œuvre de l'*API* par la suite.

3.1.3. AMM2: Amundsen niveau 2

En quittant les niveaux qui exposent les modèles de données de mise en œuvre de l'*API*, nous arrivons à des niveaux supérieurs d'abstraction et de découplage des modèles internes. Au niveau deux, l'*API* parvient à découpler les modèles de données internes de mise en œuvre et de persistance du modèle de représentation des ressources. Cela peut être réalisé à l'aide du « Representor Pattern », où un modèle de données interne est remplacé par une représentation de ressource avant d'être partagé via une couche de transport.

Les *API* de niveau 2 se concentrent sur leur interface et fournissent les données dont les consommateurs ont besoin (et plus encore, voir *AMM3*). En outre, les *API AMM2* n'exposent aucun modèle interne et ont généralement tendance à utiliser correctement les mécanismes *HTTP Content-Negotiation*.

3.1.4. AMM3: Amundsen niveau 3

Enfin, au niveau 3, les *API*s sont axées sur les actions que l'*API* permet et que les consommateurs peuvent entreprendre. Si vous considérez une *API* comme un produit, les possibilités offertes par l'*API* sont modelées en fonction des histoires des utilisateurs du produit *API*. Les actions que l'*API* permet de réaliser reflètent les besoins des consommateurs. Avec les *API* orientées affordances, on ne demande plus aux clients de jeter un coup d'œil aux données et de déterminer ce qu'ils peuvent en faire. Au lieu de cela, on leur présente directement l'action qu'ils peuvent entreprendre. Les *API* de niveau trois sont axées sur les actions et non sur les données. Souvent, les *API* de niveau trois fournissent les actions au moment de l'exécution (*API REST-Hypermédia*) et utilisent les types de médias nécessaires à leur communication (tels que « application/vnd.restful+json »). En tant que telles, les *API* centrées sur les affordances excellent dans la facilité d'utilisation pour les consommateurs, le découplage des éléments internes et l'évolution dans le temps.

3.1.5. Niveaux du modèle de maturité d'Amundsen en exemples

Niveau 0 – « Database Model »

Requête

```
GET /movieinfo?movie=...
```

Réponse

```
HTTP/1.1 200 OK

[

    {

        "_id": "59a0867835f56c7a9476ce92",

        "createdAt": "2017-08-25T20:20:08.864Z",
```

```
        "updatedAt": "2017-08-25T20:20:08.864Z",
        "__v": 0,
        "some": "..",
        "fields": "..",
        "from db": "..",
        "director information": "..."
    }
]
```

Niveau 1 – « Implementation Objects »

Requête

```
GET /movieinfo?q={ movie: { $in: [ ".." ] } }
```

Réponse

```
HTTP/1.1 200 OK
{
   ...
}
```

Niveau 2 – « Resources »

Requête

```
GET /movieinfo?movie=...
```

Réponse

```
HTTP/1.1 200 OK
{
    "some": "..",
    "fields": "..",
    "about": "..",
    "the": "..",
    "movie": "..",
    "director information": "..."
}
```

Niveau 3 – « Affordances »

Requête

```
GET /moviedirector?movie=...
```

Réponse

```
HTTP/1.1 200 OK
{
    "director information": "..."
}
```

Avec le modèle de maturité d'Amundsen, s'affranchir des modèles de données internes permet à la mise en œuvre de votre système d'évoluer au fil du temps. En atteignant au moins le niveau *AMM2*, vous bénéficiez des *APIs* pour éliminer la complexité et créer des systèmes durables.

« Your data model is not your objectmodel is not your resource model is not your affordance model. » Mike Amundsen, 2016.

3.2. Le modèle de maturité de Richardson

Le Richardson Maturity Model (*RMM*) est un modèle de maturité proposé en 2008 par Leonard Richardson qui classe les API Web en fonction de leur adhésion et de leur conformité à chacun des quatre niveaux

du modèle. L'objectif de la recherche du modèle, tel qu'énoncé par l'auteur, était de découvrir la relation entre les contraintes de REST et d'autres formes de services Web. Il divise les parties de la conception RESTful en trois étapes : l'identification des ressources (*URI*), les verbes *HTTP* et les contrôles hypermédia (par exemple, les hyperliens). Le *RMM* a été cité comme étant utile pour évaluer la qualité de la conception d'une *API* Web particulièrement *RESTful* (même s'il n'est pas limité à *REST* uniquement) et critiqué pour ne pas avoir abordé la manière dont un système pourrait atteindre les niveaux de maturité les plus élevés du modèle ainsi que pour avoir considéré un nombre limité d'attributs de qualité.

Le *RMM* peut être utilisé pour déterminer dans quelle mesure une *API* adhère aux principes *REST*. Il classe une *API* en quatre niveaux (de 0 à 3), chaque niveau supérieur correspondant à une adhésion plus complète à la conception *REST*. Le niveau suivant contient également toutes les caractéristiques du niveau précédent.

3.2.1. RMM0: Richardson niveau 0

Le point de départ du modèle consiste à utiliser http comme système de transport pour les interactions à distance, mais sans utiliser aucun des mécanismes du web. En fait, il s'agit d'utiliser http comme un tunnel pour votre propre mécanisme d'interaction à distance, généralement basé sur l'invocation de procédures à distance. Supposons que je veuille prendre un rendez-vous avec mon médecin. Mon logiciel de prise de rendez-vous doit d'abord savoir quels sont les créneaux horaires disponibles chez mon médecin ait une date donnée. Il fait donc une demande au système de prise de rendez-vous de l'hôpital pour obtenir cette information. Dans un scénario de niveau 0, l'hôpital expose un point de terminaison de service à un certain *URI*. J'envoie ensuite à ce point de terminaison un document contenant les détails de ma demande.

```
POST /appointmentService HTTP/1.1
[various other headers]

<openSlotRequest date = "2010-01-04" doctor = "mjones"/>
```

Si tout se passe correctement, nous recevons une réponse disant que mon rendez-vous est pris.

```
HTTP/1.1 200 OK
[various headers]

<appointment>
  <slot doctor = "mjones" start = "1400" end = "1450"/>
  <patient id = "jsmith"/>
</appointment>
```

S'il y a un problème, par exemple si quelqu'un d'autre est entré avant moi, je recevrai une sorte de message d'erreur dans le corps de la réponse.

```
HTTP/1.1 200 OK
[various headers]

<appointmentRequestFailure>
  <slot doctor = "mjones" start = "1400" end = "1450"/>
  <patient id = "jsmith"/>
  <reason>Slot not available</reason>
</appointmentRequestFailure>
```

Jusqu'à présent, il s'agit d'un système simple de type *RPC*. C'est simple, car il s'agit simplement d'envoyer et de recevoir des messages *XML* (*POX*). Si vous utilisez *SOAP* ou *XML-RPC*, il s'agit essentiellement du même mécanisme, la seule différence étant que vous enveloppez les messages *XML* dans une d'enveloppe.

3.2.2. RMM1: Richardson niveau 1

Le premier niveau dans le *RMM* est d'introduire les ressources. Ainsi, au lieu d'adresser toutes nos demandes à un point de terminaison de service unique, nous commençons maintenant à parler à des ressources individuelles.

Ainsi, avec notre requête initiale, nous pourrions avoir une ressource pour un médecin donné.

```
POST /doctors/mjones HTTP/1.1
[various other headers]
```

```
<openSlotRequest date = "2010-01-04"/>
```

La réponse contient les mêmes informations de base, mais chaque créneau est désormais une ressource qui peut être traitée individuellement.

```
HTTP/1.1 200 OK
[various headers]


<openSlotList>
  <slot id = "1234" doctor = "mjones" start = "1400" end = "1450"/>
  <slot id = "5678" doctor = "mjones" start = "1600" end = "1650"/>
</openSlotList>
```

Avec des ressources spécifiques, prendre un rendez-vous signifie s'inscrire à un créneau particulier.

```
POST /slots/1234 HTTP/1.1
[various other headers]

<appointmentRequest>
  <patient id = "jsmith"/>
</appointmentRequest>
```

Si tout se passe bien, je recevrai une réponse similaire à la précédente.

```
HTTP/1.1 200 OK
[various headers]

<appointment>
  <slot id = "1234" doctor = "mjones" start = "1400" end = "1450"/>
  <patient id = "jsmith"/>
</appointment>
```

La différence maintenant est que si quelqu'un a besoin de faire quelque chose à propos du rendez-vous, comme réserver des tests, il doit d'abord s'emparer de la ressource du rendez-vous, qui peut avoir un *URI* comme http://royalhope.nhs.uk/slots/1234/appointment, et poster vers cette ressource.

Plutôt que d'appeler une fonction dans l'éther et de passer des arguments, nous appelons une méthode sur un objet particulier en fournissant des arguments pour les autres informations.

3.2.3. RMM2: Richardson niveau 2

Nous avons utilisé les verbes *HTTP POST* pour toutes les interactions ici aux niveaux 0 et 1, mais certaines personnes utilisent les *GET* à la place ou en plus. À ces niveaux, il n'y a pas de grande différence, ils sont tous deux utilisés comme des mécanismes de tunneling vous permettant de tunneliser vos interactions via *HTTP*. Le niveau 2 s'éloigne de cela, en utilisant les verbes *HTTP* aussi près que possible de la façon dont ils sont utilisés dans le protocole *HTTP* lui-même.

Pour notre liste de créneaux, cela signifie que nous voulons utiliser *GET*.

```
GET /doctors/mjones/slots?date=20100104&status=open HTTP/1.1
Host: royalhope.nhs.uk
```

La réponse est la même que celle qui aurait été donnée avec le *POST*.

```
HTTP/1.1 200 OK
[various headers]

<openSlotList>
  <slot id = "1234" doctor = "mjones" start = "1400" end = "1450"/>
  <slot id = "5678" doctor = "mjones" start = "1600" end = "1650"/>
</openSlotList>
```

Au niveau 2, l'utilisation de *GET* pour une requête comme celle-ci est cruciale. *HTTP* définit *GET* comme une opération sûre, c'est-à-dire qu'elle ne modifie pas de manière significative l'état de quoi que ce soit. Cela nous permet d'invoquer les *GET* en toute sécurité un nombre illimité de fois, dans n'importe quel ordre, et d'obtenir les mêmes résultats à chaque fois. Une conséquence importante est que cela permet à tout participant à l'acheminement des requêtes d'utiliser la mise en cache, qui est un élément clé pour que le Web fonctionne aussi bien qu'il le fait. Le protocole *HTTP* comprend diverses mesures de prise en charge de la mise en cache, qui peuvent être utilisées par tous les participants à la communication. En respectant les règles du protocole *HTTP*, nous sommes en mesure de tirer parti de cette capacité.

Pour prendre un rendez-vous, nous avons besoin d'un verbe *HTTP* qui change d'état, un *POST* ou un *PUT*. Je vais utiliser le même *POST* que précédemment.

```
POST /slots/1234 HTTP/1.1
[various other headers]

<appointmentRequest>
  <patient id = "jsmith"/>
</appointmentRequest>
```

Les compromis entre l'utilisation de *POST* et de *PUT* sont plus importants que ce que je veux dire ici, peut-être que je ferai un article séparé à ce sujet un jour. Mais je tiens à souligner que certaines personnes établissent à tort une correspondance entre *POST/PUT* et create/update. Le choix entre les deux est plutôt différent de cela. Même si j'utilise le même post que le niveau 1, il y a une autre différence significative dans la façon dont le service distant répond. Si tout se passe bien, le service répond avec un code de réponse 201 pour indiquer qu'il y a une nouvelle ressource dans le monde.

```
HTTP/1.1 201 Created
Location: slots/1234/appointment
[various headers]

<appointment>
  <slot id = "1234" doctor = "mjones" start = "1400" end = "1450"/>
  <patient id = "jsmith"/>
</appointment>
```

La réponse 201 comprend un attribut de localisation avec une *URI* que le client peut utiliser pour obtenir l'état actuel de cette ressource à l'avenir. Ici, la réponse comprend également une représentation de cette ressource, ce qui évite au client un appel supplémentaire. Il existe une autre différence si quelque chose se passe mal, par exemple si quelqu'un d'autre réserve la session.

```
HTTP/1.1 409 Conflict
[various headers]

<openSlotList>
  <slot id = "5678" doctor = "mjones" start = "1600" end = "1650"/>
</openSlotList>
```

La partie importante de cette réponse est l'utilisation d'un code de réponse *HTTP* pour indiquer que quelque chose a mal tourné. Dans ce cas, un 409 semble être un bon choix pour indiquer que quelqu'un d'autre a déjà mis à jour la ressource d'une manière incompatible. Plutôt que d'utiliser un code de retour de 200 mais d'inclure une réponse d'erreur, au niveau 2, nous utilisons explicitement une sorte de réponse d'erreur comme celle-ci. C'est au concepteur du protocole de décider des codes à utiliser, mais il doit y avoir une réponse non-2xx en cas d'erreur. Le niveau 2 introduit l'utilisation des verbes *HTTP* et des codes de réponse *HTTP*. Il y a une incohérence qui se glisse ici. Les défenseurs de *REST* parlent d'utiliser tous les verbes *HTTP*. Ils justifient également leur approche en disant que *REST* tente de tirer les leçons du succès pratique du web. Mais le web mondial n'utilise pas beaucoup *PUT* ou *DELETE* dans la pratique. Il existe des raisons valables d'utiliser davantage *PUT* et *DELETE*, mais la preuve de l'existence du web n'en fait pas partie. Les éléments clés qui sont soutenus par l'existence du web sont la forte séparation entre les opérations sûres (par exemple *GET*) et non sûres, ainsi que l'utilisation de codes d'état pour aider à communiquer les types d'erreurs que vous rencontrez.

3.2.4. RMM3: Richardson niveau 3

Le dernier niveau introduit quelque chose que vous entendez souvent désigner sous l'acronyme de *HATEOAS* (*Hypertext As The Engine Of Application State*). Il aborde la question de savoir comment passer d'une liste de créneaux horaires ouverts à la connaissance de ce qu'il faut faire pour prendre un rendez-vous.

Nous commençons avec le même *GET* initial que nous avons envoyé au niveau 2.

```
GET /doctors/mjones/slots?date=20100104&status=open HTTP/1.1
Host: royalhope.nhs.uk
```

Mais la réponse comporte un nouvel élément :

```
HTTP/1.1 200 OK
[various headers]

<openSlotList>
  <slot id = "1234" doctor = "mjones" start = "1400" end = "1450">
    <link rel = "/linkrels/slot/book"
```

```
          uri = "/slots/1234"/>
  </slot>
  <slot id = "5678" doctor = "mjones" start = "1600" end = "1650">
     <link rel = "/linkrels/slot/book"
          uri = "/slots/5678"/>
  </slot>
</openSlotList>
```

Chaque créneau dispose désormais d'un élément de lien qui contient un *URI* nous indiquant comment prendre un rendez-vous. L'intérêt des contrôles hypermédia est qu'ils nous indiquent ce que nous pouvons faire ensuite, ainsi que l'*URI* de la ressource que nous devons manipuler pour le faire. Plutôt que de nous demander où poster notre demande de rendez-vous, les contrôles hypermédia de la réponse nous indiquent comment procéder.

Le *POST* copierait à nouveau celui du niveau 2.

```
POST /slots/1234 HTTP/1.1
[various other headers]

<appointmentRequest>
  <patient id = "jsmith"/>
</appointmentRequest>
```

Et la réponse contient un certain nombre de commandes hypermédia pour différentes choses à faire ensuite.

```
HTTP/1.1 201 Created
Location: http://royalhope.nhs.uk/slots/1234/appointment
[various headers]

<appointment>
  <slot id = "1234" doctor = "mjones" start = "1400" end = "1450"/>
  <patient id = "jsmith"/>
  <link rel = "/linkrels/appointment/cancel"
        uri = "/slots/1234/appointment"/>
  <link rel = "/linkrels/appointment/addTest"
        uri = "/slots/1234/appointment/tests"/>
  <link rel = "self"
        uri = "/slots/1234/appointment"/>
  <link rel = "/linkrels/appointment/changeTime"
        uri = "/doctors/mjones/slots?date=20100104&status=open"/>
  <link rel = "/linkrels/appointment/updateContactInfo"
        uri = "/patients/jsmith/contactInfo"/>
  <link rel = "/linkrels/help"
        uri = "/help/appointment"/>
</appointment>
```

L'un des avantages évidents des contrôles hypermédia est qu'ils permettent au serveur de modifier son schéma *URI* sans perturber les clients. Tant que les clients recherchent l'*URI* du lien « addTest », l'équipe du serveur peut jongler avec tous les *URI* autres que les points d'entrée initiaux. Un autre avantage est qu'il aide les développeurs clients à explorer le protocole. Les liens donnent aux développeurs clients un indice de ce qui pourrait être possible ensuite. Il ne donne pas toutes les informations : les commandes « self » et « cancel » pointent toutes deux vers le même *URI* - ils doivent comprendre que l'une est un *GET* et l'autre un *DELETE*. Mais au moins, cela leur donne un point de départ pour savoir à quoi penser pour obtenir plus d'informations et pour rechercher un *URI* similaire dans la documentation du protocole. De même, cela permet à l'équipe du serveur d'annoncer de nouvelles capacités en ajoutant de nouveaux liens dans les réponses. Si les développeurs clients sont à l'affût de liens inconnus, ces liens peuvent déclencher une exploration plus approfondie. Il n'existe pas de norme absolue sur la manière de représenter les contrôles hypermédia. Ici nous utilisons les recommandations actuelles de suivre le *RFC 4287* (*RFC 4287*, https://datatracker.ietf.org/doc/html/rfc4287). Nous utilisons un élément <link> avec un attribut *URI* pour l'*URI* cible et un attribut « rel » pour décrire le type de relation. Toute relation spécifique à ce serveur est un *URI* pleinement qualifié.

4. Adoption des APIs

Dans cette section nous allons voir l'adoption des *APIs* de manière générale au cours du temps.

Les preuves de l'adoption des *API* Web proviennent de ProgrammableWeb.com, qui est la principale ressource communautaire pour les amateurs et les professionnels du secteur. Cette ressource rassemble les points de terminaison des *API* publiques dans un répertoire complet contenant des informations fournies par les développeurs eux-mêmes. La figure 1 montre le nombre d'enregistrements d'*API* web qui ont été enregistrés depuis 2005. Au premier trimestre 2019, le répertoire ProgrammableWeb comptait 21202 enregistrements.

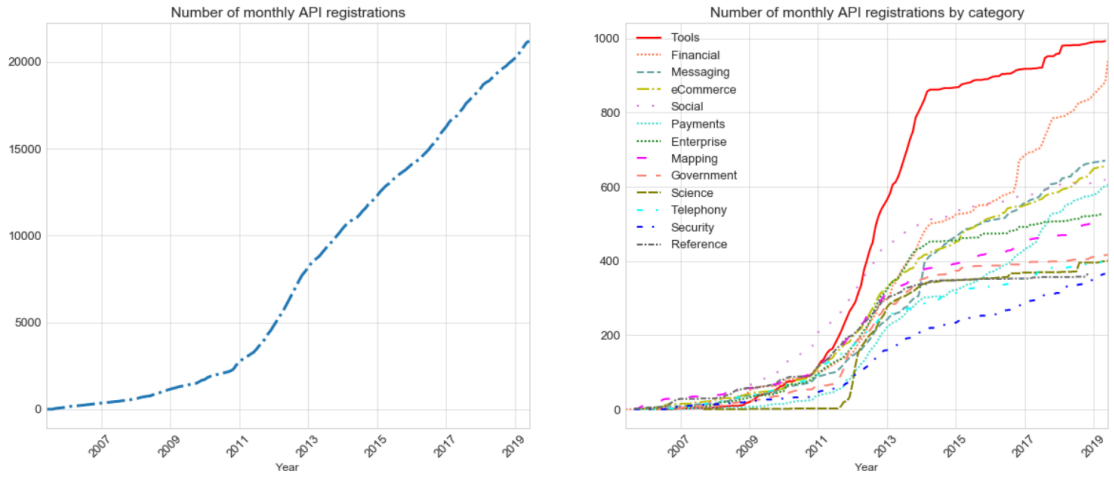

Figure 4 : Courbe d'adoption des APIs

Une indication de l'objectif de l'*API* est contenue dans l'information sur sa catégorie, telle que répertoriée dans le répertoire. Le tableau 1 énumère les catégories les plus fréquentes associées aux enregistrements d'*API* enregistrés. Parmi les catégories les plus fréquentes figurent les catégories financières et de commerce électronique, ainsi que les catégories paiements et entreprises. Le panneau de droite de la figure 1 montre également que les tendances du nombre d'enregistrements d'*API* enregistrés dans les catégories paiements et finance ont augmenté après la publication de la directive révisée sur les services de paiement (*DSP2*, Directive sur les services de paiement) ce qui pourrait avoir influencé le développement de cette tendance. Enfin, nous soulignons que la catégorie gouvernementale se classe parmi les catégories d'enregistrements les plus fréquentes.

Rank	First category	Number	Rank	First category	Number
1	Tools	993	**11**	Telephony	398
2	Financial	944	**12**	Security	366
3	Messaging	671	**13**	Reference	366
4	e-commerce	657	**14**	Search	346
5	Social	619	**15**	Email	346
6	Payments	605	**16**	Video	340
7	Enterprise	528	**17**	Travel	321
8	Mapping	510	**18**	Education	311
9	Government	417	**19**	Sports	303
10	Science	401	**20**	Transportation	292

Figure 5 : Nombres d'APIs par secteur

5. APIs et CA-GIP

CA-GIP ou Crédit Agricole Group Infrastructure Platform est l'entité de production informatique du groupe. Elle a été créée en 2019 à la suite de la fusion des différentes entités de production informatique du groupe. Depuis janvier 2020, *CA-GIP* est organisé en clusters et en socles qui permettent à chaque entité de travailler en étroite relation avec sa production informatique. Le schéma donne un aperçu général de l'organisation *CA-GIP*.

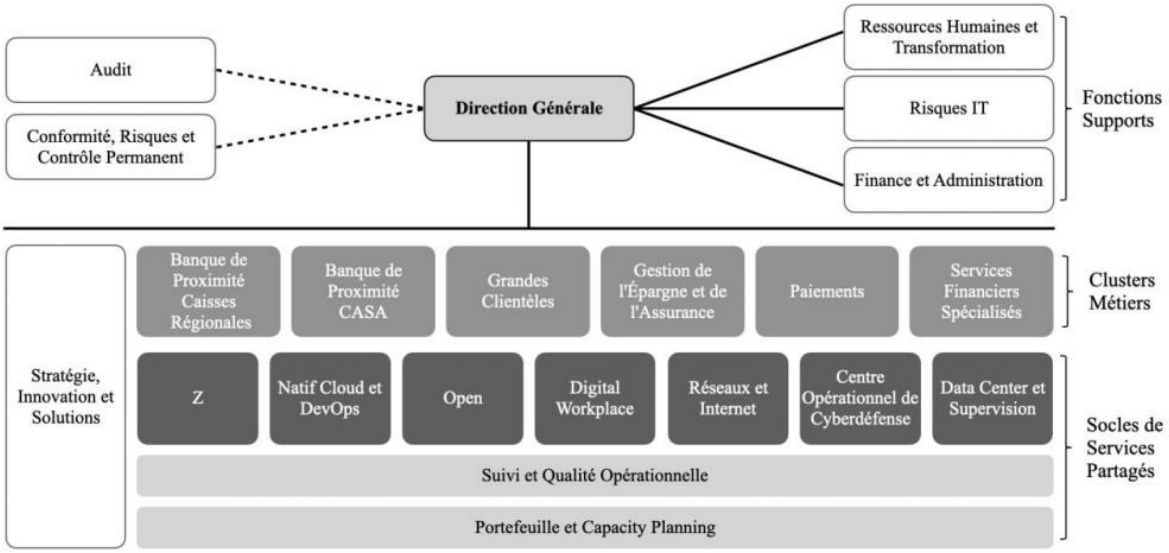

Figure 6 : Schéma organisationnel de CA-GIP

Avec cette fusion le cluster « Grande Clientèles » dans lequel je travaille a besoin d'uniformiser les processus de gestion de projet. En effet, les équipes possédaient déjà leur propre outil de gestion de projets respectif. La mutualisation des outils nous a conduits à nous interroger sur la pertinence des fonctionnalités dans notre nouvel outil. Plusieurs applications existantes permettaient de répondre à nos besoins, nous avons donc dû analyser plusieurs scénarios pour mettre en place la nouvelle solution.

La solution retenue est celle de développer un nouvel outil de gestion de projet en prenant le meilleur de chaque outil déjà existant.

5.1. Impact sur l'organisation : conduite du changement

La mise en place de cette nouvelle solution implique de remettre en cause le statu quo. Il faut désormais changer des process sur lesquels les différentes entités se sont construites. Il faut donc réaliser au mieux le changement à venir.

Conduire ou accompagner le changement consiste à mettre en place des actions pour aider les équipes concernées à accepter, comprendre, adopter et s'approprier le changement. Cela implique de considérer l'impact du changement sur les différents aspects de leur travail, qu'ils soient relationnels, matériels, même psychologiques.

Conduire le changement, c'est également faire le lien entre une stratégie d'entreprise générale impliquant un ou plusieurs projets de transformation, et des collaborateurs plus ou moins prêts à cela. La finalité étant que chaque projet aboutisse et que les objectifs soient atteints, sans qu'il n'y ait de dérapage (de délais et/ou de budget), ni de mécontents.

Il existe de nombreuses méthodologies qui ont été développées pour aider les organisations à gérer leurs processus de transformation le mieux possible sur le plan humain.

Pour autant, la conduite du changement est trop souvent négligée et cela affecte directement la réussite des projets. Un exemple courant où nous possédons beaucoup de données statistiques concerne la gestion de la relation client avec la mise en place de *CRM (Customer Relationship Management)* : alors que le marché du *CRM* est plutôt florissant (36,5 milliards de dollars en 2017 selon Gartner), le taux d'échec des projets *CRM* est de l'ordre de ~30% à ~60%, et peut même atteindre ~70%. Parmi les principales raisons évoquées, un manque d'accompagnement et une adoption difficile par les utilisateurs. Par ailleurs, d'après une autre étude de Gartner datant de 2012, ~30% voire ~50% du budget des projets *CRM* ayant réussi (qui représentaient alors ~20% de tous les projets) étaient consacrés à la conduite du changement. On peut facilement faire la comparaison avec le marché global de l'*API* qui est actuellement en forte croissance.

On constate avec ces statistiques l'importance de la conduite du changement et la nécessité d'y consacrer des suffisamment de ressources (non seulement en budget, mais aussi en temps et en personnel) : la conduite du

changement doit être considérée comme un investissement nécessaire à la réussite d'un projet, et non comme une dépense dont on pourrait s'abstenir.

Faire de la conduite du changement :

Appliquer une méthodologie

Suivre une méthodologie permet à la gestion du changement de coexister avec le projet, de n'en oublier aucun aspect, de proposer un plan d'action adapté à la situation, et de réorienter ce plan si nécessaire. Sans méthodologie, la gestion du changement peut être simplement négligée ou réduite à quelques actions de formation, ce qui est rarement suffisant.

Dédier une personne ou une équipe à la conduite du changement

Le contenu de la gestion du changement dépend du contexte, de la taille du projet, des équipes impliquées et de l'entreprise. Il est recommandé de dédier au moins une personne à la gestion du changement, car elle doit être gérée indépendamment de la gestion opérationnelle du projet. Dans certains cas, il est possible et même recommandé de faire appel à un consultant externe.

Établir un diagnostic

Le diagnostic est l'étape essentielle avant toute action. Il est important de bien identifier le changement et ce qu'il implique pour chaque catégorie d'employés, surtout si plusieurs projets de transformation sont prévus simultanément. Sur cette base, un plan d'action approprié peut ensuite être élaboré.

Incarner le changement

Le changement doit être conduit au niveau de la direction de l'entreprise, par une personne dont la légitimité est avérée. Son rôle est de transmettre une vision claire de l'évolution souhaitée et des objectifs, et de soutenir visiblement les acteurs du changement (managers, employés, équipe de conduite du changement).

Impliquer les managers

Les managers sont en première ligne de tout projet de changement, car ce sont eux qui vont le gérer au quotidien. Leur tâche est délicate, car ils doivent s'approprier la stratégie de l'entreprise, être capables de l'expliquer, mettre en œuvre les changements opérationnels tout en gérant les réactions humaines associées, et enfin montrer l'exemple à leurs équipes.

Impliquer les collaborateurs

Les décisions de changement imposées unilatéralement d'en haut (top-down), sans explication et sans consultation, échouent presque toujours. Dans un tel contexte, les personnes qui se voient imposer un changement peuvent perdre leurs repères, être frustrées ou développer un sentiment de malaise au travail. Imposer un changement est donc le meilleur moyen de le voir échouer ou dévier.

Au contraire, une approche ascendante, basée sur l'écoute des besoins et des attentes des personnes concernées, est beaucoup plus profitable. Impliquer les collaborateurs dès le début du projet, par exemple, permet de les faire participer en sollicitant leurs connaissances et leur expérience du terrain.

Gérer les résistances

Naturellement, un changement génère d'abord une "résistance" chez ceux qu'il concerne au premier chef. Elle se manifeste sous forme d'opposition ou de passivité, et peut s'expliquer par la peur, l'incertitude sur la situation à venir. Le rôle de la gestion du changement est d'inverser cette tendance en expliquant à chacun ce qu'il va gagner. Parfois, une réunion d'information suffira. Dans d'autres cas, il faudra plus de temps et des discussions individuelles.

Créer les conditions pour le changement

Toutes les actions doivent être mises en place pour que le changement soit réussi. La formation peut être utile. Une autre façon de faciliter le changement est de demander un retour régulier aux employés, par exemple au moyen de questionnaires de satisfaction.

Piloter la conduite du changement

Le suivi est un aspect majeur de la gestion du changement. Il établit le lien entre l'avancement du projet par rapport aux objectifs fixés et en termes d'acceptation. Il permet de décider des ajustements nécessaires.

La conduite du changement permet d'accompagner les collaborateurs et de leur faire accepter le changement, lorsqu'un ou plusieurs projets créent une rupture dans leurs pratiques habituelles. Cependant, elle s'inscrit dans le contexte historique de l'entreprise et dans des trajectoires individuelles spécifiques qui ne rendent pas forcément la tâche facile. Il est d'autant plus important, dans ce cas, de maîtriser les principes et les techniques, que nous avons exposés ici. Néanmoins, il peut y avoir une grande différence entre le temps nécessaire à un projet pour atteindre le changement souhaité et le temps dont chaque individu a besoin pour s'adapter. Dans un contexte évolutif, le rôle de la gestion du changement est donc double : faciliter une transition donnée à un moment donné, et initier, voire ancrer, une habitude de changement dans l'organisation, afin de la préparer aux évolutions futures.

Ces notions de conduite du changement sont nécessaires à la réussite de projet entrainant une transition numérique comme les *API*. Cette section avait pour but de montrer la manière dont le changement est géré chez *CA-GIP* et de sensibiliser aux conséquences de l'implémentation des *APIs*.

5.2. Utilisation des APIs

La nouvelle solution en cours de développement qui porte le nom de Stargate utilise donc une *API*. C'est un des enjeux majeurs de cette nouvelle solution. Cela permettra à terme d'éviter la double saisie et d'avoir les informations de manière centralisée sans avoir à les chercher dans les différents outils de l'organisation.

Pour faciliter et industrialiser nos développements, nous utilisons Laravel. C'est un des frameworks PHP, leader du marché, qui propose un ensemble d'outils permettant de concevoir rapidement l'architecture d'un logiciel.

L'*API* utilisé par notre nouvelle solution est l'*API* de Triskell. Elle a été développée du côté de l'éditeur avec une architecture *REST*.

6. Bilan et conclusion

L'évolution des paradigmes entraine certaines tendances dans les méthodes de conception et d'utilisation des *APIs* comme nous pouvons le voir dans la figure ci-dessous.

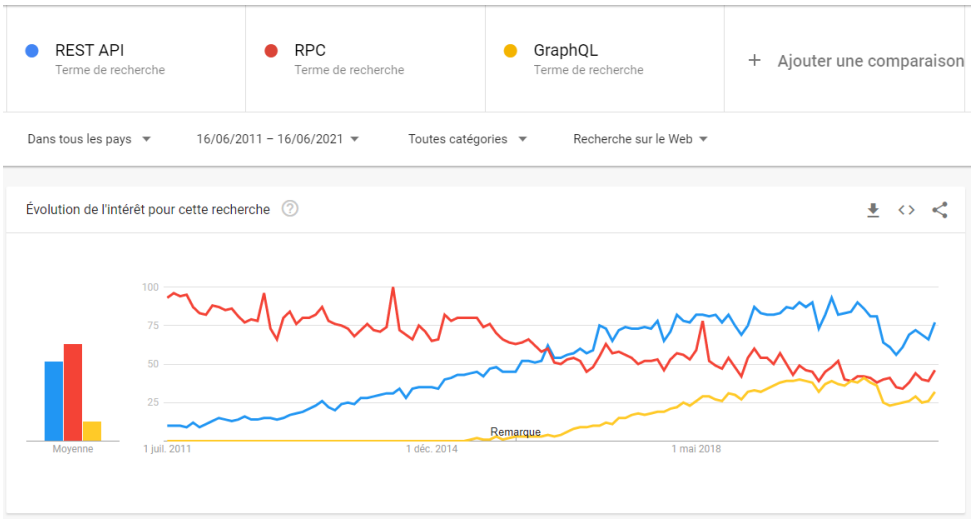

Figure 7 : Google trends des termes « REST API », « RPC », « GraphQL » de 2011 à juin 2021

Mais il est important de savoir se détacher du paradigme contemporain afin de proposer une architecture cohérente avec nos besoins.

À la suite de la lecture de ce document, nous sommes donc en mesure de définir précisément quelle méthode d'utilisation correspond le mieux à une situation donnée, tout en étant capable de justifier techniquement nos choix sans être influencée par le paradigme actuel.

Après l'analyse réaliser, on peut dire que la meilleure méthode d'utilisation pour *CA-GIP*, dans notre cluster « Grande Clientèle » est la méthode de conception et d'utilisation *REST*.

Dans ce cas précis, nous avons besoin de réaliser de la récupération de ressource et de la persistance en base de données. L'architecture *REST* et *GraphQL* sont donc totalement adaptées.

On peut donc se demander s'il ne serait pas nécessaire de transformer l'architecture *REST* de notre solution en architecture *GraphQL*.

Dans la pratique l'utilisation d'une *API RESTful* est la plus appropriée. Dans le contexte actuel aucune architecture *GraphQL* n'est nécessaire. Le gain de performance sera minime en dépit du temps passer à implémenter cette solution.

Néanmoins s'il y a une évolution de contexte dans les prochaines années, par exemple, l'outil devient la norme d'un ou plusieurs autres clusters. Cela entrainerait une forte hausse du nombre d'utilisateurs et donc de données. À ce moment-là, il sera intéressant d'implémenter une architecture *GraphQL* pour éviter que les performances diminuent.

7. Table des figures

Figure 1 : Démonstration conceptuelle de l'architecture d'une application utilisant une API……..6

Figure 2 : Démonstration conceptuelle de l'utilisation du protocole RPC……7

Figure 3 : Graphe d'objets possédant des relations complexes……22

Figure 4 : Courbe d'adoption des APIs……33

Figure 5 : Nombres d'APIs par secteur……33

Figure 6 : Schéma organisationnel de CA-GIP……34

Figure 7 : Google trends des termes « REST API », « RPC », « GraphQL » de 2011 à juin 2021……36

8. Sources

8.1. Sitographie

- RFC 1057 [en ligne], 2021 - [consulté en mai 2021]. Adresse internet : https://datatracker.ietf.org/doc/html/rfc1057
- RFC 707 [en ligne], 2021 - [consulté en mai 2021]. Adresse internet : https://datatracker.ietf.org/doc/html/rfc707
- RFC 3023 [en ligne], 2021 - [consulté en mai 2021]. Adresse internet : https://datatracker.ietf.org/doc/html/rfc3023
- RFC 7303 [en ligne], 2021 - [consulté en mai 2021]. Adresse internet : https://datatracker.ietf.org/doc/html/rfc7303
- RFC 3470 [en ligne], 2021 - [consulté en mai 2021]. Adresse internet : https://datatracker.ietf.org/doc/html/rfc3470
- RFC 7158 [en ligne], 2021 - [consulté en mai 2021]. Adresse internet : https://datatracker.ietf.org/doc/html/rfc7158
- RFC 4627 [en ligne], 2021 - [consulté en mai 2021]. Adresse internet : https://datatracker.ietf.org/doc/html/rfc4627
- RFC 7159 [en ligne], 2021 - [consulté en mai 2021]. Adresse internet : https://datatracker.ietf.org/doc/html/rfc7159
- RFC 8259 [en ligne], 2021 - [consulté en mai 2021]. Adresse internet : https://datatracker.ietf.org/doc/html/rfc8259
- RFC 2119 [en ligne], 2021 - [consulté en mai 2021]. Adresse internet : https://datatracker.ietf.org/doc/html/rfc2119
- RFC 4287 [en ligne], 2021 - [consulté en mai 2021]. Adresse internet : https://datatracker.ietf.org/doc/html/rfc4287
- W3C MISSION [en ligne], 2021 - [consulté en mai 2021]. Adresse internet : https://www.w3.org/Consortium/mission
- Quelques exemples d'API afin de comprendre son fonctionnement [en ligne], 2021 - [consulté en mai 2021]. Adresse internet : https://www.axysweb.com/decouvrir-le-fonctionnement-des-api-grace-a-des-exemples-concrets/
- Introduction aux API Web [en ligne], 2021 - [consulté en mai 2021]. Adresse internet : https://developer.mozilla.org/fr/docs/Learn/JavaScript/Client-side_web_APIs/Introduction
- La conception d'API, qu'est-ce que c'est ? [en ligne], 2021 - [consulté en mai 2021]. Adresse internet : redhat.com/fr/topics/api/what-is-api-design#:~:text=La%20conception%20d'API%20désigne,des%20développeurs%20et%20des%20utilisateurs.
- Conception d'une API web [en ligne], 2021 - [consulté en mai 2021]. Adresse internet : https://docs.microsoft.com/fr-fr/azure/architecture/best-practices/api-design
- What is API: Definition, Types, Specifications, Documentation [en ligne], 2021 - [consulté en mai 2021]. Adresse internet : https://www.altexsoft.com/blog/engineering/what-is-api-definition-types-specifications-documentation/
- Understanding RPC, REST and GraphQL [en ligne], 2021 - [consulté en mai 2021]. Adresse internet : https://apisyouwonthate.com/blog/understanding-rpc-rest-and-graphql
- Remote procedure call (RPC) [en ligne], 2021 - [consulté en mai 2021]. Adresse internet : https://whatis.techtarget.com/fr/definition/Remote-procedure-call-RPC
- Extensible Markup Language (XML) 1.0 (Fifth Edition) [en ligne], 2021 - [consulté en mai 2021]. Adresse internet : https://www.w3.org/TR/REC-xml/#sec-origin-goals
- Introducing JSON [en ligne], 2021 - [consulté en mai 2021]. Adresse internet : https://www.json.org/json-en.html
- JSON-RPC 2.0 Specification [en ligne], 2021 - [consulté en mai 2021]. Adresse internet : https://www.jsonrpc.org/specification
- Why we chose JSON-RPC over REST [en ligne], 2021 - [consulté en mai 2021]. Adresse internet : https://joost.vunderink.net/blog/2016/01/03/why-we-chose-json-rpc-over-rest/
- Know your API protocols: SOAP vs. REST vs. JSON-RPC vs. gRPC vs. GraphQL vs. Thrift [en ligne], 2021 - [consulté en mai 2021]. Adresse internet : https://www.mertech.com/blog/know-your-api-protocols
- Une API REST, qu'est-ce que c'est ? [en ligne], 2021 - [consulté en mai 2021]. Adresse internet : https://www.redhat.com/fr/topics/api/what-is-a-rest-api#:~:text=Une%20API%20REST%20(%C3%A9galement%20appel%C3%A9e,avec%20les%20services%20web%20RESTful.

- GraphQL is the better REST [en ligne], 2021 - [consulté en mai 2021]. Adresse internet : https://www.howtographql.com/basics/1-graphql-is-the-better-rest/
- Web API Design Maturity Model [en ligne], 2021 - [consulté en mai 2021]. Adresse internet : http://amundsen.com/talks/2016-06-wsrest/index.html
- Modèle de maturité de Richardson - Richardson Maturity Model Modèle de maturité de Richardson [en ligne], 2021 - [consulté en mai 2021]. Adresse internet : https://fr.xcv.wiki/wiki/Richardson_Maturity_Model
- Richardson Maturity Model [en ligne], 2021 - [consulté en mai 2021]. Adresse internet : https://martinfowler.com/articles/richardsonMaturityModel.html
- 5 Steps to API Adoption [en ligne], 2021 - [consulté en mai 2021]. Adresse internet : http://www.acloudfan.com/steps-api-adoption

8.2. Vidéographie

- Qu'est-ce qu'une API REST ? [en ligne], 2021 - [consulté en mai 2021]. Adresse internet : https://www.youtube.com/watch?v=lsMQRaeKNDk
- What is REST API? | REST API Tutorial | REST API Concepts and Examples | Edureka [en ligne], 2021 - [consulté en mai 2021]. Adresse internet : https://www.youtube.com/watch?v=rtWH70_MMHM
- CHOISIR ENTRE UNE API RPC, SOAP, REST, GRAPHQL ET SI LE PROBLÈME ÉTAIT AILLEURS - F-G. RIBREAU [en ligne], 2021 - [consulté en mai 2021]. Adresse internet : https://www.youtube.com/watch?v=UKrS_eXZfHw
- GraphQL Explained in 100 Seconds [en ligne], 2021 - [consulté en mai 2021]. Adresse internet : https://www.youtube.com/watch?v=eIQh02xuVw4
- What Is GraphQL? [en ligne], 2021 - [consulté en mai 2021]. Adresse internet : https://www.youtube.com/watch?v=VjXb3PRL9WI

8.3. Bibliographie

- MASSE Mark, REST API Design Rulebook, O'REILLY 2011.
- RICHARDSON Leonard, RUBY Sam, RESTful Web APIs, O'REILLY 2013.