Conception, maintenance et évolution non-cassante des API REST

Antoine Cheron

▶ **To cite this version:**

Antoine Cheron. Conception, maintenance et évolution non-cassante des API REST. Web. Université Rennes 1, 2021. English. NNT : 2021REN1S076 . tel-03589271

HAL Id: tel-03589271
https://theses.hal.science/tel-03589271

Submitted on 25 Feb 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

DOCTORAT BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES 1

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par
Antoine CHERON

Design, maintenance and break-free evolution of REST APIs

Thèse présentée et soutenue à Rennes, le 17 Décembre 2021
Unité de recherche : IRISA - Équipe DiverSE

Rapporteurs avant soutenance :

Naouel Moha	Professeure à l'École de Technologie Supérieure de Montréal
Philippe Merle	Chercheur à l'Institut National de Recherche en Informatique et en Automatique

Composition du Jury :

Président :	Yerom David Bromberg	Professeur à l'Université de Rennes 1
Examinateurs :	Philippe Collet	Professeur à l'Université Côte d'Azur
Dir. de thèse :	Olivier Barais	Professeur à l'Université de Rennes 1
Co-dir. de thèse :	Johann Bourcier	Maître de conférences à l'Université de Rennes 1

Invité(s) :

Djamel Eddine Khelladi	Chercheur au CNRS
Antoine Michel	Tech Advisor et encadrant industriel de cette thèse à Fabernovel

Acknowledgement

First of all, I would like to thank the members of the jury. A very big thank to Naouel Moha and Philippe Merle for having accepted to report this work. Thank you David Bromberg, this is an honor to have you as an examiner. Thank you Philippe Collet for accepting to review this thesis.

Thank you so much Johann (Bourcier) and Olivier (Barais) for your unconditional support, your kindness and your humanity. For making each meeting a pleasant and enriching moment, for your trust, for not giving up when I asked to spend most of my time in Paris (then Marseille, then Lyon), and for always following me in the direction I wanted to give to this thesis. Thank you for making this thesis a great experience! And of course, many thanks to you, Johann, for having initiated all this journey by attracting me to the lab.

I would also like to thank Djamel for his calm, his patience and the quality of his proofreading. Thank you for being so open-minded and professional. You very much helped me significantly increase the quality of my research.

Naturally, I would like to thank the whole DiverSE team for their benevolence, their open-mindedness and their love for good research. I am honored to have been part of such a human, passionate and enriching team. It is thanks to you that I discovered the values of science, learned a lot and also understood to what extent the choice of a team is determining.

Thanks to you too, Antoine (Michel), for giving me this wonderful opportunity! You have inspired me so much, pushed me to go beyond my limits, to dig deeper and to take on subjects that seemed out of reach. It is always a great pleasure to have these very technical discussions with you.

From Fabernovel, I also thank Remi Dewitte who pushed me hard to go beyond my limits, with who I've always had very interesting discussions and learned so much. Many thanks to you Bruce Mwizerwa for being so inspiring, for challenging me to make things crystal clear and for your unconditional support.

I would like to dedicate this thesis to my two parents, Florence and Jean-François. Thank you for having transmitted to me the value of always doing what I like and for having always trusted me. You taught me to test, to try things, and always supported my choices. Without that I would never have dared to follow this path that has given me so much, thank you!

(En Français: C'est à mes deux parents, Florence et Jean-François, que j'aimerais dédier cette thèse. Merci de m'avoir transmis comme valeur de toujours faire ce qui me plaît et de m'avoir toujours fait confiance. Vous m'avez appris à tester, à tenter les choses, et avez toujours

soutenu mes choix. Sans ça je n'aurais jamais osé suivre ce chemin qui m'a tant apporté, merci !)

The last person to whom I would like to say a big thank you is the one who is very close to me at the moment of writing these lines. Thank you so much, Marie, for listening to me think out loud so many times, for calming the roller-coster of feelings experienced when submitting an article and receiving the result. Thank you for making everything simpler, lighter and more fun.

RÉSUMÉ EN FRANÇAIS

Motivations

Autrefois fermé, le système d'information d'une organisation était fortement typé et contrôlé. Aujourd'hui, il a migré vers le World Wide Web, un environnement ouvert, non typé et hétérogène. Alors que cette transition a permit d'interconnecter les SI et de faciliter l'évolution des applications, elle a aussi apporté de nouveaux challenges.

Le Web est principalement composé de deux types de composants applicatifs : les applications et les services. Les applications, qu'elles soit des applications mobiles ou Web, c'est à dire ayant vocation à être utilisées depuis un navigateur, ont en commun qu'elles sont une sorte de texte à trous, elles ne peuvent pas stocker de données. Elles communiquent donc avec des services pour récupérer de la donnée et ainsi personnaliser l'application pour chaque utilisateur. Ce sont donc les services qui possèdent et gèrent la donnée. Pour rendre possible cette communication entre les applications et les services, ces derniers mettent à disposition des applications des APIs dont la majorité suit l'architecture REST. Les APIs sont une liste de fonctions qui peuvent être appelées, certaines retournant des données, d'autres la modifiant.

Cependant, un premier problème est qu'il faut identifier la ou les API REST qui peuvent fournir le service attendu. Une fois identifiées, il est nécessaire de les choisir. Et si jamais l'une d'entre elles arrête de fonctionner, il faut lui trouver une remplaçante. Toutes ces étapes sont faites manuellement et sont donc chronophages.

Un second problème est que la communication entre une application et une API REST doit être codée par un humain. En effet, les techniques qui automatisent cette étape ne s'appliquent pas API REST. En conséquence, cette seconde étape est elle aussi chronophage, et source d'erreur.

De plus, on observe que lorsque l'API évolue, il faut que ses évolutions soient répercutées dans le code de ses clients, souvent manuellement. En effet, le code de l'application doit respecter le contrat d'une version précise de l'API, à la lettre, créant ainsi un fort couplage entre les deux composants. De ce fait, la modification d'un élément de ce contrat oblige à modifier le code (ex: fullname est changé en fullName). Cette troisième phase est elle aussi chronophage et source d'erreur car il faut détecter le changement, appliquer les modifications, vérifier que des régressions n'ont pas été introduites puis déployer une nouvelle version de l'application.

Le dernier challenge encore ouvert est la mise en place de tests pour garantir que l'API se comporte comme attendu. De nombreuses techniques existent pour tester un logiciel, et d'autres

sont créées régulièrement. Par exemple, ces dernières années de nouvelles approches reposant sur l'intelligence artificielle voient le jour. Cependant, ces techniques n'ont pas toutes été appliqués aux API REST. Des recherches sont menées en ce sens.

Objectifs

Dans ce contexte, nous nous sommes intéressés aux problématiques du choix des technologies permettant la création d'une API REST, ainsi qu'à la maintenance et à l'évolution de ces APIs. Nous nous sommes ainsi fixés les objectifs suivants :

— faciliter la découverte et la sélection des technologies les plus pertinentes pour la mise en place d'une API REST qui respectent un ensemble de propriété défini

— réduire la maintenance nécessaire sur l'application lorsque les APIs REST qu'elle utilise évoluent

Contributions

Premièrement, il existe de nombreuses technologies pour créer et documenter une API. Choisir les technologies les plus pertinentes pour un projet est une tâche difficile. Comme première contribution de cette thèse, nous établissons des critères permettant de comparer ces technologies. Pour cela, nous avons dans un premier temps recensé 36 technologies qui permettent de concevoir, implémenter et documenter une API REST. La sélection ne s'est pas limitée aux technologies conçues spécfiquement pour les API REST, elle a été étnedue aux technologies qui peuvent être appliqués aux API REST. Nous avons lu la documentation complète de chaque technologie afin d'en extraire des critères qui permettent de les comparer. Ces critères s'inscrivent dans les catégories définies par Siqueira et. al. dans leur modèle de maturité des API Semantic REST[**?**]. Ensuite, nous utilisons ces critères pour comparer les technologies existantes et proposons ainsi trois matrices de comparaison. Enfin, pour simplifier cette selection, nous avons développé un assistant open-source disponible sur le Web, qui guide le développeur dans son choix.

La seconde contribution de cette thèse est liée à la maintenance et l'évolution des APIs REST. La littérature existante ne permet pas de faire évoluer une API REST librement, sans risquer de rendre inopérantes les applications qui s'en servent (leurs clients). Ainsi, dans ce second travail, nous proposons une nouvelle approche à la co-évolution des APIs REST et de leurs clients. Nous affirmons que les clients des API REST ne devraient pas être maintenus manuellement mais devraient être capables de s'adapter aux évolutions typiques des API REST, au moment de l'exécution, sans rupture et sans modification de leur code. Pour ce faire, les clients des API REST ont besoin d'une documentation riche. Nous avons donc étudié quelles informations

structurelles et contextuelles devraient être documentées dans les API pour que les interfaces utilisateur Web soient *evolvable-by-design*. Nous avons identifié qu'en suivant 7 règles régissant la documentation de l'API et les données qu'elles renvoient en répondant à ses clients, il est possible de créer de telles interfaces utilisateur Web. Pour mettre au point cette contribution, nous avons tout d'abord questionné des développeurs sur les types d'évolution d'API REST qu'ils ont rencontré. Nous avons découvert sept nouveaux types. Ensuite, nous avons adopté une approche à la fois analytique et expérimentale pour identifier ces 7 règles. L'évaluation de cette nouvelle approche repose sur 2 études : une qualitative et une quantitative. L'approche quantitative teste la capacité de l'approche à supporter un maximum de cas d'évolution. A défaut de trouver un projet open source en fournissant de nombreux, nous avons créé une API réaliste, qui présente 110 évolutions. Nous avons ainsi créé deux variantes d'une application Web qui utilise cette API : une version à l'implémentation traditionnelle et une autre, qui utilise l'approche proposée. A chaque évolution de l'API, nous avons testé les deux variantes de l'application pour voir ce qui ne fonctionnait plus et avons procédé à la modification de leur code quand c'était nécessaire. Nous avons ainsi mesuré le nombre d'évolutions qui cassait le client et le nombre de lignes de code ayant été modifiées pour les réparer. Nous avons constaté que l'approche était applicable sur ce cas. Ensuite, l'étude qualitative vérifie si l'approche peut être implementée dans des projets open-source existants. Nous en avons trouvé 5 sur GitHub. Au total, ils implémentaient 10 types d'évolution. Les résultats de cette seconde évaluation ont validé les résultats de la première : 27 des 29 types d'évolution d'API REST peuvent être supportés. Ceux non supportés représentaient moins de 1% des évolutions réelles selon une précédente étude. Tout le code de ces expérimentations est disponible sur GitHub [1], ce qui inclus une librairie JavaScript qui a été créée pour simplifier la création d'interfaces Web evolvavle-by-design.

Perspectives

Les travaux menés dans cette thèse ont permi d'identifier d'autres challenges concernant la co-évolution de systèmes utilisant des APIs REST. Ils ont aussi ouvert de nouvelles pistes.

Afin tester l'attractivité de l'approche proposée en seconde contribution, et l'effort nécessaire à sa mise en pratique, nous proposons, dans une première perspective, de mener une étude croisée. C'est à dire, de faire tester l'approche à des développeurs expérimentés. En fonction des résultats, si l'approche est complexe à comprendre ou pas assez attractive, des efforts de communication et d'enseignement seront nécessaires, mais ils sortent du cadre de la recherche scientifique. De notre point de vue, la dernière chose à aborder est la facilité de mise en œuvre. D'une part, il faut des études avancées sur l'expérience utilisateur pour concevoir l'API de la bibliothèque côté client, qui sont liées à l'industrialisation et à l'ingénierie. D'autre part, nous identifions des

1. `https://github.com/evovlable-by-design`

sujets de recherche intéressants. Nous pensons que l'un des principaux défis est la sélection et la manipulation des ontologies. Ainsi, comme seconde perspective, nous proposons d'explorer des solutions pour proposer une recommandation d'ontologie directement dans l'éditeur de code, en s'appuyant sur des techniques d'apprentissage automatique pour maximiser la normalisation et obtenir une bonne précision, ainsi qu'en exploitant le code source et la documentation de l'API.

Maintenant, si nous imaginons que les développeurs utilisent notre approche, les API REST seraient décrites sémantiquement et les applications Web seraient évolutives par conception. Dans un tel contexte, nous voyons des sujets intéressants à explorer. Un premier sujet est l'utilisation de GraphQL pour interroger un ensemble d'API REST sémantiques sans connaissance préalable de celles-ci. Un deuxième sujet est la description et l'automatisation partielle de tests fonctionnels.

Contenu du manuscrit

Ce manuscrit est organisé de la manière suivante :

— Le chapitre Chapter 1 présente le contexte de ce travail en détail. Il explique notamment comment le Web a évolué, puis passe en revue les travaux connexes. Sur cette base, il souligne les principaux défis abordés dans cette thèse.

— Le chapitre Chapter 2 présente un aperçu des deux contributions de cette thèse.

— Le chapitre Chapter 3 présente la première contribution de ce travail. Il détaille la comparaison des similitudes et des différences entre 36 technologies, propose trois matrices de comparaison qui aident les architectes et les développeurs d'API REST à choisir les technologies les plus adaptés à leurs besoins. Il démontre comment les matrices peuvent être utilisées sur un exemple réel et présente l'assistant en ligne open-source que nous avons développé pour faciliter ce processus.

— Le chapitre Chapter 4 présente la deuxième contribution de ce travail. Il présente une nouvelle approche de la coévolution sans rupture des API REST et de leurs clients, en particulier pour les interfaces utilisateur Web. Il identifie quelles métadonnées doivent être partagées par l'API REST à ses clients et à quel moment pour permettre la conception de clients *evolvable-by-design*. Il s'agit de clients Web capables de s'adapter, lors de l'exécution, aux évolutions typiques d'une API REST sans casser ni nécessiter la mise à jour de leur code. L'approche proposée est évaluée à travers une étude quantitative et qualitative. Enfin, les limites de l'approche proposée sont discutées.

— Pour finir, le chapitre Chapter 5 conclut la thèse et présente quelques perspectives de recherche.

Table of Contents

List of acronyms

AJAX	Asynchronous JavaScript And XML
API	Application Programming Interface
BPEL	Business Process Execution Language
BPMN	Business Process Model and Notation
CDN	Content Delivery Network
CQRS	Command Query Responsibility Segregation
CRUD	Create Read Update Delete
CSS	Cascading Style Sheet
DIF	Data Interchange Format
DOM	Document Object Model
DSL	Domain-Specific Language
HATEOAS	Hypermedia As The Engine Of Application State
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IDL	Interface Description Language
JAMstack	Javascript API Markup stack
JVM	Java Virtual Machine
MDE	Model-Driven Engineering
MVC	Model View Controller
OWL	Web Ontology Language
RDF	Resource Description Framework
REST	REpresentational State Transfer
SDK	Software Development Kit
SOAP	Simple Object Access Protocol
SPA	Single-Page Application
SWS	Semantic Web Services
UI	User Interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WWW	World Wide Web
WYSIWYG	What You See Is What You Get
XML	Extensible Markup Language

List of Figures

List of Tables

Introduction

Context

The World Wide Web is probably the technology that has known the fastest adoption rate ever. Its major strength has been its ability to use the Internet to enable the users using a computer of any brand to access documents (later named websites) shared by other users of the World Wide Web. To accommodate the evolving needs of its users, it evolved a lot since its creation in 1989. It has therefore been the subject of a lot of research, both before and after this year.

Convinced of the usefulness of the technology, universities and research labs rapidly joined the World Wide Web [3]. Later, companies joined the movement and started selling goods and services online. However, they needed to create more appealing websites. So, new technologies emerged to animate and style websites, namely Flash, Java Applets, CSS, and others. Yet, before the 2000s, Internet connections were so slow that most users could only consume content on the Web but could not create content.

Thanks to ADSL (Asymmetric Digital Subscriber Line), the speed of Internet connections increased significantly. It enabled people to not only download Web pages faster but also upload information in a reasonable amount of time, which transformed how people used the Web. Indeed, people started to use the Web as a collaborative platform where social media became the most used category of Web sites [4]. This switch corresponds to the so-called Web 2.0 that started around 2004. Before 2004, Web sites were independent. But with the advent of the collaborative Web, users began to seek to use the Facebook "like" button outside of Facebook. Websites thus started to embed other website features. It increased the number of communications between the websites.

Another event significantly increased the number of communications between websites. A few years later, Apple announced the App Store on the iPhone. Smartphones will not use Web applications but dedicated mobile applications. Since then, people used more than one device to access the World Wide Web and wanted to access the same content from all their devices. To enable this, Web applications have been more and more divided up. As a result, the application layer of the Web became a global service-oriented architecture [5]. Indeed, when mobile applications did not exist, a whole website was a single application with a single code base. However, when mobile applications had to be added into the system, the layer managing the data has been separated from the website itself, which role is to display the data. Data management

has been placed within servers, and Web and mobile applications became standalone software that consume and display the data held and managed by the servers. The Web and mobile applications could therefore use the same data by contacting the same server. Within this new paradigm, Web and mobile applications of one provider can also use the data of other providers. Cross-organization exchanges thus increased a lot. These servers are then not solely named servers but services because they are designed to provide fine-grained services to other Web systems. For example, a service can be responsible for the ordering of goods, another service for the payment, and in a different context, a service can be dedicated to providing weather forecasts.

As a result, the Web is now composed of a lot of services that are either employed by other services or by user interfaces. So, today we describe the Web as a global service-oriented architecture.

The technologies powering the Web have matured significantly over the past 30 years, and we see every day how fluid Web applications have become, to the point where the differences between native and Web applications have become very subtle. Complex websites and Web applications load fast and progressively, video streaming is incredibly smooth, we can collaborate in real-time on the edition of a document with people who are anywhere on earth and all the services that we use can synchronize in real-time. Something that we do not like, however, is when these technologies are used to serve us personalized ads, but at least it demonstrates how effective cross-site data exchange can be.

When we take a step back at what the Web is composed of, we can see Web applications running in Web browsers, mobile applications running on smartphones, and servers all communicating with each other. While the technologies embedded in the Web browsers and smartphones can be improved to ease or shorten developments, to enable new usages such as 3D rendering and Augmented Reality, or to make downloads even faster, in this thesis we will focus on the interface contracts driving the interactions between the systems of the Web.

Web APIs are used to enable the systems of the Web to communicate with each other. Basically, an API is a catalog of functions that a software can invoke on another software. In the Web, a client (a service or user interface) sends a request for data or action to a server (a service). The request takes the form of an HTTP request. The server processes the request and sends a response with the data or result represented in a given format, most often XML or JSON but many others have been proposed. Web APIs present many challenges.

Challenges related to Web APIs

Web APIs present many challenges. While numerous research answered challenging questions, there are still open challenges to explore. In this section, we present the main challenges that

we are aware of.

First challenge: automated API selection and composition

Each Web API exposes its own set of functions to its present and future consumers. And on the other hand, each consumer has its own functional requirements that a single Web API does not always cover completely. There are therefore many consumers of Web APIs that need to use multiple Web APIs.

In this context, an open challenge is the following: as a Web application, what are the Web APIs that I should use? This is a challenge for many reasons. First, we need the ability to discover the available Web APIs. Second, we need to perform a selection between them, which requires the consideration of multiple properties such as the available features, the pricing, the service-level agreements, and so on. In addition, one goal is to make the use of multiple Web APIs as seamless as possible, which requires verifying the compatibility between the Web APIs.

Although this challenge is not new and was a hot topic in the early 2000s within the communities of Web Services and the Semantic Web, this is still an open challenge as recent publications are still exploring new solutions [6].

Second challenge: Web API testing

A second challenge regarding Web APIs is to test that they behave as expected by their designers. While a significant amount of research has already been conducted on this topic, there are still many open questions. For example, in recent years, research has been conducted to explore the use of machine learning techniques for the testing of Web APIs [7]. Besides, researchers apply techniques that have proven to be effective in other kinds of software to assess their applicability and measure their performance regarding Web APIs [8]. Model-driven techniques are also explored [9] and sometimes used to generate test cases [10].

Building on the publications of recent years, additional research is necessary to apply the machine learning techniques to more cases and domains, other techniques may be tested on Web APIs and model-driven techniques should be applied to more domains.

Third challenge: selecting the right set of technologies to achieve a target functionality level

To create a Web API, one must start by specifying its requirements. For this purpose, many design methodologies are available and maturity models help the designer select a target functionality level. Then, the creator must select a set of technologies for the design, documentation, implementation, and testing of the API.

This step is time-consuming and error-prone for Web API designers. Indeed, since the creation of the Web, many research and industry initiatives proposed design methods, frameworks, interface descriptions languages, and data-interchange formats for Web APIs. There is therefore a wide variety of technologies available. In addition, they are all documented in plain text documents. As a result, to select the set of technologies that is best suited to achieve a target functionality level, a software architect should read the documentation of each technology and then make a manual comparison and selection, which is time-consuming and error-prone.

It is thus an open challenge to assist developers and architects in the selection of the proper set of technologies to use to achieve a target functionality level for a given Web API.

Fourth challenge: evolution

The major challenge regarding Web APIs is their evolution. Indeed, to make the communication between two web systems effective, the client must send a very precise request to the Web API. Any mistake would make the communication ineffective. And on top of that, if the developers of the service offering the Web API change something, the code of the client will have to be updated manually by developers to take the changes into account. This task is time-consuming and error-prone. The evolution of APIs, and not specifically Web APIs, has therefore been a major research topic in the software engineering community.

The common recommendation for service providers is to create different versions of their Web API. They can modify an existing version only by adding elements, **but any modification of a Web API is discouraged [11]. To modify things, it is recommended to create a new version**. There are so many rules constraining the evolution and design of REST APIs that it can become very confusing, if not overwhelming. For example, Zalando has 110 of them [2]. In addition, the developers of the client systems are usually reluctant to move to the next version, so to avoid losing clients, service providers tend to avoid changes. We can indeed easily observe this with very popular REST APIs. For example, the Twitter REST API introduced in 2006 is still in its first version. The second version has been announced in July 2020 and in July 2021, it is still in Early Access mode.

The problem with this approach is that the evolution of the Web APIs is highly constrained. Yet, the systems in the Web must be able to evolve with as much freedom as possible. Indeed, on the one hand, the needs of the users of the Web evolve constantly so the systems must be evolved and on the other hand, it is desirable to be able to improve the architectural properties of the Web APIs without fearing a breakage or losing users. We therefore argue that break-free evolution is an important property to bring to the Web.

2. Source: `https://opensource.zalando.com/restful-api-guidelines/`

There are actually many approaches to break-free evolution in the Web. The historical approach for the co-evolution of software is the use of Automated Program Repair techniques [12, 13, 14, 15, 16, 17]. They repair the code of the client of a third-party component. These techniques work well for the evolution of libraries but not in the Web, due to the heterogeneity of its components and because the source code of REST APIs is usually closed. In addition, they require to deploy a new version of the code, which can take up to six months in big corporations [18]. In another category, some approaches propose to place an adapter between the client and the server [19, 20, 21, 22]. The adapter, sometimes a proxy, is responsible for adapting the client's request to the evolving API. These approaches finally hide the evolutions from the client's point of view and are thus not able to manage all types of evolutions. In a third category, industrial and academic efforts studied the types of evolutions that can break clients [23, 24, 25]. They proposed guidelines on how to evolve REST APIs to avoid breakage. This is actually the most used approach currently, but instead of enabling break-free evolution, they considerably reduce the ability to evolve REST APIs. In a fourth category, the Semantic Web and Linked Data, extensive research leveraged the Resource Description Framework (RDF) aiming at semantically describing information on the Web. This research proposed solutions for the automatic discovery, integration, and composition of Semantic Web Services [26, 27]. Unfortunately, while promising, automatic approaches are not sufficiently precise to be trusted in a fully-autonomous environment. In addition, they are based on SOAP and XML, that Web developers have abandoned in favor of REST and JSON so they have not reached wide adoption. Besides, REST itself proposes a solution by recommending to include hypermedia controls in the responses to guide the user through the API similarly to whats hyperlinks do in websites. This is an effective solution when the Web API is used by a human who creates requests on the go but this is not sufficient for software. In addition, REST does not detail how to design, document, and use these hypermedia controls [28].

Contribution

In this thesis, we focus on REST APIs, the current de-facto standard for building Web APIs. We argue that REST APIs present the same challenges as other forms of Web APIs. The lessons learned from the studies on REST APIs can therefore be generalized to Web APIs. In addition, limiting the scope to REST APIs is a manner to design solutions that are more actionable for web engineers. However, we acknowledge that an additional effort can be necessary to adapt the technical solutions to the broader scope of any sort of Web API.

REST is an architectural style for the design of Web APIs. It lists a set of properties that are desirable to create scalable applications at internet scale leveraging on the rich semantics of the HTTP protocol. The goal of REST is to improve performance, scalability, simplicity, modi-

fiability, visibility, and reliability. This is however not a framework such as SOAP or CORBA. Practitioners are therefore free to achieve the properties emphasized by the REST architectural style with the technologies of their choice.

This thesis addresses the third and fourth challenges mentioned in the previous section. First, selecting the proper set of technologies to design and implement a Web API offering a given set of properties. Second, enabling the break-free co-evolution of Web APIs and their clients.

As a first contribution of this work, we propose three comparison matrices that help REST API providers choose the proper set of technologies to create a REST API with the desired properties and we propose a novel approach to the break-free co-evolution of REST APIs that enable web user interfaces to adapt to most types of API evolution at runtime, without breaking neither updating their code.

As the first step of this thesis, we reviewed the many existing technologies to design, document, and implement Web APIs. We identified 36 Interface Description Languages, Data-Interchange Format, and Implementation Frameworks.

Then, as a first contribution of this thesis [29], we proposed three comparison matrices based on the analysis of the differences and commonalities between these technologies. We showcase how they can be used on a realistic example. Then, to facilitate the selection of the best set of technologies suited for a Web API project, we have developed an assistant available online.

In the second contribution of this thesis, we argue that a novel approach to the break-free evolution of REST APIs is necessary. **We propose to stop precisely coding how to invoke the services in the client but to make it smarter instead. The client should be able to automatically adapt to evolving Web APIs at run-time without changing its code. We name these clients *evolvable-by-design*. We therefore study how to make clients of REST APIs evolvable-by-design.** In particular, we study Web User Interfaces. They are easier to make evolvable-by-design than autonomous systems because any input that the system can not obtain by itself can be asked to the user. It is thus a better candidate for the design of a novel approach.

To achieve this, Semantic REST APIs are needed. They are Web APIs aimed at promoting the adherence to REST principles and the adoption of Semantic Web technology to improve the design, reuse, and documentation of Web APIs. However, it is not clear which metadata should be in the static documentation of the REST API or into hypermedia controls and ultimately how to leverage this metadata on the client-side to achieve break-free evolution. We thus study these questions and clarify exactly that.

We validate our approach with a quantitative analysis in the form of a case study that mimics Jira and implements 110 API evolutions and through a qualitative analysis where we implement our approach on 5 open-source web applications that implement 20 evolutions of 10 types.

Therefore, using our approach that relies on a mix of API documentation enriched with

semantic descriptors and on hypermedia controls, we were able to design a Web user interface that adapts to 27 of the 29 types of API evolutions, at runtime, without updating its code. We were also able to make five real open-source web applications *evolvable-by-design* for the 20 evolutions they implemented.

Context of the thesis

This thesis has been conducted in collaboration with Zengularity, a company acquired by Fabernovel during the collaboration. The collaboration took the form of a CIFRE thesis [3], a program set up by the French government. During the thesis, I spent an important part of my time at Fabernovel with the role of a Web developer.

Zengularity is a company of 80 persons including 60 web developers. Their specialty is the design and implementation of web platforms and web applications. They mainly create web-oriented architectures with data referential systems, event-based architectures, big data architectures, and REST APIs, including hypermedia REST APIs. The expertise did not change while joining Fabernovel.

Thesis outline

The remainder of this document presents this thesis in detail. It is organized as follows:

— Chapter 1 presents the background of this work. It explains how the Web evolved and then reviews related work. Building on this, we highlight the main challenges addressed in this thesis.

— Chapter 2 presents an overview of the two contributions of this thesis.

— Chapter 3 presents the first contribution of this work. Based on the comparison of the similarities and differences between 36 REST API design, documentation, and implementation technologies, it proposes three comparison matrices that help REST API architects and developers choose the proper set of technologies for their needs. It demonstrates how the matrices can be used on a real-world example and present the open-source online assistant that we developed to further ease this process.

— Chapter 4 presents the second contribution of this work. It presents a novel approach to the break-free co-evolution of REST APIs and their clients, especially the Web user interfaces. In particular, it identifies what metadata should be shared by the REST API to its clients and at what moment in time to enable the design of *evolvable-by-design* clients. They are web clients able to adapt at runtime to typical REST API evolutions without breaking

3. Cifre program description: `https://www.enseignementsup-recherche.gouv.fr/cid22130/les-cifre.html`

neither requiring the update of their code. The proposed approach is evaluated through both a quantitative and qualitative study. Finally, the limitations of the proposed approach are discussed.

— Chapter 5 concludes the thesis and presents some research perspectives.

Chapter 1

Background and state of the art

Foreword

Organized in eight sections, this chapter presents the background and state-of-the-art of the thesis. The first section presents how the World Wide Web evolved from its creation to today. By summarizing the main evolutions of its technologies, usage, and architecture, it highlights the peculiarities of the current architecture of the Web where REST APIs became the backbone of communications. The following two sections summarize the basic concepts of the REST architecture and the separation of concerns in the current Web. The following sections are dedicated to the state-of-the-art related to the creation, maintenance, and evolution of systems based on Web APIs. The domains of the software contracts, the Semantic Web, the co-evolution of systems, and the design and evolution of REST APIs are studied. The last chapter synthesizes the state-of-the-art and points out the two challenges that are addressed in this thesis.

1.1 How the architecture of the Web evolved?

The World Wide Web was introduced in 1990 [30] as a set of three standards that aimed to use the Internet as a global information system. With 5 billion users in December 2020 [1] (64% of the world population), it indeed became a global information space.

In the Web, a core execution platform is the Web browser. Similarly, as the device and operating system on a smartphone, the Web browser defines what can be achieved with websites and Web applications. Therefore, the architecture of the Web has significantly been influenced by the possibilities offered by Web browsers. The other major influence on the architecture of the Web has been how people use it, which has been driven by the number of people using the Internet and the speed of their connection.

1. Source: `https://www.internetworldstats.com/`

In the last thirty years, Web browsers evolved a lot. The weight of the Web browsers illustrates that well. Mozaic, the first popular Web browser weighed 9.6 megabytes in 1993 while Chrome and Firefox weighed between 350 and 500 megabytes in 2021. Web engineers have therefore continuously adapted their systems to leverage the new features regularly brought to the Web execution platform. As a result, we can distinguish three major technological phases in the history of the Web.

The rest of this section goes through each major phase of the Web. For each phase, the evolution of the Web browser is detailed, then it is explained how it shaped the architecture of popular Web systems and finally how Web development practices were impacted.

1.1.1 The first phase: 1989 to 2004

The first phase of the Web architecture can be drawn between 1989 and 2004. At that moment in time, the Web was mostly a place where website providers proposed their own content. It could be websites presenting research laboratories, the news, entertainment, or online shops. Internet connections were slow back then. ADSL connections started to be deployed in 1999. Before 2004, most users were thus connected through the telephone line, a technology named dial-up connection [2] that averages 56 kb/s.

Evolution of the Web browser

In 1989, Tim Berners-Lee proposed to leverage a distributed hypertext system to manage the general information about accelerators and experiments at CERN [31]. A few months later, he and Robert Cailliau started to generalize this idea to use the Internet as a global information management system [30]. Between October and Christmas 1990, they specified the three main technologies of the World Wide Web (URL, HTTP, and HTML), implemented the first Web server and the first-ever Web browser.

Then, in 1993, they decided to release the WWW's technologies into the public domain. At that moment in time, the Web was solely a collection of interlinked text documents and the Web browsers could only display text or images in separate windows. In addition, they were available to only 20% of the computer users.

As a consequence, the first efforts were focused on improving the appeal of the Web, with media, animations, and style. These innovations were mostly driven by the Web browsers and the Web browsers war started rapidly. Mozaic has been the first popular Web browser. It was first released in 1993. Being easy to use, available on most operating systems, and able to embed graphics in text documents it popularized the Web. In 1994, Netscape was released followed by Opera and Internet Explorer in 1995.

2. `https://en.wikipedia.org/wiki/Dial-up_Internet_access`

Between 1993 and the beginning of 1994, lots of browsers had added their own bits to HTML; the language was becoming ill-defined [3]. HTML 2 was thus specified to accommodate everyone and try to conciliate practices. The World Wide Web Consortium (W3C) has therefore been created in late 1994 with the objective to bring open standards to the Web. The first proposition of an open standard for HTML has not been successful until representatives of the browser companies met and worked together on the standard in November 1995. Since 1996, Web browsers started to align their implementations of HTML. Later, HTML 3 was proposed but it was so different from HTML 2 that it is HTML 3.2, close to HTML 2, that has been adopted in January 1997. It added tables, applets, text flow around images, superscripts, and subscripts. HTML 4 superseded it in December 1999 with support for more multimedia options, scripting languages, style sheets, better printing facilities, and documents that are more accessible to users with disabilities [4].

Also during 1993 and early 1994, each Web browser styled documents differently. Some let the user configure the style, some not. For example, Mozaic did not. However, developers were not satisfied with this and wanted to be able to customize their pages [5]. The Cascading Style Sheets (CSS) enabled this since 1994 and became a W3C Recommendation in December 1996 [32]. Yet it had not been adopted by the Web browsers immediately. Indeed, the first commercial browser to support CSS was Microsoft's Internet Explorer 3, which was released in August 1996. The next browser to announce support for CSS was Netscape Navigator, version 4.0 released in June 1997 and the third was Opera in November 1998. The other browsers followed progressively. However, all browsers had a different implementation of the CSS support. A website looked therefore different on each Web browse. As a result, Todd Fahrner created the acid test in October 1998 to help Web browser providers test their implementation in order to improve their CSS support. Other tests suites followed. Since its first draft in 1998, CSS 2 added attribute selectors, improved positioning, media-types, and more features. These features have progressively been implemented in the Web browsers but with differences once again. While CSS1 and CSS2 allowed Web designers to customize the appearance of their site, they could not animate pages before March 2009 [6].

Therefore to create dynamic and animated pages, several technologies have been proposed and coexisted for several years.

The first have been LiveScript, created by Netscape and shipped into a beta version of Navigator in September 1995. It was renamed JavaScript four months later, in December 1995, for the final release of Netscape Navigator 2. Its name was chosen to take advantage of the trend following the Java programming language at that time. Then, Microsoft decided to implement

3. https://www.w3.org/People/Raggett/book4/ch02.html
4. https://www.w3.org/TR/html40/
5. https://www.w3.org/Style/CSS20/history.html
6. https://www.w3.org/standards/history/css-animations-1

a competitor to JavaScript, namely JScript, in Internet Explorer since 1996. Then in 1998, Netscape submitted JavaScript to ECMA International to propose it as a cross-browser standard. However, JScript stayed the de-facto standard for browser-side scripting until 2004. During this period, client-side scripting was stagnant. But it started to change in 2004 with the introduction of Mozilla Firefox, which is discussed later in the document. Therefore, before 2004, client-side scripting was mainly used to make websites more appealing. For the record, it was at the time considered as a language for non-engineers.

Another initiative to animate Web pages has been the Java applets, shipped with the first version of Java and also introduced in Netscape Navigator 2. They had to be executed on the Java Virtual Machine. They could therefore only be executed aside from the HTML document. As a result, Java applets were used to create stand-alone applications or games that could be executed inside Web browsers (and on mobile phones later) but outside the Web page. Applets were however not well suited to manipulate the DOM and had security issues [33]. In addition, mobile operating systems such as Android and iOS did not embed JVMs. So, the technology was discontinued in 2013.

Besides, the third technology proposed to animate Web pages was Macromedia Flash, later acquired by Adobe. It has been created as a vector drawing application in 1993 and evolved into an animation tool in 1995 before moving to the Web browser in November 1996. At that moment in time, it was mainly used to make websites more fun (which was the ultimate goal back then). Between 1996 and 1999, Macromedia added MovieClips, Actions (the precursor to ActionScript), Alpha transparency, and other features. After that, Flash improved to enable the design of Web applications. In 2004, ActionScript 4 added support for object-oriented programming, more advanced UI components, and more. Adobe acquired Flash in December 2005. Flash has therefore been an alternative to Java applets that were much more popular thanks to its lighter execution platform.

To sum up, during the first phase of the Web architecture, Web browsers were fighting for the first place. The challenge was to make the Web as appealing as possible. To this end, each browser had a different interpretation of the standards, which was lightweight at the time. Moreover, they extended them with their own features. And above that, many alternatives were proposed to style and animate the Web pages, from Web browser propositions such as user-defined configuration to new technologies such as JavaScript, JScript, Flash, and so on. Because the Web was mainly driven by vendor-specific implementations, websites were optimized for only one Web browser and complex to make good looking. Community efforts thus started to bring open standards to the Web, to bridge the gap between Web browsers. At the time, the features added to the Web browsers were mainly focusing on making it a more fun place to navigate with more customization options such as tables, site-specific style, animations, advanced media, and games.

Architecture of popular Web systems

Between 1993 and 2004, the number of people connected to the Internet increased rapidly, with an average of 2 percent more of the world's population each year (so 20% in 10 years). Most users were connected using the telephone line, resulting in Internet speed of on average 56 kb/s [7]. Loading Web pages or sending e-mails was therefore slow and music or video streaming was not viable.

Keeping the focus before 2004, we observe that the Web was mainly used to consume content and buy items online. While usages are more diverse now, this statement is still true, and many websites created in the 1990s remain popular nowadays.

Indeed, by taking a look at how the usage of the Web evolved, we observe that between 1991 and 1992 it had been used to present research organizations, who were the first users of the Internet. In 1993, many new kinds of websites appeared: search engines (Aliweb), news (Bloomberg), commercial (global network navigator), and religious movement (chabad.org) websites along with map services. In 1994, Yahoo was created along with the first government and association websites (e.g. Amnesty International). More commercial websites appeared (Pizza Hut, airline sites, etc.) and *Web Crawler*, the first full-text search engine was created. Then, 1995 saw the creation of Amazon, eBay, and Alta Vista (a famous search engine). Hotmail followed in 1996 and Google in 1998. The same year, the first website for designers was created, K10k.

From an architectural point of view, the World Wide Web has originally been designed to be a global information system consumed by humans, where authors publish documents that will then be consumed by other persons. Fig 1.1 is the original WWW architecture diagram from 1990 that demonstrates this. Hence, at that moment in time, users could solely access documents and databases over the Web [30].

Yet, as shown before, usages evolved rapidly and since 1995, commercial websites such as Amazon or eBay needed to personalize their website for each user, for example, to manage their shopping cart. To achieve this, the website placed a cookie on the user's browser and when he requested a page, the Web server used the content of the cookie to generate a personalized page. Therefore, the WWW rapidly moved from static HTML documents to dynamic pages generated on the fly.

In addition to this, some companies wanted to create services that could be used by multiple websites. So, Salesforce launched the first Web API in February 2000 [8], a sales force automation as a "Internet as a service", XML API. Nine months later, eBay launched its Developers Program and offered a Web API to enable partners to "create applications based on eBay technology". Then in 2002, Amazon launched Amazon.com Web Services allowing developers to incorporate Amazon.com content and features into their own websites.

7. Source: `https://www.plus.net/home-broadband/content/history-of-the-Internet/`
8. Source: `https://apievangelist.com/2012/12/20/history-of-apis/`

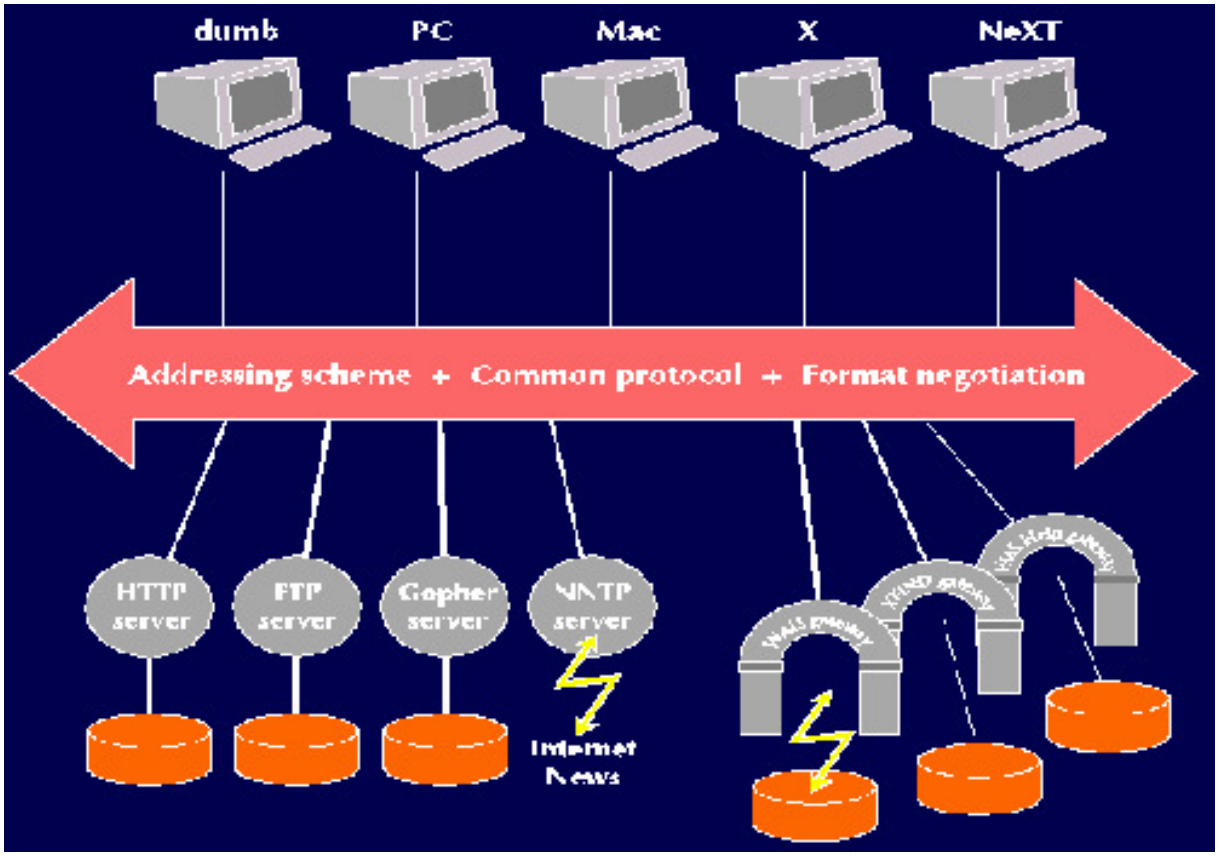


Figure 1.1 – Original WWW architecture diagram from 1990. The pink arrow shows the common standards: URL, and HTTP, with format negotiation of the data type. Source: w3.org

As a result, between 1991 and 2004, websites were either static HTML documents or dynamically server-side generated and personalized HTML documents. In addition, some Web developers leveraged available Web APIs to build their own websites. Figure 1.2 represents the architecture of the most common Web applications of the World Wide Web before 2004. Indeed, it represents the type of applications that have been present since the beginning of the Web. In addition, it depicts the changes in the architecture of server-side generated Web applications that sometimes use external Web APIs during the generation of the HTML page. It therefore represents that websites communicate with each other.

Web development practices

The World Wide Web rapidly gained the attention of many developers, and we thus observe that it has been used by a lot of developers before its foundations could be solidified. As the standards were still to create, Web browser developers implemented and shipped new features that they then tried to standardize. It has thus been long to have similar implementations of

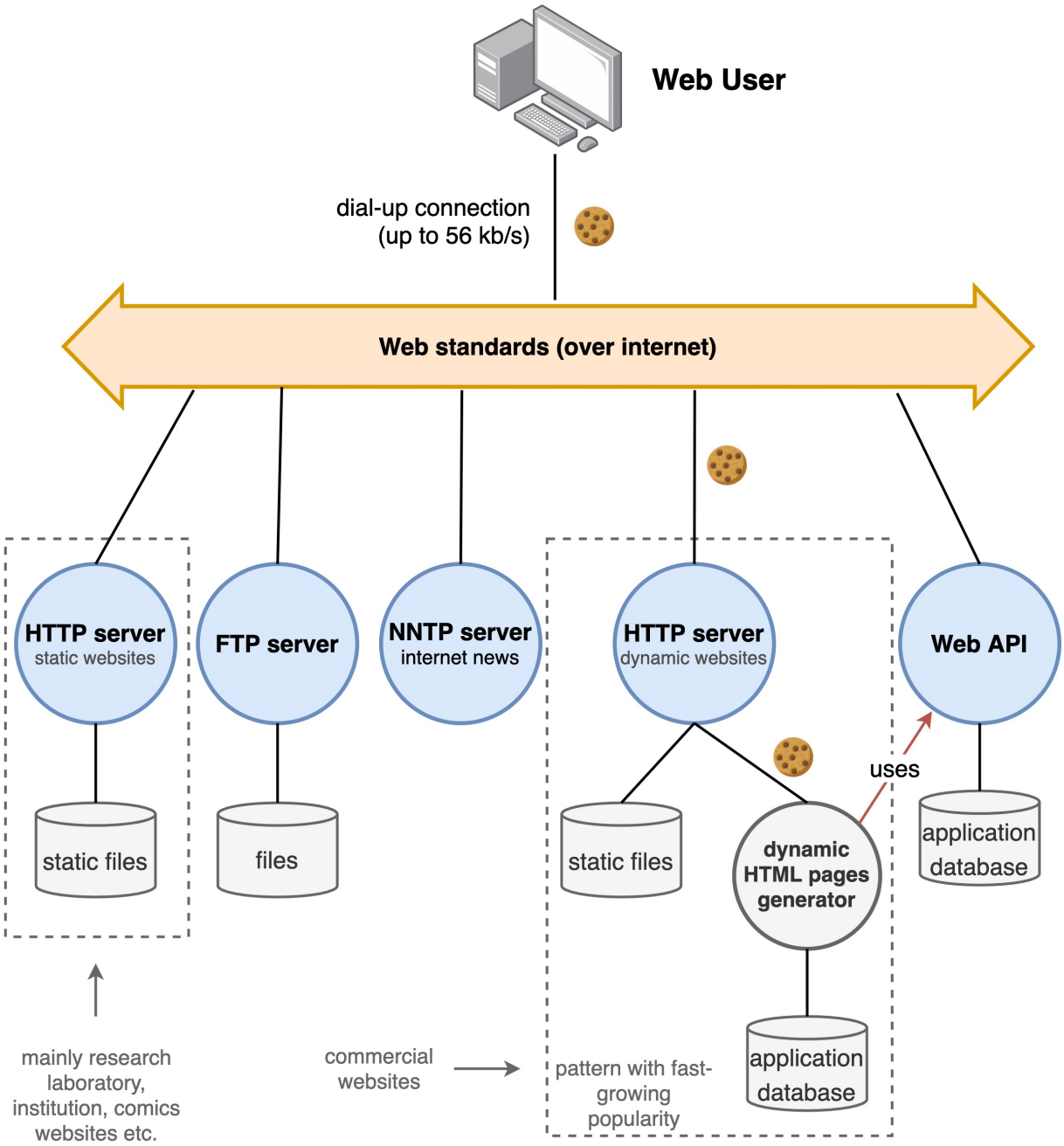


Figure 1.2 – Architecture diagram of the WWW before 2004

HTML, CSS, and JavaScript in the browsers, and the release of the acid test in 1998 demonstrates this.

Web developers could therefore not rely on solid foundations to build the front-end of their website. As an illustration, they preferred to use HTML tables instead of CSS to manage the

layout of their webpages because it was more consistent across browsers [9]. Most websites were therefore accompanied by banners indicating for which browser they were optimized.

And while the Web browsers were not stable at that moment in time, developers could find a few technologies that they could rely on. They rapidly aligned their practices to design websites. Thanks to this, they could easily share and find best practices to overcome the limitations of the technologies or Web browsers.

Consequently, the most common way to build websites before 2004 was the following. Websites were mostly managed by small teams or even only one developer [34]. They had to be made small enough to be transferred in a reasonable amount of time over dial-up connections. From a technological point of view, HTML was used to structure and design the layout of Web pages. CSS was used with parsimony to color elements, use background images, manage the font and round elements. Flash rapidly became more popular than Java applets and was used to animate websites with interactive features or videos. In order to personalize the website for the user, server-side technologies appeared quickly. Indeed, since June 1995, PHP enabled the server-side generation of personalized Web pages. Since then, PHP has been one of the most popular languages to build server-side rendered websites. Later, in December 1996, Microsoft introduced ASP (Active Server Pages) offering similar possibilities. These technologies therefore eased the creation of commercial websites such as eBay. About practices, most website projects were relatively small at the time. Small teams of developers often used only one PHP file per page of the website. They could therefore easily locate changes to do and deploy updates of the website. On the other hand, bigger teams continued to use the best engineering practices of the time. Yet, most IT projects were not Web systems and not moving to the Web either [34]. In the end, websites started to be split into repositories with many files when the development teams started to grow. This period coincides with the advent of the Web 2.0.

1.1.2 The second phase: 2004 to 2010 (the collaborative Web 2.0)

The second phase of the Web started around 2004. The Web became collaborative and has been named the Web 2.0. Some websites are particularly representative of this period such as MySpace, Facebook, YouTube, or Reddit.

This shift to a collaborative Web has mainly been triggered by two technologies: ADSL and Ajax. On the one hand, Internet connections have been made significantly faster thanks to ADSL. And in addition to this, Ajax made collaborative websites a lot faster by introducing asynchronous processing in the Web browser.

A third phase of the Web can be observed and started in 2010. Therefore this section details the evolution of the Web browser, the architecture of popular Web systems, and Web

9. A testimony of how a developer built websites in the early years of the web: `https://nickjanetakis.com/blog/was-web-development-better-back-in-the-early-2000s`

development practices between 2004 and 2010.

Evolution of the Web browser

The technologies that shaped the Web 2.0 were released before 2004. They then took time to be implemented in the major Web browsers and to be adopted by the developers. This subsection details the technologies that were added to the Web browsers and that shaped the Web 2.0.

First of all, the technology that has had the most impact on the evolution of the Web 2.0 is AJAX (Asynchronous JavaScript And XML) [35]. As its name stands for, it enables Web browsers to send and receive data from a server asynchronously, i.e. without blocking the Web page. This quite simple idea transformed the web. Indeed, instead of building Web pages where most user interactions force to reload the entire page, Web developers could then create Web applications that change content dynamically without the need to reload the entire page. Websites therefore need to transfer a lot fewer data over the network and hence become significantly faster. Indeed, the quantity of data to transfer had a significant impact on the overall load time with the slow Internet connections of the time. In addition to this, the website becomes more responsive as users can continue to use it while the data loads in the background.

AJAX is actually a set of Web development techniques that leverages many technologies. There is one browser-side API that has been created to make AJAX possible: the XMLHttpRequest (XHR) object. This API enables the Web browser to make asynchronous requests to Web services. It has been created by Microsoft and first shipped in Internet Explorer 5 in 1999 as an ActiveX control [10]. In December 2000, Mozilla shipped an implementation in their Gecko layout engine [11] but it has not been completely functional before June 2002. In the following years, XMLHttpRequest became a de-facto standard in most major Web browsers. It had been implemented in Netscape 7.1 in 2003, Firefox 1 and Safari 1.2 in 2004, and then in Opera 8 in 2005. Surprisingly, standardization has been longer since the W3C released a first working draft in 2006 [12].

In parallel, HTML has not changed much between HTML 4 released in 1999 and HTML 5 released in 2008. It has thus not had a significant impact on the Web 2.0. During this period, most efforts focused on XHTML which was supposed to replace HTML. Yet, Web browser developers were not satisfied with how the W3C worked on the standard. They also judged it too complex to implement. As a result, they created the Web Hypertext Application Technology Working Group (WHATWG) and started to work on what became HTML 5.

CSS is the second standard that has not changed much during this period. While CSS 2 was a single recommendation published in 1998, CSS 3 is a living standard that has been split into

10. https://web.archive.org/web/20090130092236/http://www.alexhopmann.com/xmlhttp.htm
11. https://www-archive.mozilla.org/releases/
12. https://www.w3.org/TR/XMLHttpRequest/

multiple modules with dedicated working groups. The first major changes in CSS appear in 2008 and they did not have a significant influence on the Web 2.0 either.

Besides, Web browsers improved their support of the standards and scripting languages to make Web browsers faster [13] [14] [15]. At the time, Web browser providers focused on improving the performance of their software, its user experience and tried to pass the Acid test. Yet, JavaScript engines were too slow to enable the design of complex Web applications.

In the end, it is noticeable that the Web 2.0 has not seen many technological changes in the Web browser, the only significant being AJAX. Indeed, the Web 2.0 is at first a change in how the Web is used, which has been driven by the growing number of users of the web, the constantly increasing speed of JavaScript engines, the deployment of the ADSL, and the availability of AJAX. It is therefore interesting to dive deeper into how popular Web systems have been designed to offer new collaborative features. This is what the next subsection details.

Popular websites representative of the phase

The Web 2.0 is foremost a change in the usage of the web, driven by the rapid and significant increase in Internet speed and a trend towards building collaborative websites. Therefore, some websites are particularly representative of the Web 2.0.

Among them are the wikis and we observe that Wikipedia has been introduced early, in January 2001. Similarly, forums are collaborative websites, the most famous today being Reddit that has been launched in June 2005.

Yet, the categories of websites that most people would cite as being representative of the Web 2.0 are social media. The first social media is Born which was created in 1997 [16]. Friendster followed in 2002 but was not able to handle an important number of users. Finally, the first social media to be used by millions of people, and that is often cited as the first social media is MySpace. It has been founded in August 2003 and it rapidly attracted many famous people, which is one of the main reasons explaining its popularity. For the record, in France, this is Skyblog that has been the most popular in this category in the early 2000s. It has even been released before MySpace, in December 2002. Not long after, in February 2004, Facebook has been launched to very few users. It has first been open to a few universities in the United States and it is finally on September 26, 2006, that it has been opened to everyone over 13 years old. In March of the same year, Twitter was also founded.

Another category of website that represents the Web 2.0 and that is still very popular in 2021 is media sharing. These websites are most of the time dedicated to a single type of media. For example, Flickr is dedicated to photo sharing and is a typical representative of the Web

13. https://en.wikipedia.org/wiki/History_of_the_Opera_web_browser#Version_6
14. https://en.wikipedia.org/wiki/Safari_(web_browser)
15. https://en.wikipedia.org/wiki/Internet_Explorer
16. https://blog.hootsuite.com/history-social-media/

2.0, which means that the content of the website is created and shared primarily by the users themselves. It was created on February 10th, 2004. On February 14th 2005, this is YouTube that has been launched by three former employees of PayPal.

In the end, we notice that many collaborative websites were created between 2004 and 2006. This is also notable that most of the websites that are popular in 2021 were created between 1995 and 2005.

With the advent of personal blogs and the increased presence of news websites, people also started to consume information from many sources. They therefore needed a central place where they could get the latest articles from a selection of websites. For this purpose, news aggregators leveraging the RSS technologies have been created. According to Wikipedia, RSS "is a Web feed that allows users and applications to access updates to websites in a standardized, computer-readable format". RSS has rapidly been implemented in many news websites and blog platforms since its creation in 1999. It is therefore a technology that had a significant impact on the websites of the Web 2.0.

To make a link with the evolving technologies, there is one website, or Web application in this case, that is worth mentioning: Gmail. It has been the first popular Web application that really showed off what was possible with client-side JavaScript since 2004 [17]. It was therefore a great example showcasing what is possible to achieve with AJAX.

To finish, it is interesting to note that Salesforce launched the first Web API in 2000. More and more companies followed the trend. An example that is often cited is Amazon, who launched its first Web API in July 2002. Amazon then heavily relied on Web APIs to boost the adoption of Amazon Web Services, which has been effective. As another example, Flickr launched its REST API only six months after the website in 2004. In 2006, Facebook and Twitter launched their Web API [18]. Google Maps also launched its Web API in 2006 after discovering that developers hacked the JavaScript interface and developed applications such as housingmaps.com and chicagocime.org [19].

Architecture of popular Web systems

We have previously seen that with the Web 2.0, websites became more collaborative and were accessed by more people at a time. This was due to the increasing number of users of the Web and the faster Internet connections. As a result, web designers have had to create websites that can be served to a larger number of users simultaneously, and as new websites have emerged, it has become increasingly important to design websites that load quickly to retain users. These are the major constraints that developers faced to design the architecture of their websites between

17. `https://dri.es/a-history-of-javascript-across-the-stack`
18. `https://apievangelist.com/2012/12/20/history-of-apis/`
19. `https://web.archive.org/web/20130411055812/http://apievangelist.com/history/google-maps.php`

2004 and 2010.

As Internet connections became faster, computers and by extension servers too. Therefore, the increase in users could easily be handled by new, more powerful Web servers. On the other hand, the advent of AJAX slightly changed the architecture of Web applications. Indeed, on the server-side, the application server had to handle the AJAX requests in addition to the page requests. At Internet scale, the biggest change is finally about the flows but not architecture: requests to Web services increased during this period. Thanks to AJAX, they could be done from the browser and not only the server.

Similarly, as more and more websites were created, the percentage of HTTP servers on the Internet increased and the percentage of FTP servers decreased.

As a result, Figure 1.2 stays relatively relevant to represent the architecture of systems in the Web 2.0. Yet, Figure 1.3 is an update of Figure 1.2 and represents the slight changes. In addition, Figure 1.4 shows the difference in the interactions between the Web user interface and the application servers. Indeed, before the Web 2.0, application servers served HTML, CSS, JS, and media files only while in the Web 2.0 they also include a Web API serving responses formatted as XML or JSON documents.

About Web APIs, there is one architectural change that had a major impact on the practices and that is interesting to discuss. In the early 2000s, Web APIs were mostly implemented as SOAP services, but their popularity decreased and REST APIs started to become the de-facto standard for building Web APIs around 2006 when major players such as Google and Yahoo deprecated their SOAP services in favor of REST APIs. There were many reasons that lead to the demise of SOAP-based services. The most important may be that the architecture is based on Remote Procedure Call (RPC) that has been known to be flawed [36]. Moreover, the protocol uses HTTP as a transport protocol instead of an application protocol. SOAP services send POST requests to get data from a server. They therefore break intermediaries that serve as proxies or caches which typically operate based on the standard semantics associated with the HTTP verbs. Other factors lead to interoperability problems, such as the abstract data types used by WSDL. Overall, the XML Schema language has a number of type system constructs that complicate implementation and interoperability.

Last, one research initiative of this period is interesting to discuss as it has been an important inspiration for this work. A few years after standardizing the foundations of the web, Berners-Lee continued to work on the WWW. He wrote that “A new form of Web content that is meaningful to computers will unleash a revolution of new possibilities" [37]. Then, since 2000, he and many other researchers defined a set of standards to enable this, a Semantic Web. They thus envisioned a different architecture for the Web that is discussed in section 1.5. This vision has not reached a wide adoption, so semantic Web services [38] are not represented in the figures of this section.

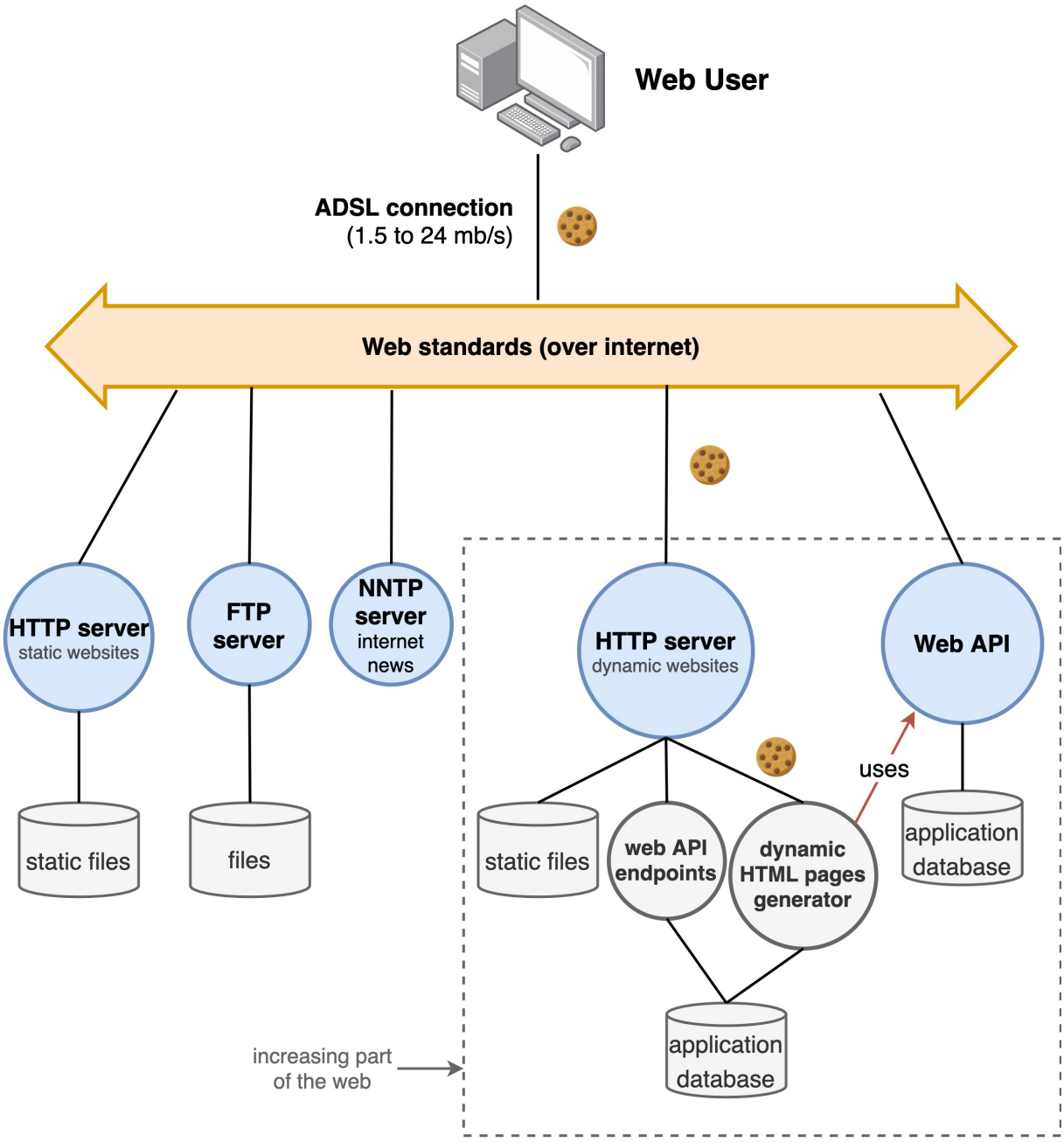


Figure 1.3 – Architecture diagram of the WWW between 2004 and 2010

Web development practices

Previously, we saw that the major browser-side improvements of this period have been the increasing speed of the scripting engines and the support of the XMLHttpRequest API to enable AJAX.

On the client side, developers have started to build Web applications that fetch data asyn-

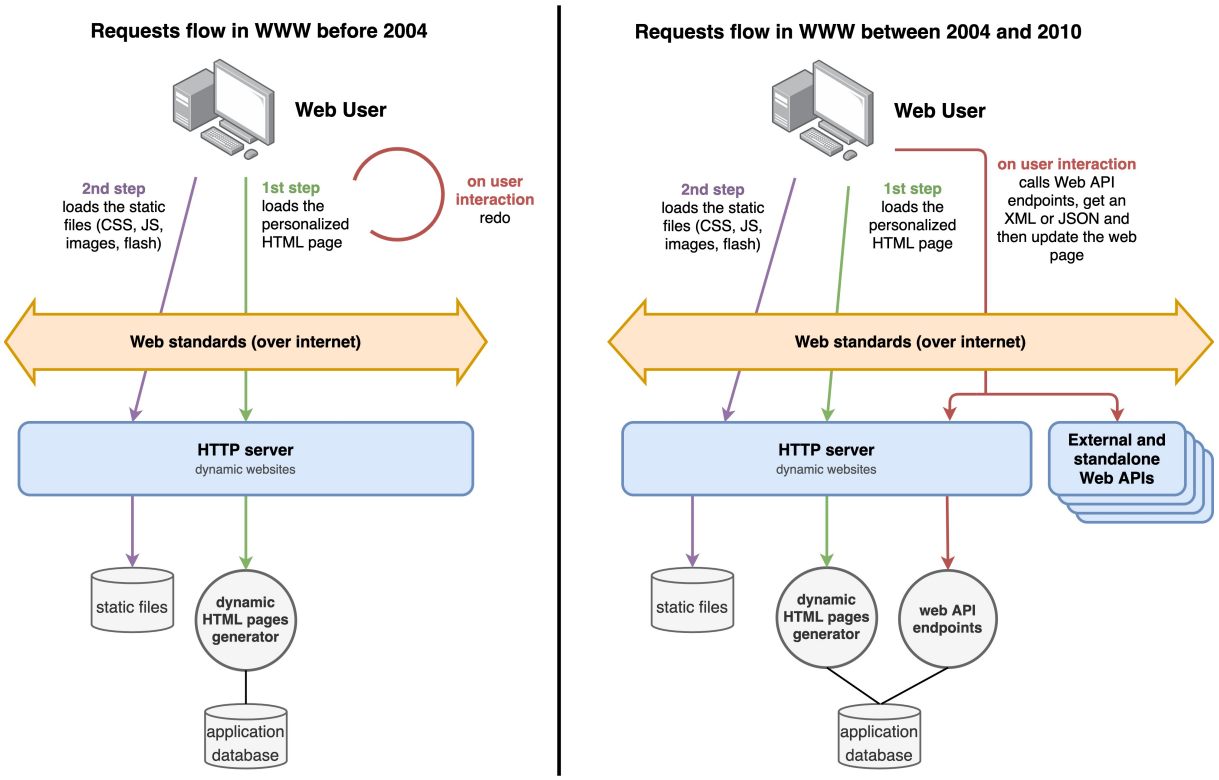


Figure 1.4 – Diagram opposing the client server interactions within the Web before 2004 and between 2004-2010

chronously and update the DOM of the page in the background. Yet, the DOM APIs are verbose and implementations vary between Web browsers. The jQuery [20] JavaScript library has been created in 2006 to ease the development of Web applications. As it states on its home page, “it makes things like HTML document traversal and manipulation, event handling, animation, and Ajax much simpler with an easy-to-use API that works across a multitude of browsers". jQuery has been a massive success and is still the most popular JavaScript library. In 2017 it was used on 69.2% of the top 1 million websites (according to Libscore) [21].

On the server-side, developers have started to add Web APIs to their application servers to make their Web applications faster. In addition, because AJAX enabled developers to build Web applications without developing complex application servers but instead by composing existing Web APIs directly from the client-side, more and more standalone Web APIs appeared. For example, headless Content Management Systems (CMS) enable developers to build complex blogs without implementing a server.

As more and more developers built collaborative websites and Web APIs, the technologies

20. `https://jquery.com/`
21. `https://web.archive.org/web/20170219042532if_/https://libscore.com/#jquery`

and practices evolved fast. So, many frameworks appeared since 2004 to ease the development of such software. Among the most famous, Spring 1.0 has been released in March 2004, Ruby on Rails in February 2005, Django in July 2005, Symfony in October 2005 and Play! Framework in May 2008. For the record, Play is a framework that we developed at Zengularity, now owned by Fabernovel. These frameworks all emphasized the Model View Controller (MVC) pattern and can therefore be called MVC frameworks.

With the help of these MVC frameworks, developers could build Web applications much faster. They also tended to join communities of developers that use the same framework. As a result, communities of developers using a given framework shared the same practices, but not all communities share the exact same practices.

1.1.3 The third phase: 2010 to today (mobile applications and the JAMstack)

In its third and current phase, the Web is much more distributed and loosely coupled than before. Nowadays, many websites became Web applications and are very similar to mobile applications for the web: they are standalone applications that use Web APIs to access the content. Thanks to this architecture, websites load faster, the Web is loosely coupled and it is possible to collaborate in real-time with several people on the edition of the same page.

This third manner of building Web systems has been driven by the arrival of smartphones. To propose an application to a wide range of users, developers had to offer web, Android, and iOS applications. Then to interconnect these applications they have to implement a single server and three user interfaces. As a result, new technologies have been released to ease the development of applications following this architecture. In addition, new technologies appeared in the Web browser and to build and deploy Web APIs for the cloud. The rest of this section details them.

Evolution of the Web browser

In 2008, Google introduced the V8 JavaScript engine in Chrome, which was ten times faster than Internet Explorer thanks to Just-In-Time (JIT) compilation [22]. Two years later, they doubled the speed of the engine with a new optimizing JIT compiler. This evolution has enabled the development of complex Web applications able to be as smooth as native applications.

The Web standards also evolved a lot and had an important influence on this third phase. In addition, they have rapidly been supported by the most popular Web browsers.

Among them, HTML5 has been well received when it was released in January 2008. Many new features were added such as native support multimedia such as the video, support for MathML, new tags with more semantic and more APIs such as drag and drop, local storage, and Web messaging. HTML5 became a living standard and is improved regularly.

22. `https://dri.es/a-history-of-javascript-across-the-stack`

The CSS standard has also been improved with many new features between 2008 and 2011. The working groups of CSS3 added responsive design, 2D and 3D transformations, native animations, filters, flexbox, and more. The latter is a Web layout model that is so more powerful than previous models that it became the current de-facto standard. It allows responsive elements within a container to be automatically arranged depending upon screen size.

On the other hand, many new standards have been created since 2010. They have made the Web browser suitable for many more applications. Among them are Web Workers. Since 2010, they offer a sort of multi-threading for Web applications to run heavy computations in the background [23]. In 2011, WebSockets have enabled real-time two-way communication between the browser and server. Another worth mentioning is the concept of Progressive Web Applications, which appeared in 2015, enabling Web applications to use a simulated and secured file system and to be cached into the Web browser to load immediately. Also, Web Assembly, a 2019 W3C recommendation, enables Web browsers to run C/C++/C# and Rust code with near-native performance. It can for example be used to run complex image analysis from the Web browser [24].

During this period, some technologies of the Web browser declined. The most famous being Flash that loose interest after the release of HTML5 and CSS3 offering similar features. *Thoughts on Flash*, the open letter written by Steve Jobs [25] to Adobe criticizing the closed nature of the Flash platform and the inherent security problems explained why Flash was not supported on iOS. This had a decisive impact on Flash. Then, in 2011, Adobe ended support for Flash on Android. In July 2017, Adobe deprecated Flash and announced its end of life at the end of 2020.

Popular websites representative of the period

The Web applications that represent this period the better are the real-time collaboration tools and applications that are available on the Web and mobile with seamless synchronization.

Google Docs [26] is the first Web application that offered a very smooth real-time collaboration between multiple users on a single document, and even on a single sentence, with almost no merging errors [27]. Released in 2010, the application was revolutionary at the time.

Leveraging the canvas API, some companies have been able to offer collaborative sketching, drawing, and prototyping tools. For example, Figma [28] enable designers to collaborate on user interfaces. Miro [29], another example, offers a collaborative tool illustrated on Figure 1.5.

On the other hand, Trello, Airbnb, and Facebook are good examples of applications that offer Web and mobile interfaces with real-time synchronization.

23. `https://www.html5rocks.com/en/tutorials/workers/basics/`
24. `https://www.youtube.com/watch?v=drRaRRZ5AJk`
25. `https://newslang.ch/wordpress/wp-content/uploads/2020/06/Thoughts-on-Flash.pdf`
26. `https://docs.google.com`
27. `https://drive.googleblog.com/2010/09/whats-different-about-new-google-docs_21.html`
28. `https://www.figma.com`
29. `https://miro.com/`

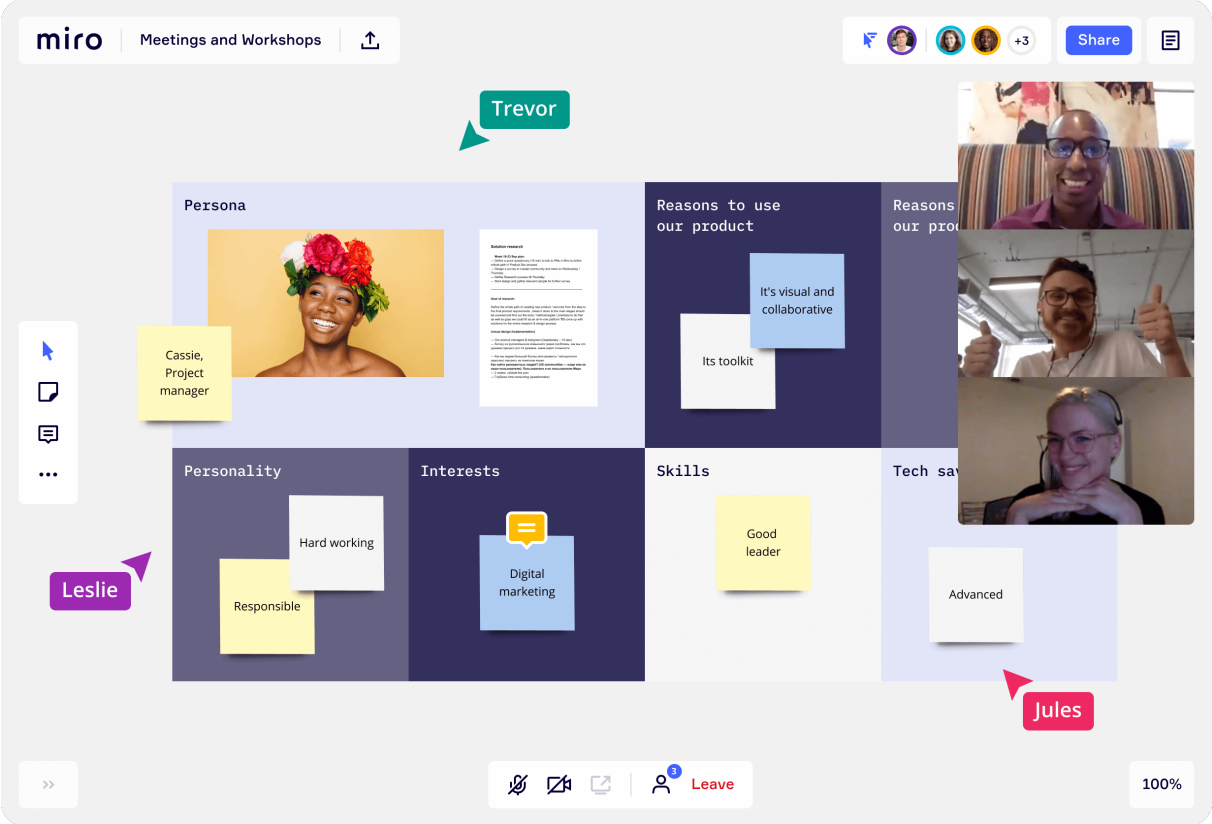


Figure 1.5 – Example of the real-time collaboration of multiple users on a single document with the Miro Web application

Architecture of popular Web systems

The architecture of popular Web systems evolved under the influence of three factors: the arrival of mobile applications, the increased performance of JavaScript engines, and the evolving cloud computing services.

With the advent of mobile applications since 2008, developers wanted to make their services also available on mobile. In the beginning, they started by adding mobile consumers to Figure 1.3. With mobile applications, they could not leverage the *dynamic HTML pages generator* of their application server. Instead, the mobile application contains the user interface and logic of the application but not the data. It must therefore access the Web API used by the website. As a result, application providers had to open their API to mobile consumers. In the process, they often had to create new API endpoints to expose the data that was previously consumed by the *dynamic HTML pages generator*.

Unfortunately, this architecture blurs the separation of concerns as one application server is now responsible for the Web user interface and for mobile-specific API endpoints. At the same time, it became possible to build more powerful Web applications similar to mobile applications,

namely Single-Page-Applications [39]. These Web applications are standalone and packaged as a set of static files. Dynamic HTML pages generators are not necessary anymore to serve Web applications. Instead, to offer multi-platform applications, developers started to build the Web API, the Web application, and the mobile applications as separate artifacts. Mobile applications are hosted on OS-specific application stores, Web applications on Content Delivery Networks (CDN), and the Web APIs as HTTP servers.

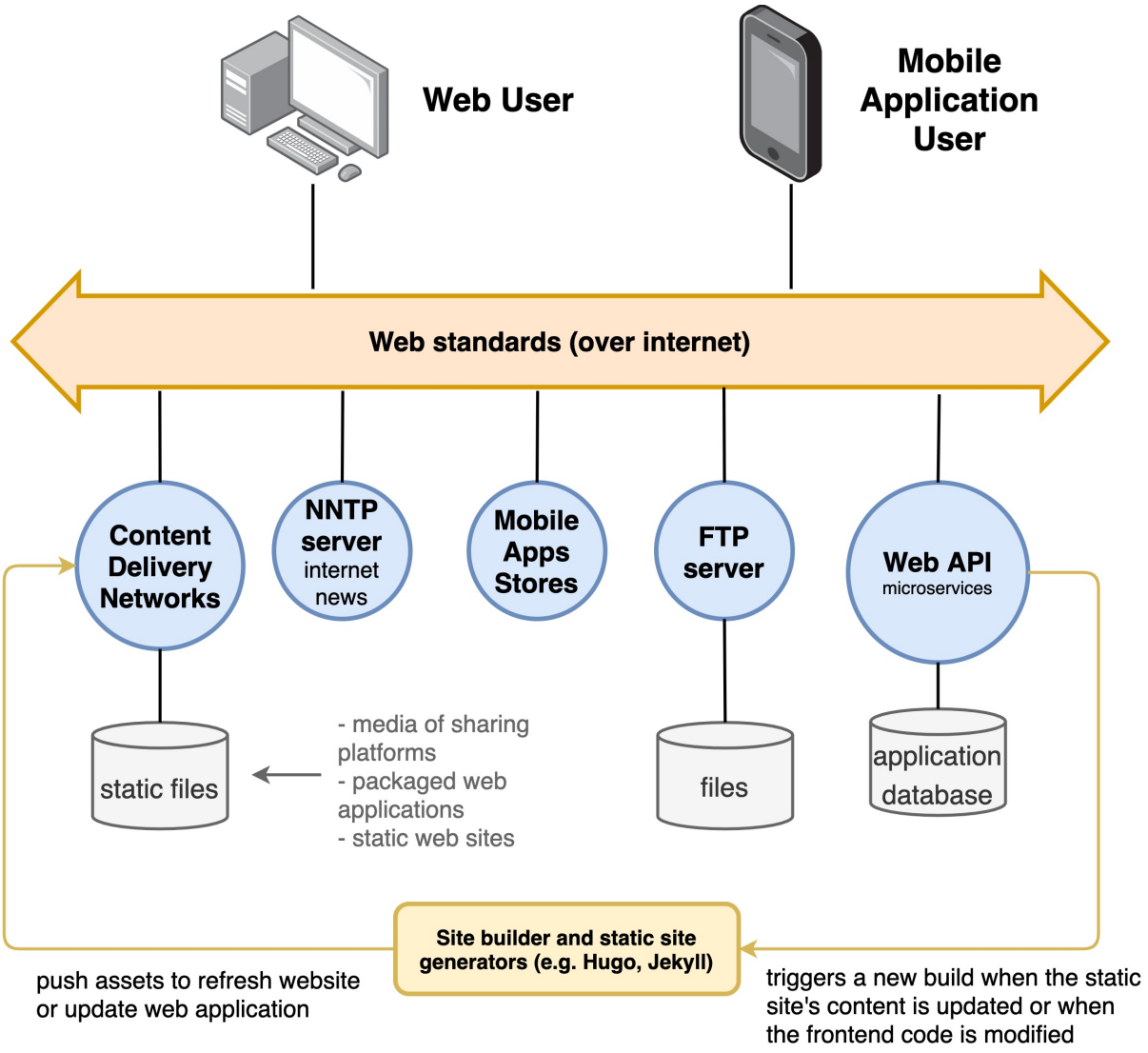


Figure 1.6 – Architecture diagram of the WWW after 2010

This new architecture is named the JAMstack [30] and is represented on Figure 1.6. It stands for JavaScript, API, Markup stack. Markup corresponds to the entire frontend being prebuilt into

30. JAMstack officiel website: `https://jamstack.org/`

highly optimized static pages and assets during a build process. This process of pre-rendering results in sites that can be served directly from a CDN, allowing very fast scaling. Here, JavaScript makes the markup dynamic and personalized, by handling user inputs and communicating with Web APIs. Last, API refers to the Web APIs used to get and manipulate data and authentication. These APIs can either be from the organization of the application provider, or outside of it, on the Internet.

To illustrate how the architecture has changed from the Web 2.0, Figure 1.7 opposes the client-server interactions of these two periods. For example, nowadays, when a user goes to a blog, the already rendered HTML pages with up-to-date content will be downloaded from a CDN that physically places the content near to the user. This is very similar to the original architecture of the web. Yet, the pre-rendered website will not contain, for example, comments or the number of likes on Facebook. For this purpose, the JavaScript code will fetch a comment management REST API and the Facebook API, and then add the data to the page. As a result, the most important part of the blog, the post, will load very fast, and the additional content will load progressively. Noticeably, this kind of website makes heavy use of pre-rendered HTML pages and a lightweight use of Web APIs to make the website participatory. To update the website, authors will use another application that will trigger a new build of the website when they will validate the updates. This site generator will then push the updated website to the CDN. Thanks to DevOps automation, some website managers are able to achieve this hundreds of times a day.

As another example, highly collaborative websites such as Google Docs [31], Trello [32] or Miro [33] will first load a very light markup, basically a content-less Web application with the logic and style only. Indeed, the content of this kind of website depends on the connected user. Then during the collaboration, as illustrated in Figure 1.5, users will see what the other connected users are doing. As a result, these applications manipulate fast-changing data and should be able to download and process them in real-time. To achieve this, they highly rely on JavaScript code and Web APIs to constantly get up-to-date data, update the page shown to the user and share the user's modifications in real-time.

As a result of this new architecture, websites are built only once instead of being built each time a user requests a Web page. They are also much more optimized: they load faster, consume fewer network resources, and offer new possibilities such as real-time collaboration. In addition to this, Progressive Web Applications enabled Web applications to be cached on desktop and mobile Web browsers to be used offline [40]. Web applications can thus load immediately, similarly to mobile applications.

The many technological and architectural changes of this period therefore influenced the

31. `https://docs.google.com/`
32. `https://trello.com/`
33. `https://www.miro.com`

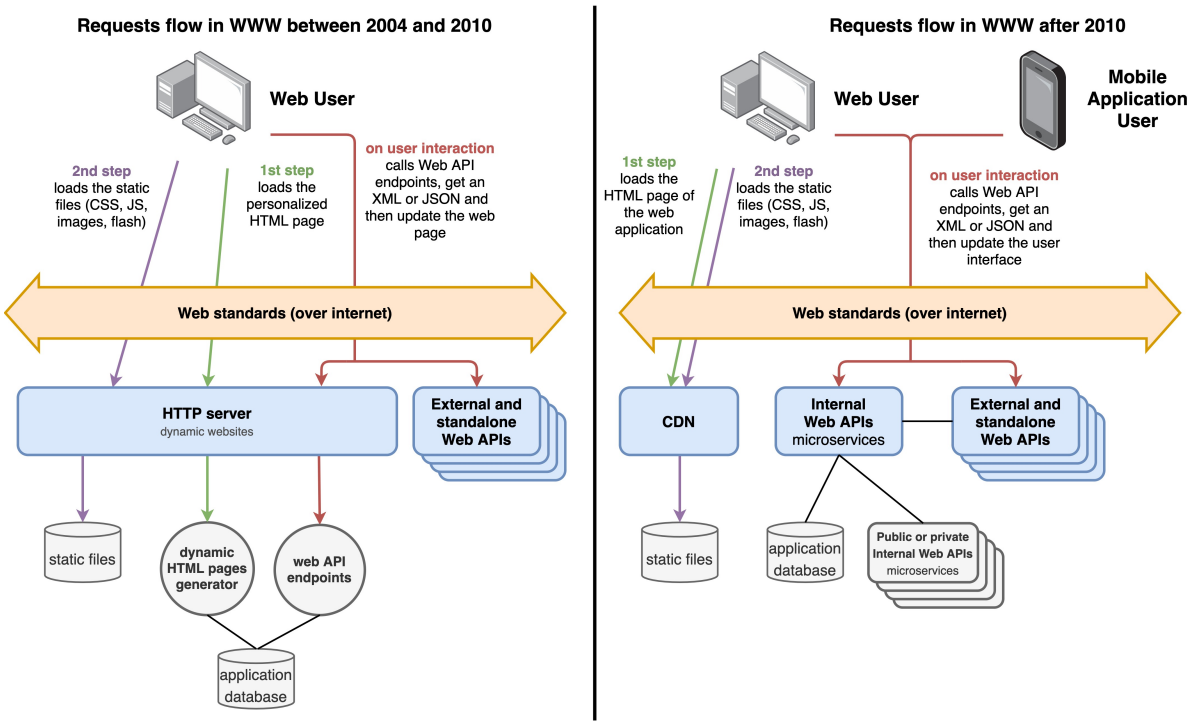


Figure 1.7 – Diagram opposing the client server interactions within the Web and between 2004-2010 and after 2010

practices a lot. Since then, more and more developers were involved in the development of Web systems. The next subsection details how the practices evolved.

Web development practices

With more and more people on the web, the amount of websites and Web applications has increased, consequently increasing the competition between them. In addition, mobile applications became a new kind of system on the web. Web systems therefore had to be improved on the following aspects:

— applications had to become highly available, i.e. available 24/7 to any number of users as any downtime could cost millions to some companies

— application data had to be open to mobile applications to offer the same experience on all devices

— Web user interfaces had to load faster, be smoother, offer advanced history management, be available in many languages, accessible to people with disabilities, and compatible with many browser versions as the Web was used by almost 2 billion users in 2010 (27% of the world population) and more than 5 billion in 2021 (65% of the world population) [34].

34. Source: `https://www.internetworldstats.com/emarketing.htm`

The developers' practices therefore evolved to address these new constraints.

First, to create Web user interfaces meeting the above constraints, developers leverage the high increase in performance of the JavaScript engines. As a result, they started to build single-page applications. Once packaged, they are composed of a single HTML document that loads JS scripts that contain all the markup and continuously update the DOM (the content of the page) upon user and Web API interaction. To do so, they rely on JavaScript and CSS frameworks. The first JS framework to reach a wide adoption is AngularJS which eases the development of single-page applications relying on the component-based approach. It was released in October 2010. AngularJS launched the front-end frameworks movement. The two most popular nowadays are React and VueJS, which have respectively been created in May 2013 and February 2014 [41]. About CSS, the first CSS framework to become very popular is Bootstrap. It proposed already implemented, responsive, and customizable UI components that could be reused to implement high-quality user interfaces very quickly. Bootstrap has first been released in August 2011 by Twitter. As with AngularJS, many alternatives were proposed quickly. These frameworks completely changed practices for building front-end applications. Instead of HTML pages enriched with scripts, developers are building scripts generating HTML content. They also offer different paradigms and thus the practices are diverging over time. For example, in recent years, there have even been communities emphasizing that the CSS should be written within the JavaScript code [35].

Besides, to create more advanced user interfaces and open the data to the mobile applications, with the architectural patterns of the Web 2.0, development teams and systems grew. Maintaining and evolving the software became increasingly complex and time-consuming. One architectural pattern therefore gained traction and became the de-facto standard: micro-services [42], inspired by the service-oriented architecture [43, 44].

With micro-services, developers split their systems into smaller software components that can generally be developed by teams of less than ten developers. Each microservice is responsible for one dedicated business purpose such as authentication or inventory management. They are easier to maintain than monolithic applications because the code is smaller and concerns are better separated.

Using the micro-services architectural pattern has changed architecture practices. Developers now design architecture as depicted on the right part of Figure 1.7. Separating the user interface from the data access and management enabled to open the data to mobile applications and to interact with the data in the exact same manner from the mobile application or Web user interface.

In addition, since the introduction of Docker in March 2013, developers started to package their micro-services into lightweight containers. These containers are black-boxes from the in-

35. `https://medium.com/dailyjs/what-is-actually-css-in-js-f2f529a2757`

frastructure standpoint that can be deployed with a single command. Before, developers had to manage complex deployment scripts on virtual machines. Docker containers therefore eased the deployment of Web APIs and changed practices.

With the growing interest in containers, cloud providers such as Amazon Web Services, Google Cloud, and Microsoft Azure started to support Docker containers. In addition to offering easy deployments to robust worldwide infrastructures, the providers developed advanced proxy servers that automatically deploy or shut down new instances of a container to adapt to the varying demand in order to offer 24/7 availability, namely load-balancers and API gateways.

To maximize automation and enable the setup of efficient DevOps practices [45], cloud providers propose Web APIs for most of their services. Developers can therefore leverage them to automate deployments, database backups, notifications, and so on.

This is representative of a more general trend where application providers build very specific services that interact with other services using Web APIs to collect data, automate tasks, extend business processes, and even manage the entire operational and service aspects of a business [36]. For example, Datadog [37] builds a software dedicated to cloud monitoring that is technology-agnostic. It therefore leverages REST APIs to collect the user's data from cloud providers. **As a result, REST APIs became the backbone of data and service exchange within the web, i.e. between back-end services, Web user interfaces, mobile applications, and even voice-user interfaces.**

A new term has even been coined to refer to the set of business models and practices designed around the use of APIs in today's digital economy: the API economy [38]. That demonstrates that Web APIs have gone beyond the boundaries of computer science to reach marketing. For example, Stripe bases its marketing on proposing a Web API for payment.

1.2 Basic concepts of the REST architecture

This section aims at giving the reader the concepts of the REST architecture. They are necessary to get a good understanding of the following sections and of the challenges that this thesis tackles.

The REpresentational State Transfer (REST) architecture has been introduced in 2000 by Roy Thomas Fielding [28], who actively participated in the design of the HTTP 1.0 and 1.1 standards at the same time. The frontier between HTTP and REST is therefore subtle. While it is often stated that REST very well integrates with HTTP, to me, it is more likely that HTTP has been designed with the REST architectural style in mind. Nevertheless, the important point is that HTTP offers constructs to design distributed, collaborative, and hypermedia information

36. `https://blog.api.rakuten.net/evolution-of-apis/`
37. `https://www.datadoghq.com/`
38. `https://searchapparchitecture.techtarget.com/definition/API-economy`

systems. However, it does not enforce any property so it can also be used as a transport protocol, which is for example how SOAP Web services use HTTP. In addition to not enforcing any property, HTTP does not even provide guidelines on how to use it in order to design distributed and loosely coupled Web applications. The REST architectural style has therefore been created to fill this gap. As its author state, it has been designed *"to form a new architectural style that better reflects the desired properties of a modern Web architecture"*. So, it defines a set of constraints to guide application designers in creating Web applications with good performance, scalability, simplicity, modifiability, visibility, portability, and reliability. If they do follow these constraints, they can claim to offer a RESTful application.

Concretely, the six guiding constraints of the REST architectural style are the following:

1. use a **client-server architecture** to separate concerns
2. communication must be **stateless** in nature
3. a response to a request [should] be implicitly or explicitly labeled as **cache**able or non-cacheable
4. use a **uniform interface** between components
5. follow **layered system** constraints
6. (optional) leverage on the **code-on-demand** style to simplify clients by reducing the number of features required to be pre-implemented

Client-server architecture. In his doctoral dissertation, Fielding states that "Separation of concerns is the principle behind the client-server constraints. By separating the user interface concerns from the data storage concerns, we improve the portability of the user interface across multiple platforms and improve scalability by simplifying the server components. Perhaps most significant to the Web, however, is that the separation allows the components to evolve independently, thus supporting the Internet-scale requirement of multiple organizational domains.". This statement is particularly interesting. Considering how the architecture of the Web has evolved, it is clear that this property has started to be achieved in the third phase of the web. Indeed, before 2010, the user interface and data were managed by the same, MVC, server. This is the advent of mobile applications that finally triggered the shift to the effective separation of the data processing and the interfaces, which is actually the separation of concerns initially envisioned by Fielding. A word about independent evolution: we observed that the user interfaces can indeed evolve independently, but that the components managing the "data storage concerns" (the REST APIs) can not.

Stateless. "Each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. Session state is therefore kept entirely on the client. This constraint induces the properties of visibility, reliability, and scalability." [28]. This constraint has been achieved for a long time, the

cookies helping a lot. In addition, in practice, the implementation of stateless applications is easier than stateful applications. It also enables horizontal scalability (the deployment of several instances of an application's server to support more users), which is essential to support a high number of concurrent users.

Cache. To improve network efficiency, which was particularly important back in 2000 with the 56kb/s dial-up connections, "cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable.". In most cases, HTTP server implementations manage this aspect automatically, requiring minimal to no effort from developers. It works particularly well for media files, that rarely change. It is however a lot more challenging to manage the cache for the data manipulated with REST APIs due to their amount and varying change frequency, but we talk about fine-grained optimization here, which is rarely necessary.

Layered System. "The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot "see" beyond the immediate layer with which they are interacting. By restricting knowledge of the system to a single layer, we place a bound on the overall system complexity and promote substrate independence. Layers can be used to encapsulate legacy services and to protect new services from legacy clients, simplifying components by moving infrequently used functionality to a shared intermediary. Intermediaries can also be used to improve system scalability by enabling load balancing of services across multiple networks and processors.". This constraint has indeed been extensively leveraged. Load balancing and proxies are indeed an essential component to ensure scalability and security in cloud infrastructures. Also, API gateways became a popular component of the cloud. They merge monitoring, security, authentication, load balancing, administration, A/B testing, and more. Researchers also use proxies to maintain up-to-date documentation with usage examples for an evolving REST API [46]. The constraint is therefore achieved and proved to be very useful.

Code-on-demand. (optional) "REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This simplifies clients by reducing the number of features required to be pre-implemented. [...] ". For more discussion on the optional character of this property, the reader may refer to section 5.1.7 of [28]. In practice in the modern web, this constraint is rarely leveraged. The most common usages are loading scripts into websites to track activity (e.g. Google Analytics) or to embed other websites' features such as Twitter feeds of Facebook likes buttons with minimal development effort (usually copy-pasting the HTML tag importing the script).

Uniform Interface. "The central feature that distinguishes the REST architectural style from other network-based styles is its emphasis on a uniform interface between components. By applying the software engineering principle of generality to the component interface, the overall

system architecture is simplified and the visibility of interactions is improved. Implementations are decoupled from the services they provide, which encourages independent evolvability.". Concretely, he encourages us to use the HTTP verbs as follows and to stick to it. The REST APIs following this uniform interface are usually named CRUD REST APIs.

1. GET to retrieve a resource representation
2. POST to create a resource
3. PUT to replace the current resource representation with the provided payload
4. PATCH to apply partial modifications to a resource
5. DELETE to remove a resource
6. OPTIONS to get the resource's representation schema

However, this uniform interface comes with a trade-off that Fielding identified: "The trade-off, though, is that a uniform interface degrades efficiency, since information is transferred in a standardized form rather than one which is specific to an application's needs. The REST interface is designed to be efficient for large-grain hypermedia data transfer, optimizing for the common case of the Web, but resulting in an interface that is not optimal for other forms of architectural interaction.".

Nevertheless, the Uniform Interface is defined by the following four constraints:

1. *Identification of resources.* Within REST, the resource is the key abstraction of information. A resource is anything that can be named, such as today's weather in Paris. In practice, a resource is identified by a Unique Resource Identifier (URI) that the user can dereference to get its representation (e.g. *https://weather.com/today/Paris)*. This is typically what happens when one writes a URI into a Web browser, the representation of the website is downloaded and rendered.
2. *Manipulation of resources through representations.* Representations are used to capture the current state of a resource. "Other commonly used but less precise names for a representation include: document, file, and HTTP message entity, instance, or variant." A single resource may therefore be serialized into an XML representation, or JSON, or within any other format. The selection of a representation is achieved through content negotiation leveraging the *Accept* and *Content-Type* HTTP headers.
3. *Self-descriptive messages.* Each message includes enough information to describe how to process the message. For example, which parser to invoke can be specified by a media type (e.g. *application/json*).
4. *Hypermedia as the engine of application state.* While the term is not defined in Fielding's dissertation, it is commonly accepted that a REST API should include hypermedia controls in its responses so that the user can navigate the REST API without any prior knowledge

of it, similarly to how people navigate websites. Fielding confirmed this interpretation in an interview back in Dec 2014 [39].

The previous constraints make clear that the REST architectural style has been grounded on the World Wide Web foundations. On top of that, the architecture has been designed with the goal to design highly evolvable, reliable, and scalable Web applications. With time, it proved to be effective. Indeed, most Web applications follow the REST architecture and to the best of our knowledge, no alternative to these principles has been proposed in the past 10 years.

It is however not clear for developers how to leverage this constraint and more broadly how to design REST API interfaces.

1.3 Separation of concerns in modern Web architectures

Figure 1.7 illustrates the differences between the architecture of Web applications between the second and third phase of the Web systems architecture. In addition, we observed that the Web has indeed become the global distributed system envisioned in its early days, where systems highly interact with each other. And reviewing how the architecture of the Web evolved with time in Section 1.1, we notice that every major evolution of the Web made it an even more open and distributed platform.

The more the Web becomes distributed the more the concerns are separated between the components. When we limit our scope to observing the components used to provide Web applications and websites to the end-users, we observe the following: (i) mobile applications, (ii) Web application frontends served by content-delivery networks or rendering servers (the markup and javascript of the JAMstack) and (iii) Web APIs.

This architecture is finally a very classic service-oriented architecture [43]. Web APIs provide services and consume other Web APIs' services and both the mobile applications and Web application frontends are service consumers. This separation is due to the fact that Web APIs are hosted on secured and controlled cloud environments that are highly available for the end-user and that are secured enough to store the sensitive data of all the users of an application. On the other hand, the mobile applications and Web application frontends run on the end-user device, an unsecured and uncontrolled environment. Mobile applications and Web application frontends are often referred to as mashup services performing client-side mashup [47].

Consequently, because it is trustful, the Web API is both a data owner and service orchestrator. Its role is to secure access to the data that it owns and to ensure that it will evolve consistently. It can also have the responsibility to propagate changes to other Web APIs and ensure that they have successfully been propagated. A Web API, also named a service in the

39. Complete transcript of the interview of Roy Fielding on REST versioning and hypermedia: `https://www.infoq.com/articles/roy-fielding-on-versioning/`

web, is valuable if it holds valuable data and is able to offer an API that is easy to work with. It should give easy to read data representations to its users and meaningful operations to modify the data. **It is fundamental to understand that Web APIs are all about data management. They are therefore not responsible for how the data will be displayed to the end-user and they should not be.**

Moreover, while in the Web 2.0, it was common to have big teams working on a single MVC application that managed both the application frontend and backend, in the modern web, different teams, and even different companies, build the frontend and the backend of an application. For example, one company can build an open service, such as Stripe did to manage payments [40] and another company can build the frontend, for example, a money exchange application such a Tipeee [41]. As a result, the design teams are different. This separation is also visible inside corporations, where software departments are structured in product teams. Each product team is responsible for its own product, usually a single artifact such as a Web API. The goal is to maximize reuse and cost-efficiency.

Therefore, with the separation of the teams' responsibilities, a team designing an API generally listens to the needs of other developers while a team designing a user interface listens to the needs of the end-user. It is indeed relevant to have a design team dedicated to a user interface. As billions of people from all over the world use the web, there are many kinds of users, with very different abilities, cultures, and needs. It is therefore essential for a product to target a precisely identified type of users. Designers then study their needs and how they will interact with the product to improve the ergonomic. They identify the important data and features that the product should offer. Next, user interface designers will visually improve the user interface to make it more appealing. The following step in the software design is to identify what Web APIs to reuse and what Web APIs to create. As the last step, the Web application will be implemented.

As a consequence, this is the user interface that mainly drives the design of the Web APIs, because this is the UI that defines what data and features it needs, not the opposite. While compromises are obviously done, the Web API very rarely drives the user interface design. It is therefore not relevant to automatically generate user interfaces on top of Web APIs. There is still one exception to this rule: the creation of administration user interfaces. These particular interfaces must present the data in a raw format. They are therefore a good candidate for user interface generation.

To understand the research presented in this document, this is fundamental to memorize this point: user interfaces are designed for an accurate user target that shaped the data and features to offer. Therefore, the user interface determines which

40. Stripe - `https://stripe.com/fr`
41. Tipeee - `https://en.tipeee.com/`

Web APIs to use or not and can shape the evolutions or creation of Web APIs.

1.4 API contracts: the backbone of the modern web

We saw previously that the modern Web is highly open and distributed. Concretely, it is made out of standalone software services that communicate with each other, each service offering a given set of features. So it became what is named a global service-oriented architecture.

In this service-oriented architecture, the users expect websites and Web applications to be available anywhere at any time and to be fast. Then because websites and applications rely on other services available on the web, the application developers need a guaranty that the services that they use will behave reliably.

The need for services that behave reliably and can therefore be trusted is not new neither limited to service-oriented architectures. Indeed, service-oriented architectures are one form of component-based architectures themselves a form of object-oriented architectures where the same need has been observed a long time ago. The most popular methodology used to achieve trust has thus been presented with object-oriented architectures in mind, this is the *Design by Contract* methodology introduced by Meyer in 1992 [48]. Further work applied this methodology to many domains including service-oriented architectures and the web.

The rest of this section introduces the *Design by Contract* approach and explains how it has been applied to the engineering of Web systems in order to enable the modern Web to be highly available and reliable.

1.4.1 Contracts in software engineering

Design by Contract [48] is a software engineering methodology. Based on the idea that assertions can be interesting to specify software, it proposes to consider that software entities (objects, components, services, etc.) have responsibilities towards the other entities with which they interact. These responsibilities are thus based on rules that can be specified. Specifications (or contracts) can thus be created for each entity of the system. As a result, the interaction between the entities of a system can be limited to these contracts.

In an effort to clarify what can be specified with contracts, A.Beugnard *et al.*. extended Meyer's work and proposed a taxonomy that identifies four types of contracts [49]. Syntactic contracts enable the system to work by specifying data structures and types, behavioral contracts specify each operation, typically with a precondition and post-condition, synchronization contracts specify the coordination of operations, and quality of service contracts quantify a few features associated with operations such as availability or performance.

This classification clearly helps to structure the specification and to understand the coverage of requirements a component has. So it helps design tools that verify the behavior of the

components to bring trust.

This work has been well received by the software engineering community. It has been cited approximately 900 times. Ten years after the publication of this work, authors surveyed domains such as service-oriented architecture where the notion of contract aware components has been influential [50]. We explore the benefits of the Design by Contract methodology and of this classification in service-oriented architectures in the next subsection.

1.4.2 Contracts in service-oriented architectures

Service-oriented architectures decompose software systems into independent remote services that communicate with each other [43]. Within this concept, each service provides a set of reusable functions to its consumers. Basically, a consumer will send a message to a service and will receive a response message that contains the operation's execution result. To make this communication possible, a contract between the service provider and consumer is necessary. In addition, its formalization has many benefits to develop error-free software [51]. A service is self-contained, can be updated independently, and is a black box for its consumer, meaning that the consumer can not have information about the internals of a service that it consumes. As a result, in a single system, the services can all be implemented with different programming languages. Service-oriented architectures therefore homogenize how software entities cooperate and give freedom for their implementation.

On the one hand, they claim to ease the maintenance and evolution of the system's constituents thanks to the loose coupling that they provide. On the other hand, contracts are not set in stone at design time similar to object-oriented architecture. Instead, they are much more volatile as the services can evolve independently. The conditions enabling the cooperation between two services can thus change at any time and hence break the deal. The evolution of a service can thus break its consumers. As a result, while the evolution of a single system is simplified, co-evolution is challenging. While such property may sound like a bug, this is an essential feature of service-oriented architecture.

Such highly dynamic contracts have a significant impact on the way to create, configure and supervise contracts.

According to Beugnard and colleagues, syntactic and behavioral contracts can be automatically created and configured using service discovery mechanisms. This is also possible to do this manually. In this latter scenario, the service providers will publish the contracts that they respect. The service consumers will search for services proposing the features that they are looking for and then write an implementation that follows the contracts. Synchronization contracts are created and configured manually by both the providers and consumers. Finally, quality of service contracts are realized with service level agreements mechanisms.

The supervision of the contracts is specific to Service-Oriented Architecture. Indeed, while

libraries are embedded at compile time so they can not be changed randomly at run time, in a service-oriented architecture, the services can be changed randomly at run time, because the client does not control the execution environment. As a change of the service is likely to imply a change in the contracts, supervision can be used to monitor contracts. The monitoring system may be generated and configured by the contract management system, if any. Otherwise, external monitoring can be set up to obtain a neutral point of view of the compliance of the interested parties to the contract.

Supervision can also be one technique to effectively help co-evolve services in a service-oriented architecture. The maintenance and co-evolution of such systems pose several challenges that have been explored in many research. In this thesis, we extend this work by proposing a new approach to the co-evolution of REST APIs and their clients.

In practice, many technologies have been proposed to homogenize communications in service-oriented architectures. The most popular of them are Web Services based on WSDL and SOAP, messaging relying on RabbitMQ, ActiveMQ, and so on, RESTful HTTP, Apache Thrift, or gRPC. These technological stacks all propose their own tooling to create, configure and supervise contracts. The co-evolution of such systems is therefore dependant on the technologies involved. Since the Web relies on a relatively small set of technologies, the next subsection details how SOA contracts have been applied in the World Wide Web.

1.4.3 Contracts in SOA, applied to the World Wide Web

In the Web, services collaborate by exchanging messages on the Internet following a given protocol. Media types such as JSON are used to define the primitive data types and processing models as well as the serialization formats. Protocols describe interaction models that extend the capabilities of media types. An example illustrating how media types can be enriched with protocols is the Atom Publishing Protocol [52] that defines operations to modify Atom feeds generally formatted with the Atom Syndication Format. Depending on the expressiveness of the media types and protocols, additional technologies may be necessary to specify contracts.

As we saw earlier, the most popular protocol of the early Web was the Simple Object Access Protocol (SOAP) which used XML as a media type. Yet, SOAP does not provide contracts description. So, since 2001 the Web Services Description Language [53] (WSDL) enabled the description of the interfaces of the services and became the de-facto standard for syntactic description.

In addition, some approaches to the description of Web Services covered the other kinds of contracts but none became a widely adopted standard. For example, SoaML proposes to specify the syntactic and behavior contracts with an extension to the UML language [54]. This Oasis standard has driven many research efforts. Similarly, WS-Coordination and WS-BusinessActivity have been designed to be used together in order to coordinate business activities using Web

services and therefore serve as documentation of the synchronization contracts [55]. In [56], Ran proposes a model for Web services discovery where the functional and quality of service requirements are taken into account.

Regarding RESTful Web APIs, JSON rapidly became the preferred data-interchange format. The approaches designed to describe contracts for SOAP-based services could not be reused as-is for REST-based services. New solutions have thus been proposed.

Description languages for REST API contracts

The easiness to build and consume REST APIs has probably been an important factor in its popularity. Yet, because the developers could easily implement a client of a REST API manually, they did not need machine-readable documentation of the service to consume anymore. Indeed, it was not necessary to generate stubs from documentation anymore. A simple textual description of the service was sufficient for the developer to quickly write a code that would consume the API. For a long time, the developers did not feel a need for languages to specify contracts. For this reason, until approximately 2014, most REST APIs were documented in plain HTML documents [57]. We indeed observe that it took a long time to see industrial efforts towards standardization of REST API description formats. Academic research did not focus on the description of REST APIs either.

Syntactic contracts. Referring to Beugnard's classification, Interface Description Languages focus on the description of syntactic contracts.

The need for Interface Description Languages for REST API has been controversial [42] for a long time. This is why most REST APIs were documented in plain HTML documents before 2014 [57].

The first proposition of an IDL for REST API has been WADL [58], an equivalent of WSDL for REST APIs published by Sun Microsystems in 2006 and then submitted to the W3C in August 2009. WADL has yet never been standardized or massively adopted. Similar to WSDL, WADL uses the XML format to describe the API and this is the main reason why it did not gain much interest. In 2011, Tony Tam proposed Swagger that later became the OpenApi Specification. Along with an IDL, Swagger proposed a Web user interface offering an ergonomic way to browse the specification and the possibility to test an API right from the Web page, which is one of the main factors of its success. Swagger also proposed a code generator. Shortly after swagger, alternative structures for describing RESTful APIs were introduced, the most popular being API Blueprint [43] in April 2013, RAML [44] in September 2013 and RADL [45] in June 2014. In the end, Swagger drove more interest than the others since mid-2014.

42. https://bitworking.org/news/2007/06/do-we-need-wadl/
43. https://apiblueprint.org/
44. https://raml.org/
45. https://github.com/restful-api-description-language/RADL-2.0

Behavior contracts. We could find no proposal to specify behavior contracts. Moreover, in the Web, API providers are not always willing to communicate them. Three reasons seem to prevent REST API providers from publicly disclosing such information, especially postconditions. First, it could reveal business-critical strategies to competitors. Second, developers can rely on advanced testing strategies to verify their systems. And third, such behavior contracts can be highly dependent on the user, for example in the context of bank account management. Unfortunately with such practices, consumers can not be aware of the contracts in advance. Thankfully, the HTTP protocol offers detailed error statuses that practitioners are accustomed to exploiting to give detailed feedback to their users. Preconditions and postconditions can therefore be discovered by the end-user at runtime. Moreover, IDLs give advanced schema description possibilities to enable the description of some of the preconditions, for example, "the amount of money to transfer between bank accounts must be superior to 0". In autonomous systems, the elements that can be communicated are unfortunately regularly documented in plain text. **Considering the high volatility of the contracts in the web, this is an important risk that we try to address in this thesis.**

Synchronisation contracts. The description of synchronization contracts has driven the interest of the business process modeling community. In [59, 60], C.Pautasso proposed an extension to the BPMN standard to describe synchronization contracts. BPMN engines can therefore be used to manage the synchronization. The same author proposed Jopera, a visual composition language [61]. BPEL for REST is another approach that extends BPEL to enable contract-based RESTful service composition [62]. The JavaScript-based S language, on the other hand, is a domain-specific language designed for server-side use. It promotes a programming model based on explicit (control-flow) and implicit (process-level) parallelism control [63].

Quality of service contracts. Quality of service contracts are also mainly described in plain text documents. While interesting works address the subject, they are not closely related to the work presented in this thesis and will therefore not be detailed.

REST API contract management: alternatives to specification languages

Instead of giving complete freedom to the developer to implement a system and then offer contracts description languages, some proposed different approaches. Among them are application protocols designed on top of REST that enforce the contracts, or frameworks. Some API providers also propose libraries to interact with their REST API. They therefore rely on modern programming languages constructs to provide design-time contract checking. Another approach is to rely on models. Model-Driven approaches propose to generate code from models of the application where the contracts are formalized. There is also a trend towards automated service composition and contract checking leveraging on the Semantic Web.

Frameworks as contracts. In order to standardize the practices around well-defined con-

tracts, some proposed to use frameworks [64, 65, 66, 67]. While these are mainly specified within plain text documents, the standardization effort enables all the interesting properties claimed by the design-by-contract approach such as documentation, verification, code generation, and tooling.

One such approach has been proposed by Microsoft. OData [64] is a framework that proposes a unified way for tracking changes, defining functions/actions for reusable procedures, and sending asynchronous/batch requests. With OData Common Schema Definition Language, the developers describe the schema of the data and can then use the standardized methods of the protocol to interact with the data. That approach did not gain much traction because it is made of many very long specification texts and it is a rigid framework.

OpenSocial is an initiative to standardize APIs for building social Web applications that started in 2013 and moved to the W3C Social Web Working Group in 2015. The group published the Social Web Protocols note. It contains a collection of standards that enable various aspects of decentralized social interaction on the Web [65].

These approaches have proven to bring benefits for domains where standardization is relevant. Yet, for many domains and providers, the ability to provide custom implementations and protocols is seen as a business-critical feature that enables the provider to propose a differentiating experience or differentiating features. We therefore observe that in practice, these approaches are limited to certain domains such as social media and that most Web systems do not rely on them.

Software Development Kit as contracts. Contracts can also be provided by the REST API provider within a library. Such kinds of libraries are usually named Software Development Kits.

In [51], authors leverage the annotations of the .NET language to provide advanced contracts support. They have been able to augment the official Facebook API with static behavior contracts checking and proved that it can reduce the number of errors in third-party applications. Such a method can therefore reduce cumbersome manual testing.

We also observe that companies providing REST API intended to be consumed by hundreds to millions of systems regularly propose SDK in many programming languages. When they do so, these companies favor the SDK in place of the REST API. Famous examples are Twitter that has an SDK available for 29 languages as of June 2021 [46], Facebook, Google and Stripe [47] that is often referred to as an API that offers a great developer experience.

Finally, Software Development Kits have proven to ease the consumer's development but one library must be created and maintained for every programming language to support. They therefore require an additional effort from the API provider or the community. One of the

46. `https://developer.twitter.com/en/docs/twitter-api/tools-and-libraries`
47. `https://stripe.com/docs/api`

limitations of this approach is that all the SDKs must be evolved when the REST API evolves. In addition, the REST API must remain available and should still be well documented. In the end, SDKs are relevant for REST APIs that will be consumed by hundreds to millions of consumers, because the potential financial benefits are more important than the cost. On the other hand, this is a too expensive solution for the majority of the REST APIs, that are consumed by few to tens of clients. Moreover, an important technical limitation remains. Indeed, the client software can only be evolved at design time because the contracts are sealed in the code.

Model-Driven Engineering techniques. Model-Driven Engineering (MDE) techniques focus on creating and exploiting domain models. The goal is to ease the collaboration between the domain experts and the developers by removing the technical details from the design process. The people involved in the development of the service can therefore focus on the design of the contracts. When the team is happy with the produced contracts, generators produce code and documentation from the models. These generators can generally be configured by technical experts to adapt the generated artifacts to external constraints such as architectural or security requirements.

Numerous research studies have been conducted to apply MDE techniques to REST APIs. As a result, many approaches use different technologies for a similar goal: generating REST API server or client implementations from syntactical contracts [68, 69, 70, 71, 72, 73]. Some of the approaches generating the server code also generate the REST API documentation in various formats such as WADL. To extend this work, some also consider behavior contracts and sometimes leverage the HATEAOS constraint of the REST architecture to propose a solution to synchronization contracts [74, 75, 76].

MDE techniques are beneficial to the initial design, implementation, and integration of a REST API. They can also help manage contracts specification, especially syntactic contracts. However, maintaining the link between the models and the code is challenging. Indeed, it is a complex task to update the generated code upon modification of the models. It is also challenging for the developer to update the code accordingly. Moreover, developers are not confident enough with code generators to make heavy use of them, in particular with interpreted programming languages such as JavaScript, the language to design Web user interfaces. Code generators are consequently more often used for the back-end of REST services and within non-complex modules, usually the interface layer but not the domain logic or the behavior. MDE has thus not yet proven to be an effective solution to maintain contracts between API providers and consumers. They are instead mainly used for the maintenance of a single artifact.

REST API Contracts: State of the practice

We saw in the current section that the notion of contracts in software architecture and in web-oriented architecture has received a lot of interest. Many languages, methodologies, and techniques have been proposed with very different approaches. One might therefore wonder how the majority of the systems in the Web are designed? How do they specify and make use of contracts?

The methods to implement a REST API server or a client of a REST API have not changed much with time. Developers still like the lightness of the approach and the ability that it gives to implement a server or client in a few lines of code. The JSON format stays the preferred data-interchange format.

To specify contracts, OpenApi (previously Swagger) has become the de-facto standard to document syntactic contracts and schema models (thus a small part of the behavior contracts). API providers can also leverage the language to specify hypermedia controls in order to document the synchronization contracts but these constructs have been introduced in July 2017 and are still rarely used. Most of the time, behavior, synchronization, and quality of service contracts are specified within plain-text, either in the description field of the OpenApi specification or on API reference websites. Developers therefore read these contracts and translate them into code to implement a working client. Such a practice therefore introduces a tight coupling between the API client and server.

There are good reasons that motivate the documentation of contracts in plain text. Nowadays, REST APIs are often designed to be used by many consumers. Countless scenarios can therefore be created from the composition of API calls. It is thus very challenging to document them all and would probably make the documentation difficult to navigate. API providers therefore prefer to write plain text that presents basic usage scenarios and let the consumers decide on the suite of API calls to make to implement their own business processes. In addition, these processes may require the use of several REST APIs from different providers. Requiring multiple providers to document cross-provider synchronization contracts would not be realistic.

Such practices leave many challenges open. One of them is the integration of the REST API contracts into a new client. Indeed, developers still write the code themselves. While this is not very error-prone for very basic REST APIs such as CRUD REST APIs, it can lead to many errors for REST APIs with a complex business logic such as REST APIs using the CQRS pattern [77]. Another challenge is the co-evolution of Web systems relying on REST APIs. Indeed, a tight coupling is introduced when the developer translates the plain-text contracts into client code. Then, when the REST API evolves, the evolutions must be identified and the client code must be adapted. One contribution of this thesis is a new approach to this challenge.

1.5 A Web "interpretable" by machines

The previous sections of this document presented the evolution of the architecture of the World Wide Web along with how contracts have been specified and used to enable the Web to become a highly available and quite reliable global service-oriented architecture. In previous sections, a complete research domain was omitted: the Semantic Web. The main reason is that the works of this domain are based on foundations that have not been implemented in the modern web. We therefore find it relevant to discuss this domain in a dedicated section.

While the Semantic Web did not become the main architecture of the Web, it got the interest of many researchers and is still an active research topic. Since 2000, it proposed to make the Web "interpretable", and not only processable, by machines. In this domain, researchers studied and proposed solutions to many of the problems that are still active in the modern web. This work can then be reused to design solutions adapted to the current state of the art and practice. Interesting topics are: service description, automated service discovery, automated service integration, and composition. They are discussed in the rest of this section.

1.5.1 Presentation of the Semantic Web

At the first World Wide Web Conference in 1994, Tim Berners-Lee claimed that the Web has become an "exciting world" for users but that it contains very little machine-readable information. “The meaning of the documents is clear to those with a grasp of (normally) English, and the significance of the links is only evident from the context around the anchor. To a computer, then, the Web is a flat, boring world devoid of meaning.” He identified two things that would enable the Web to be **"interpretable"** by machines.

Enablers of an "interpretable" Web

According to Tim Berners-Lee, two things can make the Web interpretable by machines:

1. the possibility to add machine-readable semantics into documents
2. making the semantics of the links explicit

Unsurprisingly, the World Wide Web Consortium (W3C), which Berners-Lee founded later that year to coordinate the standardization of the Web, put a focus on standardizing Semantic Web technologies [1]. As a result, the Resource Description Framework (RDF) has been standardized in 1999.

Concretely, the difference between a Web that is interpretable or processable by machines is the following: within an interpretable Web, algorithms can intelligently navigate and collect

data by following meaningful links and semantically-rich data whereas, within a processable Web, algorithms can not make a difference between the different links or pieces of information. For example, HTML makes the Web processable. Automated software agents can find all the links on a page and follow them. However, these agents are not able to differentiate the links based on their meaning. All links are equal in a processable Web while in an interpretable Web the software agents can differentiate links and intelligently follow them, for example, to find a given information. This is precisely the semantic descriptors on data and links that enable them to do this.

The idea behind RDF is quite simple: information can be represented as individual statements. RDF represents data as a triple in the form subject-predicate-object. The subject is the (web) resource to describe, the predicate is the aspect to describe and therefore expresses the relationship between the subject and object. To represent that strawberries have the color red in RDF, the subject would be "strawberries", the predicate would denote the relationship "has the color" and the object would be the "red" color. In fact, RDF has been designed for the web. The subjects and predicates must therefore be Web resources, accessible with a URL. One URL would therefore be necessary to represent the concept of strawberries, another for the link "has the color" and a third one for the red color.

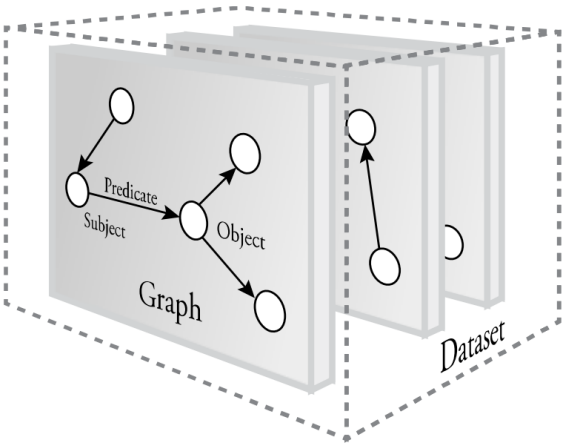


Figure 1.8 – The RDF data model, from [1]

The Resource Description Framework is an abstract model, meaning that it can take many forms. The original serialization format is RDF/XML but others exist such as RDFa [78], Turtle [79], JSON-LD [80] and more. RDFa is for example designed to embed machine-interpretable metadata into HTML documents. Figure 1.9 gives an example of an HTML document with RDFa annotations. The document can be parsed to extract many triples that will form one to many knowledge graphs resulting in a dataset, as presented in the Figure 1.8. For example, with the *base* and *meta* tags of the document's header, the triple *"http://example.org/john-d/*

dc:creator Jonathan Doe" can be extracted. The presented information is that the creator of the page http://example.org/john-d/ is Jonathan Doe.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML+RDFa 1.0//EN"
    "http://www.w3.org/MarkUp/DTD/xhtml-rdfa-1.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:foaf="http://xmlns.com/foaf/0.1/"
    xmlns:dc="http://purl.org/dc/elements/1.1/"
    version="XHTML+RDFa 1.0" xml:lang="en">
  <head>
    <title>John's Home Page</title>
    <base href="http://example.org/john-d/" />
    <meta property="dc:creator" content="Jonathan Doe" />
    <link rel="foaf:primaryTopic" href="http://example.org/john-d/#me" />
  </head>
  <body about="http://example.org/john-d/#me">
    <h1>John's Home Page</h1>
    <p>My name is <span property="foaf:nick">John D</span> and I like
      <a href="http://www.neubauten.org/" rel="foaf:interest"
        xml:lang="de">Einstürzende Neubauten</a>.
    </p>
    <p>
      My <span rel="foaf:interest" resource="urn:ISBN:0752820907">favorite
      book is the inspiring <span about="urn:ISBN:0752820907"><cite
      property="dc:title">Weaving the Web</cite> by
      <span property="dc:creator">Tim Berners-Lee</span></span></span>.
    </p>
  </body>
</html>
```

Figure 1.9 – Example of an HTML5 + RDFa document. Source: `https://en.wikipedia.org/wiki/RDFa#HTML+RDFa`

As stated previously, objects and predicates must be valid Web resources, i.e. accessible with a URL. RDF thus leverages on the property that one URL should never refer to two different concepts at the same time, even ones that may seem equivalent. This property is very powerful. Indeed, in the natural language, there are words that have multiple meanings but on the web, especially with RDF, this statement is false. In fact, if a word has two meanings, two different URLs will be used to refer to the word. Thanks to this property, the machines that interpret the Web are not subject to the ambiguities to which we humans are subject. Leveraging this very powerful property, the Semantic Web community has been able to design algorithms, tools, and standards with interesting properties. They are detailed in the rest of this section.

When the Semantic Web started, there was obviously no Web resource available to describe things such as the strawberries used in a previous example or the friend relationship. It has therefore been necessary to create vocabularies that can be used with RDF. They are formally named ontologies. RDF Schema and the Web Ontology Language (OWL) [81] have been created

for this purpose.

RDF Schema (RDFS) offers constructs to describe classes, class hierarchy, data types, and other schema properties similar to object-oriented programming languages. It also enables the description of data containers such as sets and lists. OWL can be seen as an extension of RDFS adding many concepts, which makes it a far more expressive modeling language. With OWL, it is for example possible to express that in an ontology describing families, a "hasMother" property is only present between two individuals when "hasParent" is also present, and that individuals of class "HasTypeOBlood" are never related via "hasParent" to members of the "HasTypeABBlood" class (the example is from Wikipedia).

In addition to the presented work, the W3C standardized other technologies to propose a complete technological stack for the Semantic Web. The Figure 1.10 that is an adaptation of Berners-Lee original figure [48] and represents this stack.

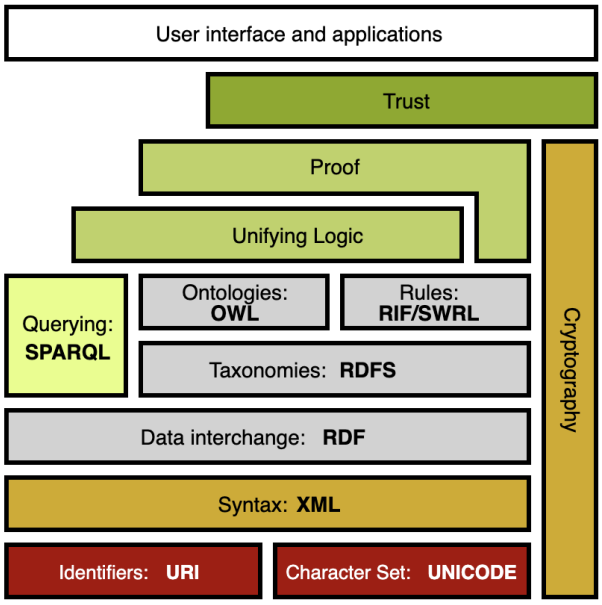


Figure 1.10 – Semantic Web technological stack. Source: Wikipedia

Before 2006, the technologies of the Semantic Web stack did not require the identifiers to be dereferenceable. Concretely, IRIs (Internationalized Resource Identifier) were used in place of URIs (Uniform Resource Identifier). This means that the identifiers used to reference terms, such as the strawberry concept, did not have to be valid Web resources. Copy-pasting them in a Web browser could lead to nothing. They were therefore opaque identifiers. So, the Semantic Web could not be browsed. In an effort to change that, Berners-Lee published the following Linked Data principles in 2006 [82]:

48. https://www.w3.org/2000/Talks/1206-xml2k-tbl/slide10-0.html

— Use URIs as names for things

— Use HTTP URIs so that people can look up those names.

— When someone looks up a URI, provide useful information, using the standards (RDF*, SPARQL).

— Include links to other URIs so that they can discover more things.

These four principles marked a turning point in the history of the Semantic Web. People started to envision it as an enabler for the publication and consumption of vast amounts of structured and open data. One result is the Linked Open Data cloud, represented in Figure 1.11.

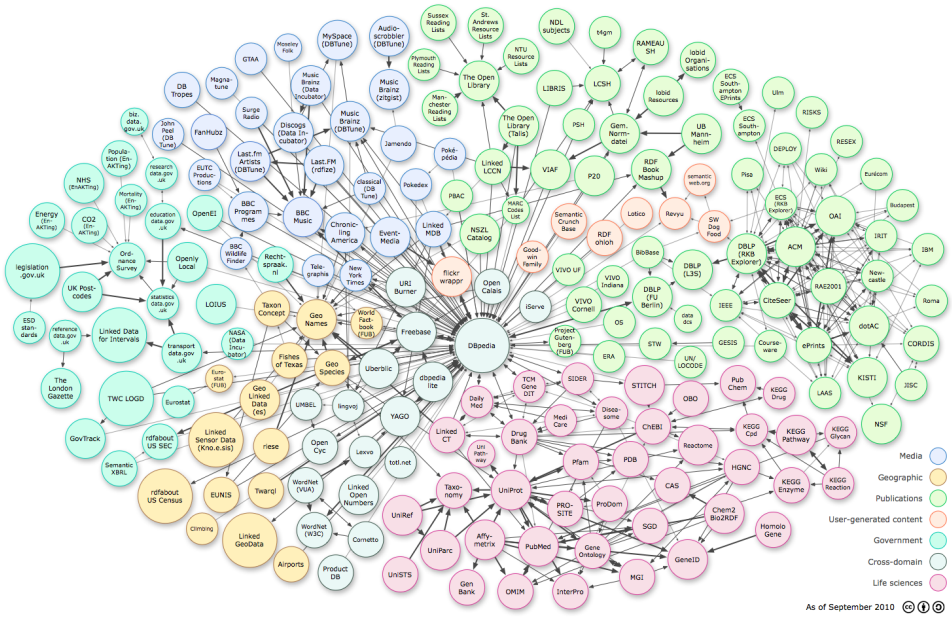


Figure 1.11 – Linked Open Data Cloud representation - Source: `https://lod-cloud.net/`

The Semantic Web and Linked Data rapidly gained traction into the Web Services community and created new research topics such as Semantic Web Services and Semantic Service Oriented Architecture. Research on Semantic Web Services (SWS) has been devoted to reducing the extensive manual effort required for manipulating Web services by enhancing them with semantic information. Such challenges remain existing in the current architecture of the Web because SWS have not been widely adopted. Furthermore, the work presented in this thesis leverages Semantic Web Services research to ease the co-evolution of RESTful APIs and their clients. The rest of this section presents an overview of the research on Semantic Web Services. The goal is to give the reader a general idea of the possibilities offered by Semantic Web technologies.

1.5.2 Semantic Web Services

Research on Semantic Web Services has been devoted to reducing the extensive manual effort required for manipulating Web services by enhancing them with semantic information. Concretely, Semantic Web services can be defined as Web services whose descriptions are annotated by machine-interpretable ontologies so that other software agents (e.g. Web user interfaces or mobile applications) can use them without having any prior 'built-in' knowledge about how to invoke them.

In practice, to enable a software agent to use a service without having any prior knowledge of how to invoke it, it must be enabled to automatically discover, select, compose and execute Semantic Web Services.

Semantic Service Description

Similar to humans, machines would not be able to discover, select, compose and execute Web services without a documentation of the services. Here, documentations must contain machine-interpretable descriptors. A first challenge has therefore been to make this possible resulting in new standards, automation techniques, languages, and ontologies.

The first technology to enable service providers to enrich their documentation with machine-interpretable semantics has been SAWSDL [83] (Semantic Annotations for WSDL). Later, an adaption for RESTful services was proposed [84]. Researchers however observed that most Web APIs were documented in plain HTML documents and that developers were generally not willing to write a new documentation from the ground up. So they proposed to create machine-interpretable descriptions on top of existing HTML descriptions by using microformats [85]. hRESTS [86] and RDFa [78] propose that. MicroWSMO [49] adapts SAWSDL to extend hRESTS. In addition, WSMO-Lite annotations can be used to make WSDL and hRESTS-based services interoperable. Similarly, SA-REST [87] applies SAWSDL to RDFa instead of hRESTS. These approaches are all correlated to a set of technologies. An ontology to describe RESTful services within any RDF compliant format has therefore been designed, namely Hydra [88]. Similarly, the Minimal Service Model MDM) enables an operation-based description of Web APIs [86]. It also aims to enable the reuse of existing Semantic Web Services and to provide means for integrating heterogeneous services.

These technologies are concerned with describing the syntactical contracts (input and output of the operations) but omit the functional aspects. To overcome this, OWL-S [89] adds support for pre and post-condition description and hence allows to describe functional relations using a variety of expression languages (SWRL, DRS, KIF, and more). An alternative is WSMO [90]. RESTDesc [91, 92] proposes to approach things differently: instead of relying on the input and

49. `http://www.wsmo.org/TR/d38/v0.1/20080219/`

outputs to describe the service functionalities, it proposes to describe the functionalities directly. On the other hand, Rauf and Porres demonstrate that OWL2 can be used to behaviorally enrich semantic RESTful interfaces [93]. Besides, in an effort to propose a solution with a low entry barrier, authors of [94] designed a solution similar to Microformats that leverages existing HTML tags and a language to partially describe syntactical, functional, and behavioral contracts.

So far we mentioned research grounded on the Semantic Web and especially on XML. However, many Web developers are reluctant to integrate such technologies into their applications, which may explain why they are not widely used. Lanthaler and Gütl proposed new technologies based on JSON to be as close as possible to what developers are used to. SEREDASj [95] is a semantic description language for JSON-based RESTful services. It focuses on the description of JSON representations and Linked Data. However, they found that in practice developers struggle with the separation of data into representations and descriptions [1]. They therefore proposed JSON-LD [96] (JSON for Linked Data), a new RDF serialization format that has a 100% JSON-conformant syntax. Developers can use them without changing their tool-set nor practices and hence focus on the description of the domain. JSON-LD is a community effort that has been well received. It is a W3C recommendation since 2014 and its adoption increases year after year. According to w3techs, in June 2021, it is used by 37.4% of all the websites [50].

For a more detailed survey on the semantic description of REST APIs, the reader may refer to [97].

Apart from technological enablers and service description ontologies, researchers also explored ways to ease or automate the semantic description of Web services. In [98], authors present MWSAF, a framework for semi-automatically marking up Web service descriptions (in WSDL files) with ontologies. Alternatively, Siqueira *et al.* present a framework for the semantic description of restful APIs implemented with JAX-RS [99]. They leverage on JSON-LD and Hydra. Similarly, SWEET [100] helps RESTful API providers annotate the HTML documentation of their APIs. SWEET takes as input an HTML Web page and offers functionalities that enable users to annotate the service properties and to associate semantic information to them. It leverages on hRESTS and MicroWSMO.

The semantic description of services on the Web has therefore been an active research topic. Researchers that explored this topic identified numerous opportunities. Among them is the possibility to automatically discover relevant services and the automatic integration and composition of such services. The next two subsections present the research on these topics.

Automated Service Discovery

A key idea of the Semantic Web is that with semantic descriptors, information search can be much more precise. So, the semantic description of Web APIs enabled to automate the search

50. `https://w3techs.com/technologies/details/da-jsonld`

and discovery of the Web APIs offering a given set of functionalities. The research in this area are affiliated with the topic of Semantic Web Services Matchmakers.

In a survey conducted by Klusch in 2008 [101], six dimensions are identified for the comparison of SWS matchmakers: (1) the markup languages used for describing the semantics of Web services, (2) the SWS discovery mechanisms, (3) the SWS discovery architecture, (4) the approaches used for SWS matching and the extent of matching for service selection, (5) the semantic parts of SWS for service matching and selection, and (6) the platforms or collections for testing the performance of SWS matchmakers. The first has already been discussed in the previous subsection and performance (6) will not be discussed here because it is too far from the work presented in this thesis. We therefore study 2, 3, 4, and 5 here and reduce this classification to two broader categories: (1) finding the services and (2) matching the service offer with the developer's need. For a detailed survey on SWS matchmakers, the reader may refer to [102, 103].

Mechanisms to search for services have mainly been explored in two paradigms: service registries and peer-to-peer [104, 103] (P2P). The first is the dominant approach and the most known registry is Universal Description, Discovery, and Integration (UDDI), though limited to XML files. However, some claimed that it is difficult to realize the catalog-based approach in the open and massively distributed environment of the Web. P2P overlay networks provide a solution to this problem. There are also other types of mechanisms such as index-based (e.g. Google) and agent-based [105] but they represent a very small proportion of the research.

The semantic matching approaches can be classified into four categories based on the involvement of logic reasoning and machine learning: logic-based, non-logic-based, hybrid, and adaptive. Non-logic-based matchmakers are limited to syntactic matchmaking [106]. Logic-based matchmakers can process complex rules and address the four kinds of contracts seen previously [107, 108]. Hybrid approaches augment logic-based methods with content-matching techniques to improve recall and precision [109, 110]. Adaptive matchmakers use machine-learning techniques. While these categories are easy to read, matchmakers are more often compared based on the supported technology (WSMO, OWL-S, SAWSDL). With so many available matchmakers, they should be analyzed in detail as their properties mainly depend on their design. In the end, the selection of a matchmaker depends on the expected properties. To help practitioners compare matchmakers, performance testing platforms such as Semantic Service Selection (S3) Contest and SWS Challenge have been created.

The matchmaking techniques cited so far are all concerned with fully automating matchmaking and composition. They require a highly accurate and trustful description of services and queries, as well as an unambiguous matching process. Toch *et al.* propose to design matchmakers that help humans select the service to use instead [111]. In such a scenario, the developers control the service selection and therefore are not expecting a fully reliable matchmaker. Authors could therefore use approximate matching.

Finally, it is noticeable that matchmakers presented many challenges for which very different solutions have been explored. Evaluated with S3 Contest, the precision of matchmakers ranges from 50% to 92% [112]. While these results seem good, they are not sufficient to give full autonomy to these systems. This is why the approach of Toch *et al.* is interesting. The presented research remains very interesting and deserved some attention. Leveraging matchmakers, some have been able to propose automatic service integration and composition of Web APIs.

Automated Integration and Composition of SWS

Sometimes, one Web API is not sufficient to achieve one's goal. In such cases, Web APIs must be composed to achieve a goal. For example, if to go to one place, it is necessary to book a train and rent a car, a trip planning service would likely need to use two Web APIs. One for the train booking and another one for the car rental. The automatic service discovery covers the search for a service providing a given service, such as booking a train. However, this is in the domain of automated service composition that such complex scenarios have been studied.

There are two steps to automatically compose services. First, a structure of the composition must be created, and then the atomic services to invoke should be selected. Most methods fall in the realm of workflow composition or AI planning.

Research in the former category is usually concerned with only one part of the process. For example, they automate only the selection of certain atomic Web services that implement a manually defined composition scheme [113, 98]. This is finally a small addition to the service discovery methods presented before. Other approaches create the structure of the composition. Basically, the developer that needs a service specifies his need in terms of inputs, outputs, goals, preconditions, and effects. Then, from a list of known services, matching algorithms try to identify a composition answering this need. For example, SPICE [114] and DynamiCoS [115] use a graph-based formal model, namely the Causal Link matrices, to find the best composition. SeGSeC [116] also leverages a graph-based approach to propose a middleware enabling dynamic service composition. In [117], authors compare two different graph-based methods: sequential and non-sequential composition.

Automated approaches, on the other hand, usually rely on artificial intelligence planning techniques that leverage semantic descriptors to ease problem representation and solving. Each service of a Web API is modeled as an action with preconditions and effects. A logical theorem prover or AI planner then tries to identify the suite of actions producing the desired goal, while sometimes also considering external factors (API availability, cost, etc.). The solver will have to match the input and output of the operations, taking into account the new resources that will be created along the way. Indeed, it may be required to reuse a resource created by an action to execute another action. Many approaches have been proposed in this field [118, 119, 120, 121, 122, 123, 124, 125, 126, 127]. For a framework assisting the classification of AI planning

approaches, the reader may refer to [128].

1.5.3 Bridging the gap between Semantic Web Services and the modern Web

In the previous subsection, we introduced the Semantic Web Service research aiming at automating tasks that developers do manually, namely the discovery and composition of Web APIs. We also presented the technologies enabling Web services descriptions to be enriched with semantic descriptors. However, the presented work does not propose a practical solution to design or even implement loosely coupled architecture leveraging machine-interpretable semantics. Consequently, this section presents the research that proposes such kinds of architectures that have interesting characteristics compared to the current architecture of the World Wide Web.

As a first example, in [129], the authors introduce an architecture and prototype based on Linked Data principles and service-orientation to resolve the integration issues for sharing educational resources. Aside from the initial goal of replacing the widespread use of unstructured text for describing resources with controlled vocabularies and open data sets, they have also been able to implement a query layer on their service that makes distributed queries across distributed and heterogeneous educational repositories (via their services/APIs) on the fly. They are therefore able to adapt in real-time to the addition and deletion of services on the Web without prior knowledge of the service's interface. This work is a good example of architecture leveraging semantic Web services research. Indeed, they adhere to the linked data principles, leverage domain-specific ontologies for the data and the Minimal Service Model for interoperability and automated service discovery.

In [130], the authors propose a lightweight declarative rule language with state transition systems as a formal grounding that enables the development of data-driven applications built upon the RESTful manipulation of Linked Data resources. Concretely, the resources can be manipulated with the standard CRUD operations mapped to the HTTP verbs, and an OPTIONS request on a resource returns the service description including the resource schema. Leveraging their rule-based language, programmers can define an interaction pattern for their client applications. With their framework, the authors propose a loosely coupled architecture for CRUD manipulation of Linked Data and hence enable good interoperability. However, they limit data manipulation to CRUD operations that are not appropriate when meaningful modifications are necessary, which the CQRS pattern emphasizes.

Khalili *et al.* propose to create flexible and reusable Web user interface components driven by Linked Data, that abstract the complexity of the underlying Semantic Web technologies to enable Web developers who are not experts in Semantic Web to develop interfaces that view, edit and browse Linked Data [131]. As a result, they ease the development of Web interfaces leveraging Linked Data technologies which provide loose coupling. While this approach is not suitable for enterprise applications due to the complex business rules they manage, it seems to

be a promising approach for open data applications.

In [132], assuming that developers will express the data needs of their applications in SPARQL queries, Serrano and colleagues propose a middleware that automatically identifies the set of Semantic REST APIs to invoke in order to respond. Expressing data needs with SPARQL is not realistic nowadays, but we observe that GraphQL has gained traction from developers in recent years. We therefore think that it could be interesting to place a component translating semantically annotated GraphQL queries into SPARQL queries in front of this middleware. For example, HyperGraphQL [133] or GraphQL-LD [6]. Such an approach could therefore seduce front-end developers that tend to favor GraphQL to REST in recent years, while also seducing service providers who would gain more flexibility towards the evolution of their services.

1.5.4 Synthesis

This section presented the basis of the Semantic Web and showed that many researchers studied how machine-interpretable semantics could be used to automate service discovery, service integration, and composition. It also covered the semantic description of the services. Finally, we reviewed some initiatives aiming at implementing loosely coupled architecture that maximize reuse within the context of the modern web, i.e. highly distributed, where most services expose a RESTful API.

Most of the research on this topic has been published before 2010 so it is now obvious that those technologies did not reach massive adoption, even though they seemed promising. One of the reasons for this is the perceived complexity of Semantic Web technologies by Web developers, and their grounding on XML which Web developers do not like. In addition, the tooling to design and reuse ontologies is not sufficient. Developers will not be willing to build semantic Web services without easy manipulation of the ontologies. Also, most approaches were designed for full automation but are not precise enough to be used in such a way. Last, while we saw that interesting examples and architectures have been presented to leverage these technologies in the modern web, we think that there is a lack of a use case that brings real value to the developers. Also, more work is necessary to make these technologies easy to use by Web developers, and more communication is necessary too.

We think that such a use case can be about the co-evolution of systems in the Web, as the evolution of Web APIs is the major challenge faced by Web API providers nowadays. So the next section introduces why and presents the major approaches towards the co-evolution of software components.

1.6 Co-evolution

In the first sections of this chapter, we reviewed how the architecture of the Web evolved and how contracts form the backbone of the modern web. Then, we explored the semantic Web technologies that propose to automate many tasks by relying on unambiguous semantic descriptors of data and contracts. Overall, we observed that the Web is a service-oriented architecture. It therefore promotes loose coupling between the components, meaning components can be added, scaled, and removed at any time. This statement is however not true. In practice, developers carefully select the REST APIs that they will use to implement mashups and expect them to be highly available. Moreover, the contracts are hard-coded into clients. As a result, new services and features can be added easily, but modifications or deletions can break other systems. So as soon as a REST API has clients, it must consider them to make decisions about its evolution. We therefore speak of co-evolution.

In this thesis is presented a new approach to the co-evolution in the Web that partially relies on Semantic Web technologies. We therefore review the approaches for the co-evolution of systems in this section.

1.6.1 Four different approaches

The issue of co-evolving an API and its client has been widely studied by the software maintenance community. The many proposed techniques can be classified into four categories. Historically, the first approaches proposed to identify API refactoring with static analysis techniques and then leverage code transformation techniques to propose automatic program repair techniques. Another approach is to automatically synthesize an adapter in order to obtain a specific connector between a customer interested in exchanging with a previous version of the API and the current version of the API. Later, researchers started to search for solutions specifically tailored for highly distributed and open architectures. A new category of techniques emerged which consisted of defining evolution policies that service providers would follow to avoid breaking consumers. The last category of techniques proposes to fully synthesize the client based on the description of the API.

The rest of this section presents the research on these topics and then concludes on their applicability to solving modern Web problems.

1.6.2 Automatic Program Repair techniques

Many approaches identified the different API refactoring combined with the use of static analysis and code transformation techniques in order to automatically identify errors resulting from an API evolution and propose automatic repair mechanisms [12, 13, 14, 15, 16]. As a

representative solution, [17] proposed an Eclipse plugin to automatically update the client code when the API changes.

These approaches are particularly well suited for integration with statically typed and compiled programming languages. Consequently, they have several limitations with respect to the evolution of a Web API. Firstly, the Web world is highly heterogeneous in terms of programming languages; the ability to statically analyze the implementation of APIs and all the clients' code is unrealistic. Secondly, transforming the client code requires, in an industrial context, to go through all the validation phases again before deployment. This could be highly expensive within a project. Still, such techniques can be relevant to deal with the evolution of REST API wrappers that take the form of SDKs.

Behavioral repair techniques, that consist of changing the behavior of the program under repair, i.e. changing its code [16], may also be explored. These techniques use specifications of the expected output as a basis to perform the repair. Having such a specification is however unrealistic with regards to REST APIs. First, no existing description language enables the comprehensive description of a REST API contract. Second, some REST API providers do not necessarily want to disclose all information related to the contract. Moreover, no repair operator is aiming at fixing the UI, especially a Web user interface [16]. Yet, to fix the client of a REST API, usually a Web UI, it is necessary to evolve the UI accordingly.

State repair techniques, which consist in changing the state of the program under repair [16] are another category of techniques that may also be explored, especially the input modification and input rectification techniques. While interesting to address some types of REST API evolution, they are not sufficient to address them all. In addition, we argue that applying such a technique would make it unclear for the developer of the client when he should fix the code or not.

Besides, we observe that automatic program repair techniques are mainly used to fix low-level code or bugs such as data structures, infinite loops, serialization bugs, or condition statements. They do not seem to be widely used for domain-specific problems. In particular, we could not find existing work aiming at repairing clients of evolving REST APIs [16].

In the end, exploring automatic program repair techniques to address the problems of the co-evolution of Web APIs would require composing many existing techniques and relying on an artifact that we are not sure API providers would release (the comprehensive API contract). Other techniques must therefore be explored.

1.6.3 Adapter-based techniques

Several approaches have been proposed to automatically synthesize an adapter [19, 20] in order to obtain a specific connector between a customer interested in exchanging with a previous version of the API and the current version of the API. Such an approach can maintain the

compatibility of the client with the API.

Fokaefs *et al.* [21] proposed to adapt request parameters to the new interface and map the new response model to the one that the client is expecting. However, if a required parameter is added to the API, the client will still break as no value can be provided with such an approach. Another approach from Leitner *et al.*. uses a proxy to route API calls to the requested version. VRESCo [22] uses plain string version names such as *latest*. In [134], Durieux *et al.* propose an HTTP proxy that uses five self-healing strategies to rewrite the buggy HTML and JavaScript code. *BikiniProxy* covers errors that occur when evolving API but it focuses on errors that exist within the UI of the Web application.

The main advantage of these techniques is that API consumers do not have to perform manual tasks to maintain the compatibility of the client with the API. However, the approaches are only able to deal with syntactical contracts, both behavioral and synchronization contract changes can not be supported, nor the addition of required parameters or deletion of information. Such approaches are thus limited in the types of evolutions that they can support. So they propose a partial solution to the problem and should be combined with other approaches.

1.6.4 Evolution policies

The use of evolution policies is the most pragmatic and used approach to the co-evolution issues faced in the current Web. These policies fall under two categories.

A first approach aims at keeping several versions of its Web API online [25]. This approach proposes to use mechanisms to declare the future deprecation as part of the API. For example, the “Chain of Adapters" introduced in [135] proposes to overlay one layer of adapters per version of the Web API. This technique allows for multiple versions to be deployed concurrently since older versions are left unchanged.

A second approach aims at defining the list of authorized evolution by ensuring either that this evolution has no impact on the client code or that an automatic transformation exists on the client code in order to support this evolution. We could cite several API definition guidelines that can be found online [23, 24].

On the one hand, these techniques have proven to be effective in reducing client breaking. However, **they considerably reduce the possibilities towards evolving REST APIs**. Indeed, all evolutions must be batched into versions, and because they require service consumers to perform manual code updates, service providers are expected to make evolutions bringing added value. Refactoring, and actually most modifications, are therefore discouraged.

This widely used technique is however not following the REST principles. In a talk for the Adobe Evolve conference, to the question "What is the best practice for versioning a REST API?" Roy Fielding answered "DON’T. Versioning is just a polite way to kill deployed applications" [51].

51. `https://www.slideshare.net/evolve_conference/201308-fielding-evolve/31`

Later in December 2014, he explained that versioning an API "either, (a) the version is eventually changed and all of the components written to the prior version need to be restarted, redeployed, or abandoned because they cannot adapt to the benefits of that newer system, or (b) the version is never changed and is just a permanent lead weight making every API call less efficient." [52]. He added that "this is precisely the problem that REST is trying to solve: how to evolve a system gracefully without the need to break or replace already deployed components.". Today we observe that this goal has not been reached with the REST architectural style. We propose a new solution to this challenge in chapter 4.

1.6.5 Automated client generation

Fully synthesizing the UI based on API description seems to be a relevant approach to cope with Web API evolution challenges. However, code generators proved to be complex to customize and evolve to integrate specific ergonomics projects [136]. Recent examples tend to confirm that.

The Swagger editor [53] is an example of such an approach. It is a generic user interface that builds nice-looking single-page API documentation based on an OpenApi description. The user interface can however not be personalized without modifying the generator.

As another example, Koren *et al.* propose a graphical tool to build a UI with already implemented Web components from an OpenAPI documentation [137]. Thus, designers can design and reuse high-quality components to build the website but the interface must be generated each time the API evolves. As a result, while custom Web components can be created to offer a good user interface, they are as tied to the input data model as the components designed without the generative approach. As a result, while high customization is possible, the maintenance effort in the event of API evolutions is not reduced. In addition, the use of custom code outside the Web components is very challenging. Thus, implementing and maintaining a custom logic to improve the performance of the Website or to manage complex business processes on the user interface is very challenging. It is probably more complex than on a website that is not generated due to the complexity added by the generator.

Generated websites are indeed complex to customize and evolve to integrate specific ergonomics projects.

1.6.6 Other approaches applied to the Web

No matter whether one chooses a generative or interpreter-based approach, an important question remains what information the Web API must provide to allow the client to adapt its API. Hervas *et al.*. [138] propose to use Semantic Web technologies to gain semantic information

52. Roy Fielding on Versioning, Hypermedia, and REST: `https://www.infoq.com/articles/roy-fielding-on-versioning/`
53. Swagger Editor - `https://editor.swagger.io`

about the user context, to adapt the UI. However, API evolutions are not considered in this work.

Another approach is LD-Reactor [131]. It builds a component-based UI from a configuration indicating which Linked Data-sets to retrieve. Thus, the UI is aware of data requirements. However, the use of SPARQL endpoints through RESTful API prevents clients to adapt to API evolution because graph, database-like, queries are sent to one single endpoint per API.

1.6.7 Synthesis and open challenges in the modern web

The co-evolution of systems has been an important research topic within the software maintenance community. However, applications to the Web are less numerous. The automatic program repair techniques that are effective with statically typed and compiled languages are not suitable to the Web. Indeed, the web exhibits a strong technological heterogeneity between the UI and the server and the common use of dynamically typed programming language to develop the UI parts. These properties make static analysis work more complex. Adapter-based techniques are transparent from the client perspective but are able to address too few kinds of API evolutions. The automatic generation of the client based on the API documentation gives great loose-coupling properties but is hard to configure. It is therefore too complex to build applications of good quality with such approaches. In addition, a UI can be costly to redeploy within an industrial project. Indeed, it was reported that redeployment takes from one week up to six months in 66% of the corporations [18]. This creates a barrier to automatic co-evolution, generation, and automatic program repair techniques that need to modify the UI's code when the API evolves, which requires application redeployment. In the end, the most effective approach is the use of evolution policies. They however considerably reduce the ability of the API to evolve, which is not a desired property in the web.

In this work, we argue that a fifth kind of approach is required to enable APIs to evolve in autonomy without breaking clients. Web API clients, especially user interfaces, should be implemented differently. They should be enabled to interpret the API contracts at runtime to continuously adapt their interactions with the APIs along with the user interface in order to maintain compatibility with the evolving API without requiring code changes. We name this property *evolvable-by-design*, a term that we define later in this work.

1.7 Design and Evolution of REST APIs

The two contributions of this thesis address the challenges of the break-free co-evolution of REST APIs and their clients and the selection of the right set of technologies to create a REST API of a given level of functionality.

To understand how REST API technologies can be selected, it is necessary to understand

the two main approaches to designing a REST API interface. We therefore review them in this section and discuss hypermedia controls, which are often discussed in the REST API design and documentation communities.

This section also presents the taxonomy of existing REST API evolutions. We base our work on the co-evolution of REST APIs on this taxonomy. Therefore, it is a necessary background for understanding the contribution. Furthermore, since the taxonomy is significantly influenced by how REST APIs are designed, we also present this topic in this section.

1.7.1 Designing the interface of a REST API

The *Uniform Interface* constraint of REST is probably the one that has been confusing practitioners the most for the past 20 years. It is "the central feature that distinguishes the REST architectural style from other network-based styles" but in his dissertation, Fielding does not provide an implementation example nor clear guidelines on how to achieve the *Uniform Interface* constraint.

While developers could easily identify how to achieve identification of resources, manipulation through representations, and self-descriptive messages, there are several practices for the design of the API interface itself and the use of Hypermedia As The Engine Of Application State (HATEOAS). Indeed, by studying 78GB of HTTP traffic collected by Italy's biggest Mobile Internet provider over one full day in October 2015, Rodriguez and colleagues observed that 95% of the REST API do not implement the *Hypermedia as the engine of application state* constraint [139].

The design of the interface that a REST API exposes generally falls under two categories: CRUD or CQRS. This separation actually reflects the trade-off that Fielding foresaw. The statement "uniform interface degrades efficiency, since information is transferred in a standardized form rather than one which is specific to an application's needs" ended up to be true. In addition to the degradation of efficiency, a uniform interface prevents the design of an expressive interface reflecting the application domain. To observe this, the reader should first understand the CRUD paradigm, which is the paradigm that corresponds the most to Fielding's description of a uniform interface. Leveraging the description of CRUD, we will then review how interfaces can offer more business meaning thanks to the CQRS pattern.

CRUD REST APIs. The first manner to design the interface of a REST API is to follow the CRUD paradigm. CRUD stands for Create Read Update Delete. They are the only four operations that can be available on a resource of a CRUD REST API. To read a resource, a user would use the GET method of the HTTP protocol. The creation is achieved with a POST, complete update with PUT, partial update with a PATCH, and the DELETE method is used for removal. One consequence of this design is that the update of a resource should not result in the update of another resource. This brings predictability to the system. In addition to

these operations, the OPTIONS method can be used to retrieve the available operations on the resource, the supported representations, and eventually the resource data schema.

The CRUD approach is the simplest and most standardized. It is also quick to implement. For these reasons, it is often the methodology that students learn in the first place. Another advantage is that the users can change the business process without the need to change the application. This is because the client holds the logic of the process. In the end, this methodology is particularly suited for the use cases where direct and complete remote access to raw data is necessary. Such kinds of applications are usually named data-centric. It is however difficult for the API server to discover the user's intent. Consequently, the management of complex business processes and access rights is challenging. Moreover, the REST API server (and code) is missing any knowledge about the application's domain, reducing its ability to implement advanced data consistency mechanisms. The CQRS pattern addresses this issue.

Co-evolution wise, the high standardization reduces the potentially changing elements to the data representations, the URL patterns, and the available methods only. Still, managing the co-evolution of these elements is challenging.

CQRS REST APIs. This pattern emphasizes Command and Query Responsibility Segregation (CQRS). It aims at enabling the design of process-centric applications, as opposed to data-centric applications. *Queries* return data but should not change data. *Commands* encapsulate business processes, they are used to modify data and should not return data; this is left to the query side. In practice, when within a CRUD approach a user would execute a *CreateUser* action, followed by *ActivateUser* and finally *SendUserCreatedEmail*, within the CQRS approach, the user would execute a *RegisterUser* command which would execute the previously mentioned three actions [54].

To apply the CQRS pattern to a REST API, the commands and queries must be considered addressable resources and should be executable with the relevant HTTP request methods according to their semantics. The *RegisterUser* command can for example be at *POST http://my-example-api.com/users/register*. The POST method denotes that it is not an idempotent operation and the URI naming tries to explicit the operation's purpose. Queries are easier to design, there are no differences with the CRUD pattern: resources are addressable and the GET method should be used to retrieve their representation.

With a business-centric API, the client should not need to implement the business processes logic anymore. However, not all business processes can be reduced to a single operation. For example, ordering a product from an online shop requires sending several commands. This approach thus significantly **reduces** the amount of business processes logic that the client embeds but can not remove it all.

Considering the interface, the REST APIs that follow this pattern offer more expressive,

54. The example is from: `https://herbertograca.com/2017/10/19/from-cqs-to-cqrs/`

easier to understand, interfaces thanks to the added domain semantics. It also clarifies the user's intentions. For these reasons, it is an architectural pattern that has been widely adopted in the enterprise.

The documentation and co-evolution of CQRS REST APIs present different challenges compared to CRUD REST APIs. Being less standardized and emphasizing the creation of more operations, all these operations must be documented and more kinds of elements must be co-evolved.

However the kind of REST API interface design, Fielding claims that applications must leverage the Hypermedia As The Engine Of Application State constraint to reduce coupling between REST API clients and servers in order to enable complete freedom towards the evolutivity of the applications, which include the servers. It is however not clear how to leverage this constraint, which we discuss in the following paragraphs.

1.7.2 Hypermedia As The Engine Of Application State in practice

We previously saw that one of the constraints of the REST architecture is that a REST API should include hypermedia controls (e.g. hyperlinks) in its responses so that the user can navigate the REST API without any prior knowledge of it, similarly to how people navigate websites.

It is often stated that users should be able to be communicated the base URL of the REST API, get the representation of the corresponding resource, and then use the returned hypermedia controls to navigate the API step by step (here, the user is the Web application, not the end-user). However, end-users do not always navigate on a Web application from its root, they usually bookmark the page they are interested in and open these pages directly. These pages therefore need to load one to many resources from the API to be able to render. Sticking to the aforementioned principle, the Web application should therefore intelligently browse the REST API to discover the path to the data. It would then retrieve the data and display the page.

As a reminder, the goal behind this mechanism is to provide loose coupling between the components in order to enable the independent evolution of both the clients and servers on the Web. Indeed, in an interview in December 2014, Roy Fielding said "Hypermedia is a constraint. As in, you either do it or you aren't doing REST. You can't have evolvability if clients have their controls baked into their design at deployment. Controls have to be learned on the fly. That's what hypermedia enables" [55]. This property is however not yet reached in the Web.

In practice, this mechanism raises many questions and remarks. As an example, we will discuss some of them but we will not review the comprehensive list of them. First, this is not efficient. Could not we directly access the resources instead of starting from the REST API's

55. Roy Fielding on Versioning, Hypermedia, and REST: `https://www.infoq.com/articles/roy-fielding-on-versioning/`

root, which imposes making several network calls? Second, how should the hypermedia controls be represented and how should the client select them?

Let's start with the selection problem. In practice, we design data structures with keywords. A *Company* is likely to have *employees*. So, a REST API managing the list of the employees of a company is likely to place an hypermedia control to the list of employees on its "home page". To access this list, the Web application would load the home page and follow the *employees* keyword to get the list of employees. As a result, the *employees* keyword must be known by the Web application. Maintenance-wise, this prevents the Web application from breaking if the URI to the employees is changed. Now, what if the REST API designer prefers to use *talents* instead of *employees*, to use a more rewarding word? (which many companies are doing to talk about their employees these days). In this case, the Web application would be broken because it uses the outdated keyword. One can argue that coupling is reduced, but there are still elements introducing a tight coupling: the keywords in place of the URIs. As a consequence, we may be looking for other techniques to select hypermedia controls, which practitioners have been doing.

Now, let's discuss the representation. If the REST API strictly follows the CRUD approach, it could use OPTIONS requests to discover how to build correct requests to the REST API and the other HTTP request methods to interact with the data. Coupling seems minimal. However, the use of the OPTIONS method is not a formal constraint and most developers do not even know the method. Moreover, considering an API that implements paging, filtering, and ordering mechanisms, how can the Web application discover the available parameters dynamically? Should it leverage the OPTIONS method or accompany the hypermedia controls with pieces of documentation? The questions become more complex when we start to address CQRS REST APIs. In this case, more operations are usually available. And each operation has its own input parameters. Complex query operations are also more frequent. Consequently, practitioners face more questions regarding the use of hypermedia controls and the documentation of the REST API.

As the last example, we consider that developers figured out how to represent and select hypermedia controls. The next question that they face is: how to leverage them to have minimal coupling with the REST API? what kind of information do they give us to prevent maintenance when the REST API is evolved? It is still unclear how these controls should be used to communicate access rights or represent business rules. On the one hand, one could argue that all hypermedia controls related to a resource should be communicated with the resource, however the context. On the other hand, hypermedia controls may be communicated only if the user is allowed to invoke them, and if the target operation is available in the current state of the resource. Doing so however prevents the user from knowing all possible state transitions. In addition to this, it is also unclear whether the hypermedia controls should only communicate the state transitions of the represented resource or if they should also be used to suggest navigation

paths.

Open Challenges. In the previous discussion, we saw that the representation and selection of hypermedia controls can be done in many different ways. The diversity of the open-source and academic propositions for the description of hypermedia controls confirms this. Such diversity tends to confuse practitioners who must understand the differences between them to choose the right technology for their needs. A solution to this problem is proposed in this thesis (see 3). In addition, it is unsure if hypermedia controls are sufficient to reduce the client-server coupling to the point where the REST API server can evolve independently without breaking clients. As one example, using keywords to select hypermedia controls proved to introduce coupling. We therefore argue that HATEOAS is not sufficient to address all kinds of REST API evolutions and that more mechanisms are necessary. We explore this question in the second contribution of this work (see 4).

1.7.3 How REST APIs evolve?

We saw several times that REST APIs can be evolved at run-time, sometimes breaking their clients. In the following chapters of this thesis, a new approach to the co-evolution of REST APIs and their clients is presented. To understand this work, it is necessary to know how REST APIs can actually be evolved. So, in this section, we first review the kinds of REST API evolution, from a technical perspective. Second, we will discuss for what reasons practitioners decide to evolve their REST APIs.

The 22 kinds of REST API evolution

As observed by Sohan *et al.*, who studied public, widely-used, REST APIs, they tend to evolve frequently [140], particularly the young APIs that companies keep internally. Yet, little research studied how REST APIs evolve. A first work [141] identified 16 evolutions in 2013 and was extended in 2015 [140] with 6 other patterns. Another work was done by Wang *et al.* in 2014 [142]. Table 1.1 lists these 22 evolutions where the breaking (✗) and non-breaking (✓) evolutions are differentiated. Non-breaking evolutions relate to behavioral elements. While they don't break at compile-time, they may create inconsistent behaviors. For example, 14 (*"Change default value of parameter (non-breaking)"*) can lead the ordering process of an e-commerce website to become impossible to complete in certain cases when the user inputs unaccepted values.

Why do practitioners evolve their REST APIs?

The reasons that motivate developers to evolve their REST APIs are very similar to the reasons motivating developers to evolve any kind of system, especially the systems that have

Table 1.1 – List of evolutions of RESTful APIs

From existing studies	
1	Add or Remove Parameter (✗ breaking)
2	Change Type of Parameter (✗ breaking)
3	Change Type of Return Value (✗ breaking)
4	Delete Method (✗ breaking)
5	Rename Method (✗ breaking)
6	Rename Parameter (✗ breaking)
7	Change Format of Parameter (✗ breaking)
8	Change Format of Return Value (✗ breaking)
9	Change XML Tag (✗ breaking)
10	Combine Methods (✗ breaking)
11	Split Method (✗ breaking)
12	Expose Data (✗ breaking)
13	Unsupport Request Method (✗ breaking)
14	Change Default Value of Parameter (✓ non-breaking)
15	Change Upper Bound of Parameter (✓ non-breaking)
16	Restrict Access to API (✓ non-breaking)
17	Move API elements (✗ breaking)
18	Rename API elements (✗ breaking)
19	Behavior change (✓ non-breaking)
20	Post condition change (✓ non-breaking)
21	HTTP header change (✗ breaking)
22	Error condition change (✓ non-breaking)

clients.

The first reason that naturally comes to mind is the addition of new features. Within a REST API, it can result in the addition of properties to resources to provide more information, of methods, or even of new resources.

Second, developers evolve REST APIs to reflect changes in the exposed domain processes. We saw that CQRS REST APIs are used to offer process-driven services. As with any process, they are influenced by regulations, users' needs, organizational constraints, and so on. For example, a regulation enforcing website providers to collect the users' id card numbers when subscribing would force many REST APIs to add this field to their subscription operation. Similarly, enforcing the use of passwords that are at least 8 characters long would also force providers to update the API. These kinds of changes are somehow technical as they do not change the feature nor add value to the API. We therefore argue that they should be automatically supported by the clients and that evolving the client manually does not add value.

Developers may also refactor the interface. Refactoring can modify any kind of element of the REST API. They can thus impact the features offered by the API, or not. There can be many reasons motivating a refactoring.

Let's first consider refactoring that should not impact features. As a first example, developers may have created the API with limited knowledge of the domain. Over time they gain a better understanding of it and finally judge that the interface can be improved to better match the domain. In this case, keywords, resource, and operation names are likely to be changed along with the data structures. However, the features are less likely to be changed. This kind of refactoring could ease future developments by aligning the words used in the interface with those that the developers use to work, and also ease consumption by clients who are familiar with the domain. Another reason motivating a refactoring can be the willingness to follow new trends. With time, clients' expectations may change. Indeed, in recent years we observe that (buzz) words can be powerful to drive adoption. Changing the data interchange format or data structure may also be effective. Also, developers are likely to introduce naming inconsistencies in their API over time. For example, sometimes they write *fullName* and sometimes *fullname*, one day they decide to standardize. They can also make design mistakes such as using the wrong HTTP methods, or decide to follow new design recommendations. And finally, who never left work after a long day of development and realized a few hours later that it is absolutely necessary to refactor everything the next morning? And who never had to maintain another team's code, and "observed" that it MUST be refactored? (for obviously objective reasons). The point is that there are also subjective reasons motivating developers to refactor code. In the end, because such kinds of refactoring do not change the features, we argue that they should be automatically supported by the clients.

Other refactorings may however modify the features offered by the REST API. For example, the transition from a CRUD to CQRS REST API would transform the interface in-depth and

therefore require to rethink the usage of the API on the client-side. As another example, an operation can be split into several operations to offer a different path to go through a process. In such cases, the semantics of the interface is significantly changed. We argue that designers should study how to evolve the client when such changes are done. There may however be some ways to keep compatibility with the evolving API during this study, which we study in the second contribution of this thesis (see chapter 4).

1.8 Synthesis

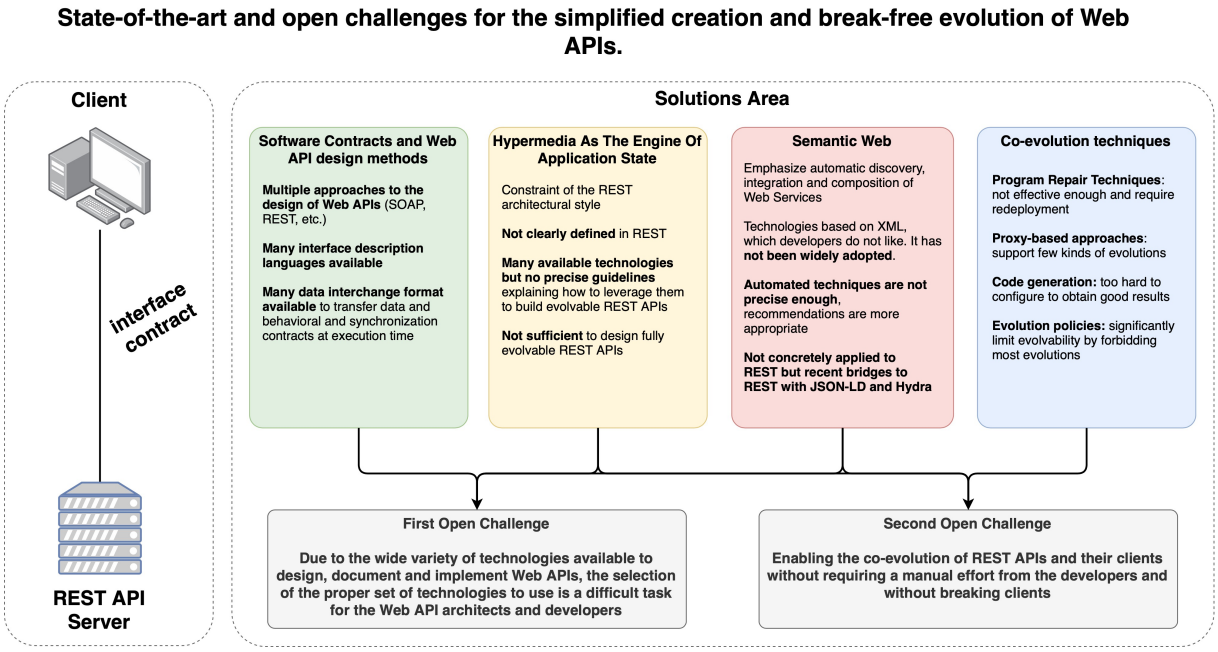


Figure 1.12 – Overview of the problems, research and limitations presented in Chapter 1

A visual synthesis of the state-of-the-art and open challenges for the simplified creation and break-free evolution of Web APIs is given in Figure 1.12. The figure highlights the main peculiarities of the research topic presented in this section along with the open challenges that we identified.

In this section we saw how the World Wide Web evolved, pushed by the increasing speed of Internet connections and the increasing number of users. From a place where webmasters pushed content, the Web rapidly became a highly collaborative place where all users publish and consume content from numerous services. Then, smartphones arrived and people started to demand access to the same content on all their devices. Developers thus started to split applications into three parts: a REST API, a Web user interface, and a mobile application. In addition, users wanted the sharing of content between services to be easy. This resulted in a

modern Web that is highly distributed, where systems from all vendors communicate with each other via REST APIs. Then to ensure the high availability of the services on the Web, developers adhere to rigid interface contracts and follow strong evolution policies. Otherwise, applications can break.

As a result, while the Web has succeeded in becoming the envisioned global information system, that is highly available and very reliable, the promise of a loosely-coupled architecture is not met. Contracts are not interpreted by the API clients but are mainly hard-coded. Most of their evolutions can therefore break clients, which is unacceptable in the modern web.

Solutions to these problems have not yet been proposed. On the one hand, we saw that software contracts are well categorized and that many research address their description or automatic generation. However, with millions to billions of systems in the Web, developers' needs can vary significantly, preventing the design of a one-size-fits-all solution. Moreover, as represented in the green box of Figure 1.12, research is being carried out with varying objectives, resulting in many available interface description languages, serialization format, and implementation frameworks. So, for a Web API creator, the selection of the relevant set of technologies to use is a difficult and error-prone task. Erroneous decisions regarding these technologies can significantly reduce the productivity of the API developers and lead to a decreased design quality. We thus identify an open challenge towards making the process of selecting the best set of technologies to design, document, and implement a Web API that meets the target functionality level and design quality.

On the other hand, consequent energy has been put into the exploration of techniques to automate otherwise extensive manual tasks, leveraging the technologies of the semantic web. Such techniques could have turned the Web into a loosely-coupled place. However, as represented in the red box of Figure 1.12, semantic description, automatic service discovery, integration, and composition techniques have not reached wide adoption due to their perceived complexity and insufficient performance to be used as fully autonomous systems. Recent efforts such as JSON-LD proposed a JSON format that can embed semantic descriptors in order to bridge the gap between Semantic Web and REST API technologies. Now, an example that demonstrates how semantic Web technologies can be used to simplify developers' life is missing. If adoption increases, previous research is more likely to be explored in more detail and then adapted for use with the modern web. Because one of the major pain points in the Web is the co-evolution of the REST APIs and their clients, we argue that such an example can come from exploring this topic.

We therefore explored the co-evolution domain. A summary of these approaches is given in the blue box of Figure 1.12. We saw that early approaches proposed to automatically detect evolutions and adapt the code following automatic program repair techniques, but applying these approaches in the highly heterogeneous Web is not realistic. Adapter techniques hide

the evolution from the client perspective which results in the support of a few sorts of API evolutions. Additional approaches are therefore necessary. The automatic generation of User Interface components is appealing but configuration generators end up being too complex to provide convincing results. Finally, the most effective and popular approach for co-evolution in the modern Web is to keep several versions of an API online and to limit the possibilities to evolve an API by defining evolution policies. However, this approach does not improve coupling but avoids facing defects and therefore considerably reduces the freedom of the developers to evolve their services. This is however not a desired property in the Web. We therefore identify a second major challenge regarding Web APIs. API providers should be able to evolve their APIs freely.

Chapter 2

Thesis overview

Overall, in this thesis, we tend to address the major scientific challenges faced by REST API providers throughout the entire life cycle of their REST APIs. Figure 2.1 puts the contributions of this thesis in perspective with the life cycle of REST APIs.

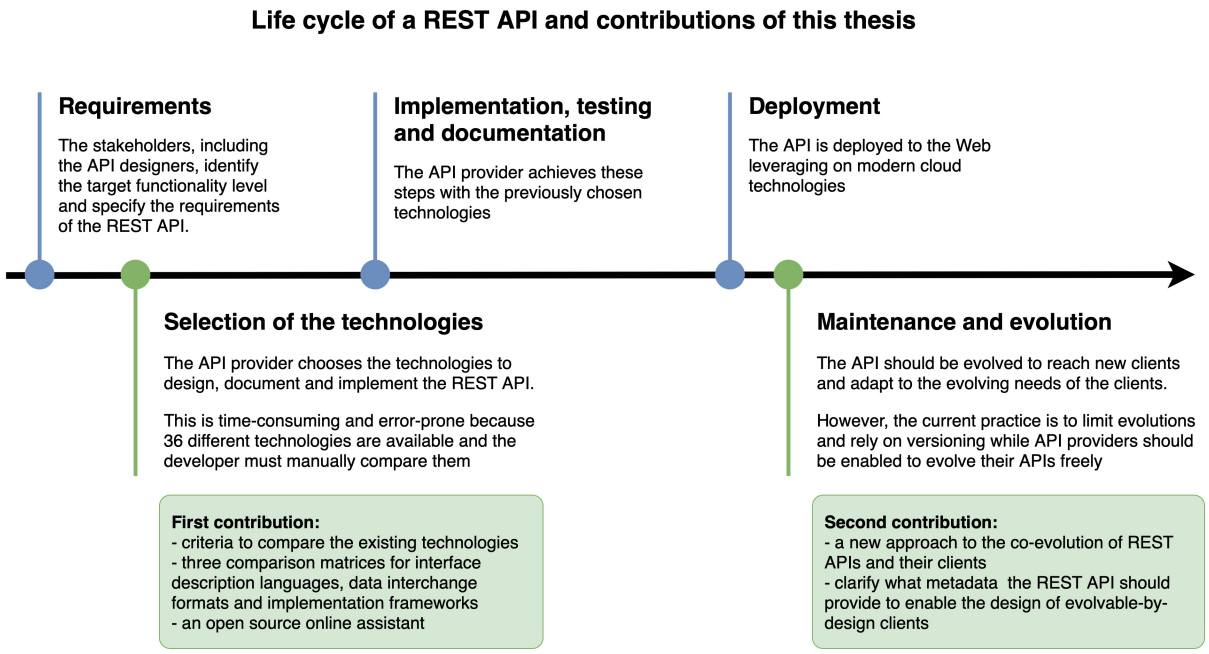


Figure 2.1 – Overview of the contributions of this thesis, put in perspective with the life cycle of a REST API

As the figure depicts, the first task is to define and specify the requirements of a REST API. This is a task that does not present major challenges and can usually be achieved in a few hours. The most challenging part of this task is to find an agreement between all stakeholders, which is out of the scope of computer science research. Next, the technologies must be chosen. In chapter 1, we saw that many technologies are available to design, document, and implement REST APIs. In addition, practitioners seem to be satisfied with these technologies as no major new technology appeared in the past four years. The following step is the implementation, testing, and documentation of the API. To achieve this, many helpful technologies are well

identified and many learning resources are available. Thus, practitioners do not face significant challenges. Thanks to modern cloud technologies, the deployment of a REST API is an easy task. The maintenance and evolution of a REST API are however challenging. Indeed, we saw in chapter 1 that existing techniques are not able to enable the break-free co-evolution of REST APIs and their clients. Indeed, the current practice is to limit evolution.

We therefore identified two major challenges preventing the easy creation and break-free evolution of REST APIs. First, we saw that due to the wide variety of technologies available to design, document, and implement REST APIs, the selection of the proper set of technologies to use for a given project is a difficult task for the REST API architects and developers. Second, when the API is up and running, evolving it is challenging because of the important risk to break clients.

As a first contribution of this thesis, we analyze the differences and similarities between the 36 technologies that we identified to design, document, and implement REST APIs. Based on this analysis, we proposed three comparison matrices that help practitioners select the set of technologies to use for their needs. We then demonstrate how the matrices can be leveraged on a realistic example. To facilitate the selection, we have developed a Web assistant, named Morice.

As a second contribution of this thesis, we tackle the evolution issues by proposing a novel approach to the co-evolution of REST APIs and their clients. As a first step, we challenge the existing taxonomy of REST API evolutions by surveying professional Web developers. We thus complete the taxonomy with 7 new kinds of evolutions. Then, to design our approach, we leverage semantically enriched contracts descriptions and dynamic contract resolution. The general idea is that instead of hiding the evolutions from the client perspective, or instead of modifying the code, which implies a new deployment, REST API clients should be designed *evolvable-by-design*, a term that we introduce. Concretely, REST API clients should become able to discover and interpret the contracts of the REST API at runtime. By doing so, a client can produce correct REST API interactions at any moment in time, even when the API is evolved. Such an adaptation occurs at runtime and therefore does not require the modification of the client code. Moreover, as the client user interfaces are designed to deliver a precise set of features to the end-user, with a carefully designed user experience, the interface should not be generated but slightly adaptive, where the designer permits this. We validate the feasibility of our approach through a quantitative analysis in the form of a case study that mimics Jira and implements 110 API evolutions and through a qualitative analysis where we implement our approach on 5 open-source web applications that implement 20 evolutions of 10 types. As a result, the approach proved to enable the design of Web UIs that automatically adapt to 27 out of the 29 known types of REST API evolution.

These two contributions tend to address the major scientific challenges faced by REST API providers throughout the entire life cycle of their REST APIs. They are presented in the following

two chapters. At the end of this document, we present opportunities for future research aiming at further evaluating the approach presented as a second contribution and at leveraging the opportunity of having break-free clients in the Web to automate more things in the Web.

Chapter 3

Comparison Matrices of REST APIs Technologies

Foreword

This chapter presents the first contribution of this thesis. It tackles the first challenge presented earlier: the difficult selection of the proper set of technologies to design, document, and implement a REST API of a target maturity level. In this chapter, a set of criteria is proposed to compare these technologies. Next, the criteria are leveraged for the design of three comparison matrices: one for each category of technology. Then, an example demonstrates how the matrices can be used for the selection of the technologies. Finally, to further ease the selection, an online and open-source assistant is proposed and its usage is presented.

This chapter presents the first contribution of this thesis [29]. It has been published at the International Conference on Web Engineering in 2019.

3.1 Introduction

In chapter 1 we saw that RESTful APIs [28] have become the de-facto standard to design services on the Web. However, 95% of them are not RESTful [139] as they claim. We can explain this in two different manners. On the one hand, developers do not know the REST architecture well enough to be aware that they are not RESTful. We can also argue that not all projects need full compliance with the REST architecture. Partial compliance may actually be sufficient for some systems, as each one has its own constraints. On the other hand, the partial compliance can be explained by the fact that it is not clear how to leverage the *Hypermedia As The Engine Of Application State* constraint in practice.

In addition, a new trend has recently emerged to create RESTful APIs that carry their own semantics, they are called Semantic RESTful APIs [2]. It is a vision that proposes to make

REST APIs compatible with the Semantic Web [37] and Linked Data [82]. As discussed in 1.5, such compatibility can offer great benefits, such as loose-coupling, automated API discovery, integration, composition [143] and very powerful querying.

There are thus many recommendations within the REST and Semantic REST architectures on how to design Web APIs. However, not all systems need all the properties of these architectures. Architects therefore have to select the recommendations that are relevant or not for the system that they design.

Moreover, we saw in chapter 1 that many technologies have been proposed to describe web API contracts, either with or without semantics. We counted 36 technologies of this kind. Consequently, architects and developers are facing the challenge of selecting the right technologies for the design and implementation of their services. Typically, they have to select the right interface description language (IDL), interchange format, and framework to ease the development of such services. However, the specific criteria and properties to be taken into account are not explicit when choosing an IDL, an interchange format, and a framework. This considerably increases the complexity for the architect to choose the appropriate technology.

The industrial needs are growing for proper tools to support trade-off decisions of the architect; a tool that would help him/her to understand the consequences of a design decision, i.e. the characteristics and limitations of each approach.

In this work, we propose to fill this gap by providing three decision matrices that help architects to choose the technologies that will best fit their needs. The main contributions of this work are:

— three comparison matrices of interchange formats, interface description languages and frameworks that help to choose the appropriate set of technologies to build Semantic RESTful APIs;

— key features that are missing from state-of-the-art technologies to assist and make the creation of Semantic RESTful APIs more beneficial.

In this chapter, section 3.2 provides the required background on Semantic REST APIs and the reference maturity model to choose the functionality level of an API along, with its limitation. The two following sections describe our comparison matrices and an illustration that highlights the benefits of our proposition. Finally, section 3.5 discusses the role of the existing frameworks to build Semantic REST APIs.

3.2 How to select and evaluate an API functionality level?

This section describes the main concepts related to the design and implementation of Semantic RESTful APIs and the process of selecting an API functionality level. It also discusses the

3.2.1 Semantic RESTful services

Combining REST with Semantic Web and Linked Data is a promising path since it could enable the description of APIs that can change without breaking client applications (see chapter 4). These APIs advertise their available state transitions, therefore enabling automatic composition to create high-level services [144]. Smart software agents can then automatically discover the suite of operations to realize complex workflows and even make APIs compatible with voice assistants. This is achieved by semantically enriching the data and operations of REST systems with Semantic Web ontology technologies and by linking resources to other resources.

3.2.2 Selecting an API functionality level

Today, Web systems offer a wide range of functionalities. For example, they may offer multiple media types or a single one, comply with the HTTP protocol or use it as a transfer protocol, or even semantically describe their resources. This diversity can make the process of comparing and selecting the minimum set of features to be implemented very time-consuming. Maturity models have been proposed as a solution to this problem [145, 2].

In companies, architects and developers use them to decide features that must be supported by their APIs. In general, a maturity model is a scale that represents the compliance of a technology with a given architecture. To reach a level, a technology must meet each constraint of the targeted level and the previous levels. Currently, the de-facto standard in the industry is the Richardson Maturity Model [146], which targets building REST APIs. However, we recommend using the WS3 maturity model [2] as it combines the models proposed by Richardson, SoHA [147], and extends them with semantic and documentation constraints.

The WS3 maturity model In [2], authors describe the WS3 maturity model for classifying Semantic REST Web APIs. It classifies APIs along three independent dimensions: *design*, *profile* and *semantic*, as shown in Fig. 3.1.

The **design dimension** represents the different modeling strategies adopted for designing the technical access to a Web API through four levels: (i) RPC, (ii) resources have dedicated URI and the API is stateless, (iii) operations on a resource are mapped to HTTP methods in compliance with the protocol, and (iv) the smallest data unit that can be handled by operations is the resource.

The **profile dimension** reflects the quality of documentation that can be interpreted by software agents through two levels. The first level: *interaction profile*, requires the description of the syntactical contracts, i.e. all available HTTP operations and how to trigger them. The second level: the *domain profile*, requires the description of domain-specific details such as the order of operation execution, pre-conditions and post-conditions, business constraints, etc. This last level corresponds to the behavioral and synchronization contracts.

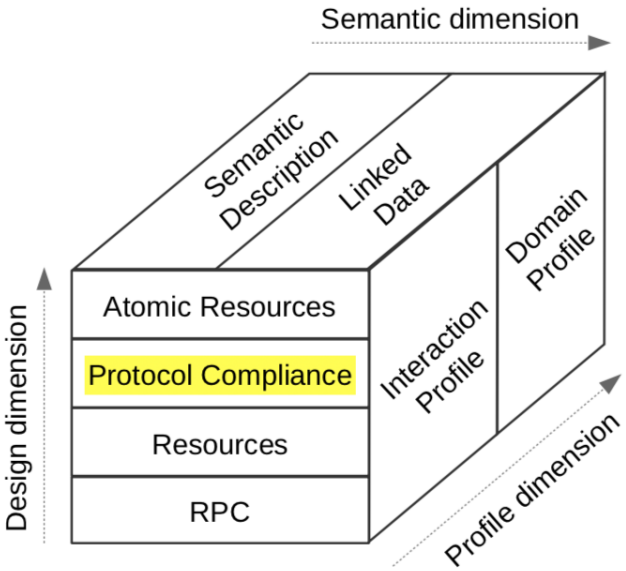


Figure 3.1 – WS3 Maturity Model (from [2])

The **semantic dimension** represents the use of semantic technologies through two levels. To reach the *Semantic Description level*, an API must semantically describe properties and operations of resources. The next level: *Linked Data*, is reached when the API semantically describes relationships between resources.

Usage In their paper [2], Salvadori *et al.* propose to rate systems along each dimension independently, with a score going from 0 to the number of levels in the dimension. For example, a non-documented API with no semantic support that reaches level 3 of the Richardson Maturity Model will be rated D3-S0-P0 [1]. As another example, a system that supports HATEOAS and provides swagger-like documentation along with the data is rated D3-S0-P2 [2].

3.2.3 Discussion on the WS3 maturity level

At Fabernovel, we experienced two limitations to the applicability of the maturity model to a wider audience. These limitations are related to the *Atomic Resources level* and the granularity of the WS3 levels.

According to WS3, the *Atomic Resources* constraint requires that the resource is the smallest data unit handled by operations. It corresponds to the design of CRUD REST APIs. As discussed in 1.7, respecting this constraint may introduce negative properties in the API. Let

1. D3-S0-P0: Atomic Resources Design, no Semantic Description, no Profile description
2. D3-S0-P2: Atomic Resources Design, no Semantic Description, Domain Profile

us consider an API handling insurance contracts that offers read and update operations on the postal address, email address, and insurance manager. Two solutions can be considered to respect the *Atomic Resources* constraint. The first solution is to create one resource, where every property can be modified at once, which increases the risk of concurrent modification. With this solution, the API would have two operations. The second solution is to create one resource for each concept: contract, email address, postal address, and the manager. The API would have eight operations. This solution increases dramatically the number of operations which complexifies the documentation and maintainability. Another solution would be to create one resource with four operations: (i) read, (ii) update email, (iii) update postal address, and (iv) update manager. This solution lowers the concurrency risk while maintaining a reasonable complexity and offering meaningful operation names. Unfortunately, this solution breaks the *Atomic Resources constraint*. We therefore argue that respecting this last constraint may not always lead to better API quality.

The second limitation relates to the granularity of the maturity levels. Indeed, each level implies more than one feature. This granularity allows for a coarse-grained categorization of systems. However, to precisely differentiate systems based on the features they implement, a deeper study is needed. Given two systems that reach P1, which means they describe all available HTTP operations and how to trigger them, one might also describe its authentication process and errors while not the other one. And yet they reach the same maturity level. We therefore argue for a finer grain categorization of APIs.

3.3 Comparison Matrices

We propose three detailed matrices which address the limits of WS3 identified in the previous section. The proposed matrices enable the comparison of technologies along a set of precise criteria to highlight their differences. These matrices extend the WS3 levels by adding new criteria which are used in practice (see section 3.3.1) and are not linked to any WS3 levels.

3.3.1 Insights from developers and architects

We interviewed 14 developers and architects from FABERNOVEL and clients on their experience with Semantic REST technologies. Raw results and the analysis are available online [3]. Our key findings are:

— *Selecting the technology:* 10 respondents have already built Semantic REST APIs: **30%** spent more than two weeks selecting the technologies; **80%** reported that the most difficult task was to understand the feature provided by each technology.

3. `https://github.com/AntoineCheron/comparison-matrices-semantic-rest-api-techno`

— *Interchange Formats:* **6 out of 7** did not find a technology providing all required features (most often the missing features were the description of HTTP operations with their data model (3/8) and the Linked Data (2/8)).

— *Interface description languages:* All respondents said that none of them provide all required features (60% said they lack the ability to describe links to other resources and business constraints; and 20% of them would like to model the resources as finite state machines (FSM)).

— *Frameworks:* 6 out of 7 reported that no framework offered the required feature. The missing features are related to the auto-documentation of the API, the automatic generation of links, and a mechanism to model resources as FSM.

— *Technology score:* The median value of the score is 2/5.

These results emphasize the difficulties in selecting technologies associated with Semantic REST APIs. They also highlight that these technologies are not yet mature and give a rough idea of the missing features.

3.3.2 Comparison Matrices Design Method

The design of our comparison matrices follows a 5-step sequential process: *(i)* **search** for candidate technologies, *(ii)* **select** candidate technologies, *(iii)* **read** carefully each candidate technology, *(iv)* **elaborate fine grain criteria** to characterize and differentiate technologies, *(v)* **verify** that the elaborated criteria highlighted the differences between technologies. We looped on step *(iv)* and *(v)* to avoid duplicating criteria or hiding important details.

The research of candidate technologies (step i) was done by:

1. Searching Google and Google Scholar for Semantic REST Technologies using combinations of keywords from the set: [“web”, “semantic”, “restful”, “rest”, “service”, “API”, “interface”, “description”, “documentation”, “language”, “modeling”, “hypermedia”, “document”, “format”, “RDF”, “data-interchange”, “linked data”, “hateoas”, “rest api”, “framework”];
2. Searching Google Scholar for tools automating tasks from services description, using keywords: “matchmakers”, “service composition”, “service discovery”, “rest service analysis”, “automated mashups”, we then selected papers and technologies from their references and the papers that cite those we selected;
3. Searching the proceedings of ICWE and WS-REST.

We selected 81 papers, standards, articles, and web pages (step ii) based on abstract or introduction. We selected documents that were specifications of interface description languages or models, frameworks supporting HATEOAS features, interchange formats that support RDF

or HATEOAS features, comparisons between these technologies or tools leveraging them. We considered frameworks available as programming libraries that help implement HTTP APIs in any programming language. We opened our research to technologies from the 1990s to today and retained those that are still available today.

Then, we read the specification of each chosen technology (step iii) and elaborated classification criteria (step iv). We included those of the H Factor [4] which *is a measurement of the level of hypermedia support and sophistication of a media-type*. Others were carefully designed to highlight differences between technologies, based on the core design of the technologies, the features they provide, and the details of the WS3 maturity model. All the material is available online [5].

As a final step (step v), we read the specifications again to verify results and validate that the selected criteria highlighted differences and commonalities well.

Popularity criteria We defined a popularity criteria to provide a rough idea of the community support and the likelihood of the technology to last in time. It respects the following rules: **0 -** Not enough to reach 1; **1 -** More than 100 questions on Stack Overflow AND (2500+ NPM weekly downloads OR 100+ maven usages); **2 -** More than 400 questions on Stack Overflow AND (500.000+ total downloads OR 15.000+ NPM weekly downloads OR 500+ maven usages).

3.3.3 Interface Description Languages

Interface Description Languages (IDLs) provide a vocabulary to document domain, functional and non-functional aspects of an API. We identified 16 candidates that are classified according to 31 criteria in Fig 3.2. Among them, 4 are meta-models from conference papers [69, 148, 149, 150]. The 11 others are open-source projects or W3C recommendations.

In [148] authors present a tool to sketch CRUD or Hypermedia APIs. On the latter mode, users sketch the application using state machines and then obtain a description in the HAL or Collection+JSON format. [149] models each resource type as a finite-state-machine with deterministic transitions and conditions to inform about the availability of transitions. However, they are not modeled in more detail, which make them not machine-interpretable. In [150], authors propose to model systems as non-deterministic state machines. This method thus makes software agents unable to discover the set of messages to exchange in order to make an operation available. Haupt et al. [69] propose a multi-layered model that separates the domain model from the URI model. However, resources have a fixed model, which prevents them from having one data model per state.

It is important to note that when **IDLs and interchange formats** are both **compatible**

4. `http://amundsen.com/hypermedia/hfactor/`
5. `https://github.com/AntoineCheron/comparison-matrices-semantic-rest-api-techno`

with RDF, they can be combined to form a file format usable as data-interchange format and IDL. This has great benefits to lower the overall complexity and increase the evolvability of the system.

Synthesis

First, the matrix highlights the fact that most technologies help with building mature systems on the *design* dimension and *interaction profile* level of the *profile* dimension, D3-P1 following the WS3 categories. On the other hand on the *semantic* dimension, we notice that 5/16 technologies support the use of RDF vocabulary, which allows building Linked Data APIs. As a reminder, this is required to reach full Semantic REST compliance. Moreover, by supporting the use of RDF vocabulary, IDLs can be enriched to reach a higher level of maturity.

Among the technologies, four can be distinguished by the number of criteria they meet: Hydra (18), RADL (18), OpenAPI (17), and RESTdesc (17). OpenAPI is the only one that has no support for RDF. Thus, it helps in building systems up to D3-P2-S0 on the WS3 scale. On the other hand, Hydra, RADL, and RESTdesc support the use of RDF vocabulary, which makes these technologies better suited to build systems that are mature on the semantic dimension.

Towards HATEOAS APIs From the matrix, we notice that most technologies target the documentation of the API in a single, non-splittable file. Hence, they are not suited to provide hypermedia controls at runtime.

On the other hand, only one approach, [149], supports the description of the conditions that determine the availability of a link, and none makes this meta-data machine-interpretable. This makes software agents unable to find a way to make an operation available when it is not.

Towards better-documented APIs Only four technologies support the description of business constraints which lowers coupling and improves user experience, e.g., with the automatic generation of forms with client-side validation.

Finally, we note that most scientific publications recommend the modeling of RESTful systems with state-machines whereas open-sourced or W3C IDL authors don't consider this design method. And yet, the use of deterministic state-machines eases the determination of the available operations of a resource.

3.3.4 Data-interchange formats

These formats provide a data structure, a vocabulary, and a layout to represent a resource and its meta-data at runtime. When the API does not need to send meta-data, JSON and XML are the two widely used formats in the industry.

On the other side, when the system to be built has to support a hypermedia interchange format, none is considered as a standard today. We selected 11 candidate technologies, which are classified in Fig. 3.3 according to 24 criteria. JSON is included for comparison purposes.

Figure 3.2 – Interface Description Languages Comparison Matrix

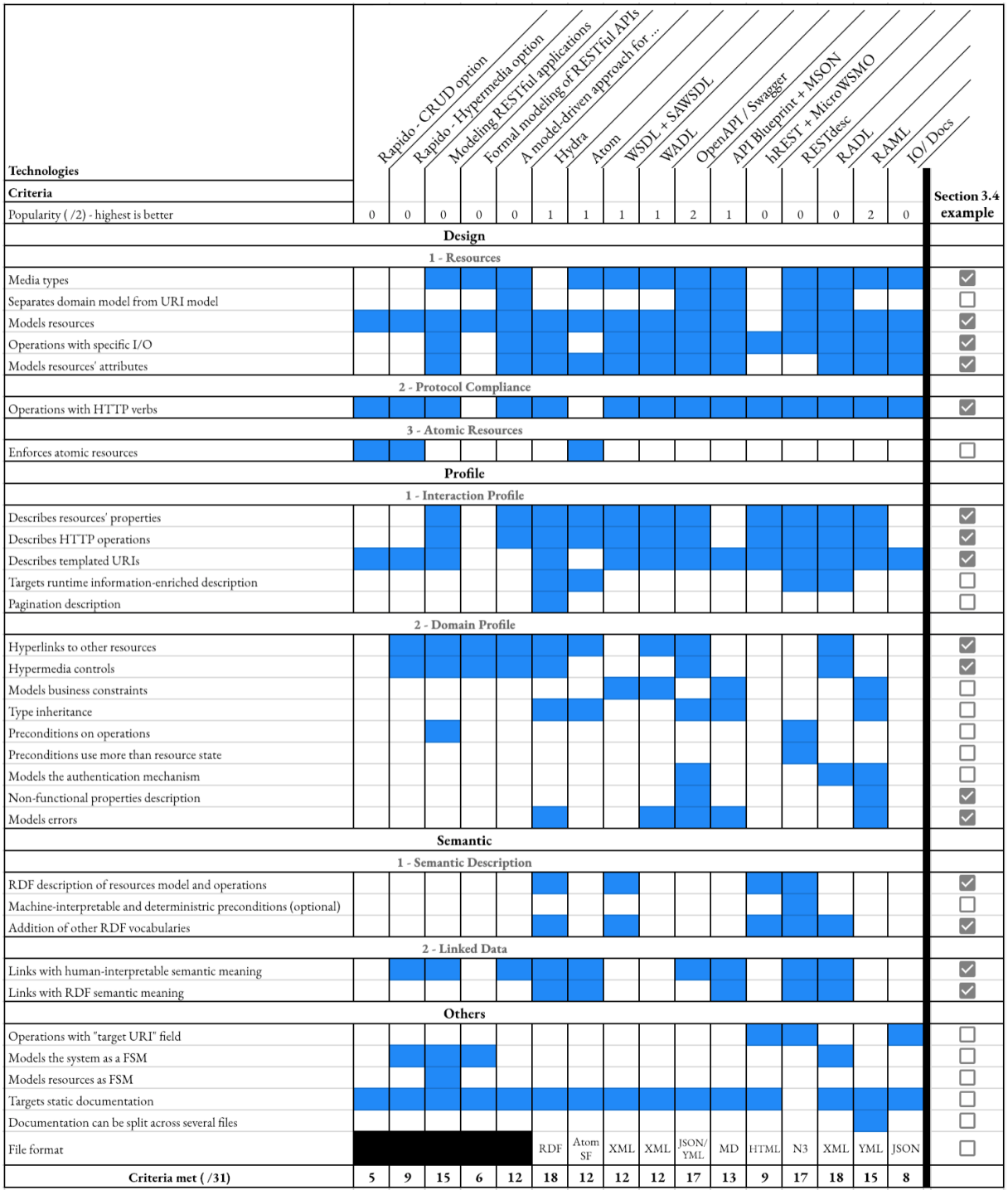


Figure 3.3 – Data-interchange Formats Comparison Matrix

Technologies	JSON	HAL	Collection+JSON	Siren	Uber	Mason	Json:Api	Atom	Turtle	RDF XML	OData Json format	JSON-LD	JSON-LD + Hydra	JSON-LD + Hydra + SHACL
Criteria														**Section 3.4 example**
Popularity (/2) - highest is better	**2**	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**1**	**2**	**0**	**1**	**1**	**0**	
Profile														
1 - Interaction Profile														
Describes templated URIs		■	■	■	■	■					■		■	■	☑
Describes media types for Read requests		■		■	■	■		■			■			■	☐
Describes media types for Update requests				■	■	■								■	☐
Describes operations' HTTP verbs				■	■	■							■	■	☑
Describes resources' properties			■	■		■							■	■	☑
Describes HTTP operations				■		■							■	■	☑
Pagination description							■				■		■	■	☐
2 - Domain Profile														
Hyperlinks to other resources		■	■	■	■	■	■	■	■	■	■		■	■	☑
Hypermedia controls				■	■	■							■	■	☑
Models business constraints						■	■							■	☐
Non-functional properties description						■	■								☑
Models errors			■		■	■	■				■		■	■	☑
Semantic														
1 - Semantic Description														
Addition of other RDF vocabularies									■	■		■	■	■	☑
2 - Linked Data														
Links with human-interpretable semantic meaning		■	■	■	■	■	■	■			■	■	■	■	☑
Links with RDF semantic meaning		■	■			■						■	■	■	☑
Others														
Human Readability														
JSON-based format	■	■	■	■	■	■	■				■	■	■	■	☑
Same structure as original JSON	■	■				■						■	■	■	☑
Designed for human readability	■	■	■	■	■	■	■		■	■	■				☐
Support for Curies		■				■						■	■	■	☐
Entity-centric document	■	■	■	■	■	■	■	■			■	■	■	■	☑
Prefixed keywords		■				■			■			■	■	■	☐
Support several meta-data levels						■					■				☐
Machine reasoning														
Designed for machine readability								■			■			■	☐
Operations with "target URI" field				■		■									☐
Criteria met (/24)	**4**	**11**	**9**	**13**	**11**	**20**	**8**	**5**	**4**	**3**	**11**	**8**	**16**	**20**	

Synthesis

First, from this matrix, we notice that formats can be differentiated based on their compatibility with RDF. Indeed, RDF formats (Turtle, RDF XML, and JSON-LD) propose very few features by default because they can be enriched with RDF vocabularies. To depict what is achievable by combining vocabularies with an RDF format, we selected two vocabularies:

Hydra and SHACL, an RDF schema validation vocabulary, that we combined with JSON-LD and evaluated them. As a result, they match 12 more criteria than JSON-LD alone. From this, we infer that combining RDF formats with vocabularies allows building mature Semantic REST systems. However, this requires additional effort to find relevant vocabularies. On the other hand, non-RDF formats help build systems that can be mature on the *profile* dimension but not on the *semantic* dimension.

Furthermore, the matrix shows that no format supports the description of constraints despite the fact that it can be leveraged to reduce coupling and improve the user experience.

Finally, it highlights that no format advertises the state of the resource even though most scientific approaches we found describe REST APIs as state-machines.

3.3.5 Implementation Frameworks

Implementation frameworks are software libraries that guide developers through the implementation of Web APIs. We limit the comparison to frameworks that claim to support HATEOAS. We identified six frameworks that do so. Frameworks to build Semantic Web Services are excluded because their triple-centric approach differs too much from REST.

Among the selected papers, in [99] authors propose *Hypermedia Web API Support*, a Java framework based on JAX-RS 2.0 that offers annotations to semantically describe REST APIs. The end result is the description of the whole API in a JSON-LD document enriched with the Hydra vocabulary. Unfortunately, the framework is not available in Maven Central. In [151] Parastatidis et al. present *Restfulie*, a framework that uses resources, state transitions and content-negotiation as its core building blocks. We found 4 other frameworks that support HATEOAS features. They are all classified in Fig. 3.4 according to 23 criteria.

Synthesis

Despite the fact that only one framework enforces the *Atomic Resources* constraint, all frameworks allow reaching the highest level of maturity on the *design* dimension easily. This is because supporting the *Atomic Resources* constraint only requires developers to use the data model of the resource as the input of write operations and as the output of read operations.
We notice that only *API Platform* and *Restfulie* offer a mechanism to model relations between resources from which links are generated, instead of adding them programmatically in the response, thus increasing maintainability.
Otherwise, most frameworks do not ease the semantic and domain description of APIs. To us, this is the biggest challenge framework designers should tackle.
Last, as for IDLs, most frameworks creators do not provide mechanisms to describe resources as state machines, thus not taking advantage of its benefits.

Figure 3.4 – Implementation Frameworks Comparison Matrix

Technologies	Restfulie Ruby	Restfulie Java	Restfulie .NET	API Platform	Spring HATEOAS	JAX-RS 2.0	Hypermedia Web API Support	Ripozo	**Section 3.4 example**
Criteria									
Popularity (/2) - highest is better	**0**	**0**	**0**	**2**	**1**	**1**	**0**	**0**	
Design									
1 - Resources									
Separates domain model from URI model	■	■	■	■	■	■	■	■	☐
Operations with specific I/O	■	■	■	■	■	■	■	■	☑
2 - Protocol Compliance									
Operations with HTTP verbs		■	■	■	■	■	■	■	☑
Content negotiation	■	■	■	■	■	■	■	■	☐
Custom data-interchange formats	■		■	■	■	■	■	■	☐
3 - Atomic Resources									
Enforces atomic resources	■								☐
Profile									
1 - Interaction Profile									
Describes resources' properties				■	■		■		☑
Describes HTTP operations				■	■		■		☑
Describes templated URIs	■	■	■	■	■	■	■	■	☑
Pagination description				■			■		☐
2 - Domain Profile									
Hyperlinks to other resources	■	■	■	■	■	■	■	■	☑
Hypermedia controls				■		■	■	■	☑
Links modeled by the framework	■	■	■	■					☐
Models the authentication mechanism				■					☐
Non-functional properties description				■					☑
Models errors				■	■				☑
Semantic									
1 - Semantic Description									
RDF description of resources model and operations				■			■		☑
Machine-interpretable and deterministic preconditions (optional)				■					☐
Addition of other RDF vocabularies				■			■		☑
2 - Linked Data									
Links with human-interpretable semantic meaning	■	■	■	■	■	■	■	■	☑
Links with RDF semantic meaning				■			■		☑
Others									
Models the system as a FSM	■	■	■						☐
Generated links for internal resources	■	■	■	■	■	■	■	■	☐
Programming Language	RUBY	JAVA	.NET	PHP	JAVA	JAVA	JAVA	PY	
Criteria met (/23)	**8**	**6**	**7**	**17**	**8**	**6**	**12**	**6**	

3.4 Matrices usage example

In this section, we present the service of an insurance company that manages insurance contracts to illustrate how the presented comparison matrices can be used in a real-world scenario. This example is a light version of projects we have carried out at Fabernovel for large French insurance companies.

3.4.1 Domain description

To manage insurance contracts, the service holds five kinds of resources: (i) third-parties, (ii) contracts, (iii) warranties, (iv) cases, and (v) services. Third-parties, for example, customers or contractors, enter into contracts with the company. These contracts include warranties from the closed list that the company offers. For example, a Person A has the following warranties: (i) damage coverage with a deductible of $500 and a maximum repair amount of $30.000 and (ii) premium vehicle loan in the event of immobilization of the damaged vehicle. A contract can have several cases. When a customer of the insurance has a claim the company creates a case that holds its details and the services provided to the insured. For example, Person A has a car accident, he opens the insurance's web application and reports a claim, which leads to the company opening a case. His car has been destroyed and he is expected to attend a diner. Thus, on the app, he asks for the loan of a car that he will immediately recover.

3.4.2 Technological constraints

The service has to communicate with both internal and external components. Internal components are front-end applications, such as mobile or web applications, and other kernel services, such as payments. External components are contractors' APIs, for example, taxi or mechanics companies.

In the insurance domain, there is a huge amount of business rules that determine (i) the warranties an insured can include in a contract and (ii) the available services for a case, based on the specificity of the given case and the warranties of the contract. Writing and maintaining these rules both on the server and its clients is very costly and error-prone. Thus, we decided to keep these business rules on the server-side only. This constraint leads to the use of HATEOAS.

The project constitutes the core of the company's business, it should then be built with state-of-the-art technologies such as Linked Data. This enables the automatic creation of mash-ups and the use of a HyperGraphQL [6] middleware to easily query the whole IS [152]. Moreover, considering that the contractors providing services are very diverse and numerous, the interactions with their APIs should leverage the automatic discovery and composition offered by the use of RDF semantics.

There is also a high probability that new client systems will be built in the future, the API must document its resources, resource attributes, operations, URI templates, HTTP verbs, hypermedia controls, and errors in a machine-interpretable way. Moreover, because the service applies the CQRS pattern [7] we needed the IDL to enable associating an operation to its own input and output data model.

6. `https://www.hypergraphql.org/`
7. `https://martinfowler.com/bliki/CQRS.html`

Last, to minimize the disruption for software developers, we have chosen to keep the interchange formats as close as possible to what developers already know. It has therefore to be entity-centric, based on JSON and its structure had to be as close as possible to a JSON document without meta-data.

3.4.3 Selection of the technologies

From these constraints, we selected the set of criteria and features that the technologies should provide. These criteria are checked in the last column of Figure 3.2, 3.3, and 3.4. We then count the number of criteria that were provided by each existing technology. Results are presented in Figure 3.5, 3.6 and 3.7. For each matrix, the three technologies matching the highest amount of selected criteria are highlighted in green. It is important to note that the technologies do not have to match every criterion to be selected. Most of the time, missing features can be implemented afterward or proposed to the maintainers of the technologies.

Figure 3.5 – Results for interface description languages

Technologies	Rapido - CRUD option	Rapido - Hypermedia option	Modeling RESTful applications	Formal modeling of RESTful APIs	A model-driven approach for ...	Hydra	Atom	WSDL + SAWSDL	WADL	OpenAPI / Swagger	API Blueprint + MSON	hREST + MicroWSMO	RESTdesc	RADL	RAML	IO/ Docs
Popularity (/2) - highest is better	0	0	0	0	0	1	1	1	1	2	1	0	0	0	2	0
File format						RDF	Atom SF	XML	XML	JSON/ YML	MD	HTML	N3	XML	YML	JSON
Selected criteria met (/15)	**3**	**6**	**10**	**4**	**9**	**13**	**8**	**9**	**9**	**12**	**8**	**6**	**10**	**12**	**9**	**5**
Other criteria met	**2**	**3**	**5**	**2**	**3**	**5**	**4**	**3**	**3**	**5**	**5**	**3**	**7**	**6**	**6**	**3**
Criteria met (/31)	**5**	**9**	**15**	**6**	**12**	**18**	**12**	**12**	**12**	**17**	**13**	**9**	**17**	**18**	**15**	**8**

Interface Description Languages Hydra, OpenAPI, and RADL are the technologies matching the highest number of selected criteria. However, none matches all criteria. Hydra lacks the ability to describe non-functional properties and media-types, which can be done with other RDF vocabularies. RADL lacks the ability to semantically describe resources models, operations, errors, and non-functional properties, which can also be done with other vocabularies. On the other hand, OpenAPI does not support the usage of RDF vocabulary. In this project, we have chosen to set up both Hydra and OpenAPI. OpenAPI because it has the most features and it is a must-have today because of its tooling and popularity. Hydra because it can be easily completed with other vocabularies and used with JSON-LD.

Figure 3.6 – Results for data interchange formats

Technologies	JSON	HAL	Collection+JSON	Siren	Uber	Mason	Json:Api	Atom	Turtle	RDF XML	OData Json format	JSON-LD	JSON-LD + Hydra	JSON-LD + Hydra + SHACL
Popularity (/2) - highest is better	2	1	0	0	0	0	0	1	1	2	0	1	1	0
Selected criteria met (/14)	3	7	8	9	8	13	6	3	2	2	6	6	13	13
Other criteria met	1	4	1	4	3	7	2	2	2	1	5	2	3	7
Criteria met (/24)	4	11	9	13	11	20	8	5	4	3	11	8	16	20

Figure 3.7 – Results for implementation frameworks

Technologies	Restfulie Ruby	Restfulie Java	Restfulie .NET	API Platform	Spring HATEOAS	JAX-RS 2.0	Hypermedia Web API Support	Ripozo
Popularity (/2) - highest is better	0	0	0	2	1	1	0	0
Programming Language	RUBY	JAVA	.NET	PHP	JAVA	JAVA	JAVA	PY
Selected criteria met (/15)	7	6	7	15	10	8	13	8
Other criteria met	4	4	4	6	2	2	3	2
Criteria met (/23)	11	10	11	21	12	10	16	10

Interchange Formats Mason, JSON-LD + Hydra are the two technologies matching the highest number of selected criteria. JSON-LD + Hydra + SHACL is ignored as it does not match more selected criteria than without SHACL. While JSON-LD + Hydra lacks the ability to describe non-functional properties, Mason does not allow to use of RDF vocabularies. Being incompatible with RDF requires a lot more effort to compensate than finding another vocabulary. This explains why JSON-LD + Hydra was preferred over Mason in this context.

Implementation frameworks API Platform, Spring HATEOAS, and Hypermedia Web API Support [99] are the three technologies matching the highest number of criteria. The latter is immediately removed from the candidates because no public library is available. In this example, API Platform should be preferred over Spring HATEOAS because it matches five more criteria than Spring. However, developers of the companies we worked with know Java and not PHP. Moreover, the Spring framework is very popular with them, which compensates for the need to develop some features by hand. This is why we decided to go with Spring.

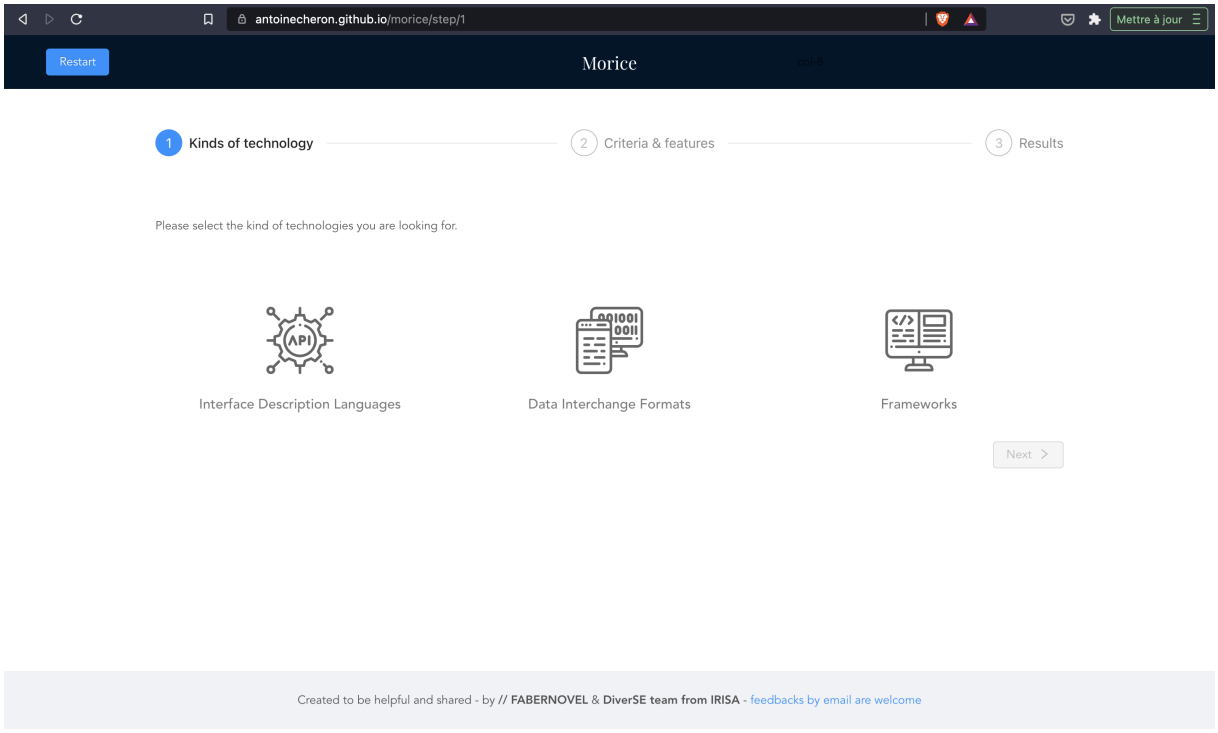
3.4.4 Easing the selection of the technologies

We have developed an open-source web recommender system named Morice [8] in which the user selects the type of technology he is looking for, the required criteria, and finally *scores* each criterion to indicate its importance. In return, the web application presents the list of technologies that meet the required criteria ordered by their respective *score*. By offering a two-step wizard, we reduce the process of identifying relevant technologies to a few minutes.

Usage example We showcase how Morice can be used in the same example as previously.

When one opens Morice, the home screen visible on Figure 3.8 is shown. Here, we are interested in all types of technologies so we select the three, that will be highlighted in blue when selected, and click the next button.

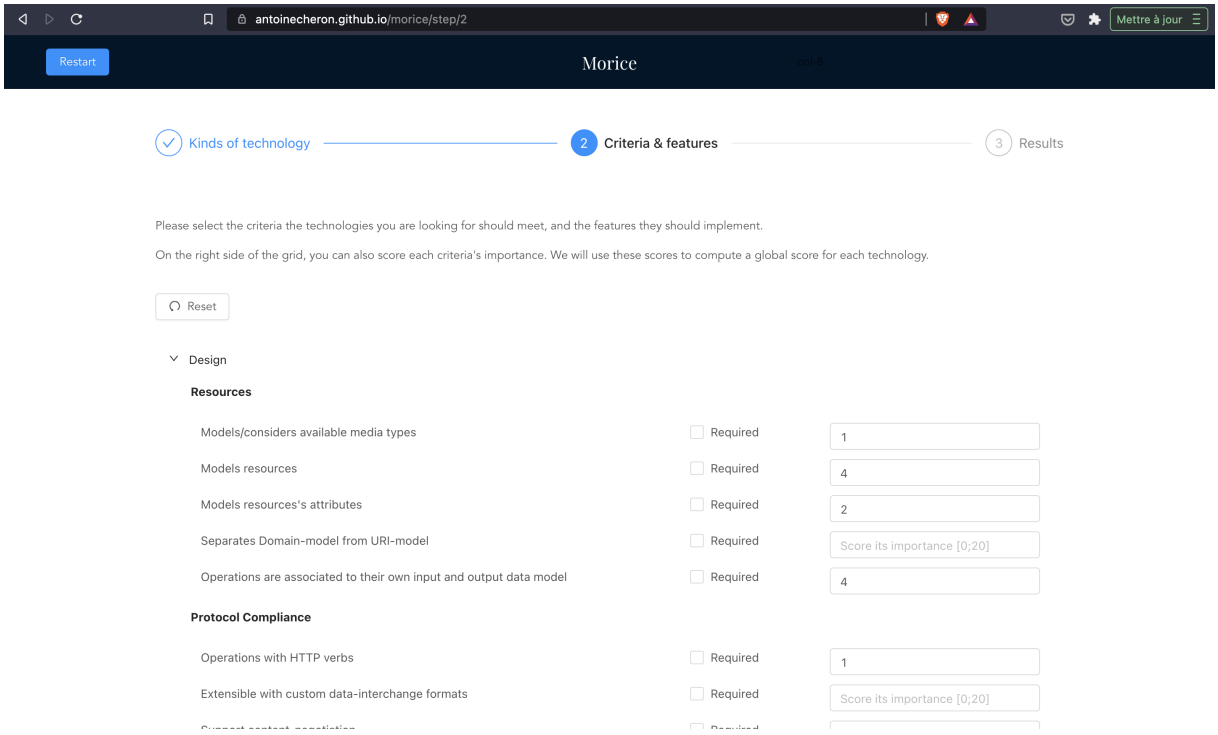
Figure 3.8 – Home screen of Morice, our web recommender system



The next page is dedicated to the scoring of the importance of each criterion and the selection of the required criteria. We therefore select the same criteria as on the figures 3.2, 3.3, and 3.4 by giving them a random score. Figure 3.9 shows Morice after the scoring of a few criteria. The score is set arbitrarily to demonstrate how the assistant behaves.

8. `https://antoinecheron.github.io/morice/`

Figure 3.9 – Scoring of the criteria on Morice, our web recommender system



At the end of the scoring, the user should click the next button at the bottom of the page. He is shown the results, as demonstrated in Figure 3.10. The figure shows only the results of the interface description languages only. The user should scroll down to see the results of the other categories.

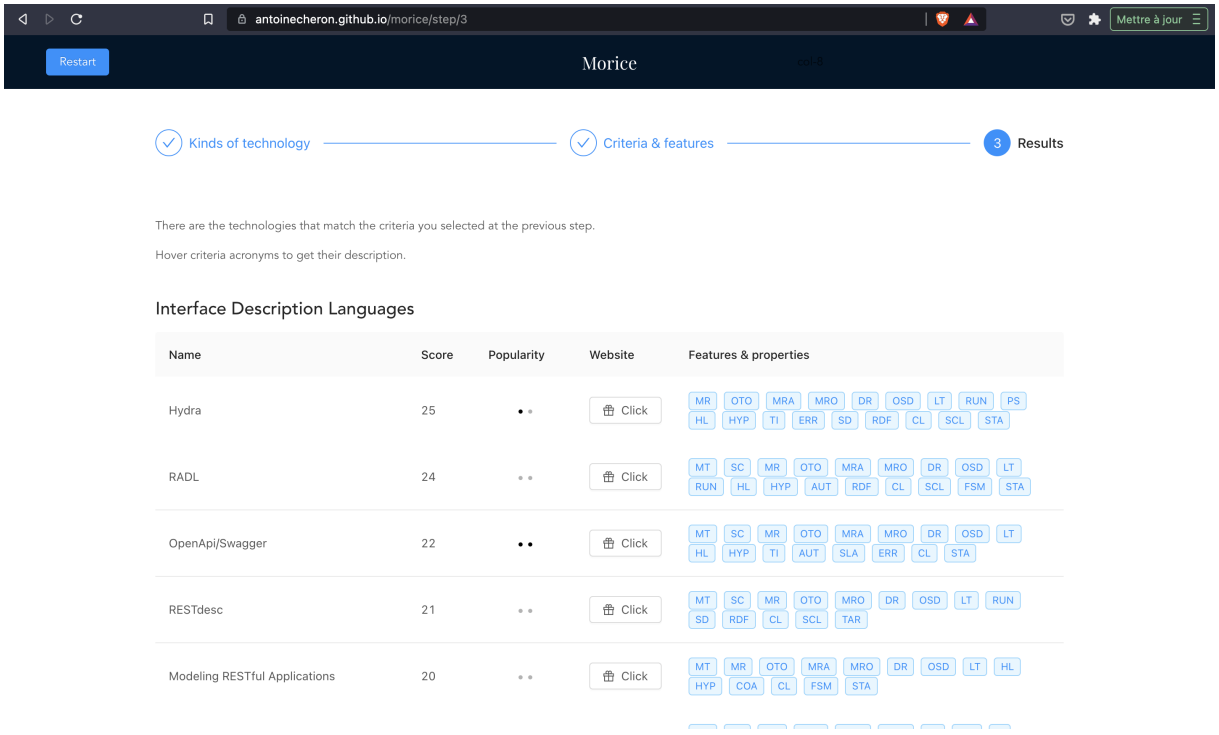
As we can see, the results are similar to Figure 3.5 but are slightly higher because for some criteria we used a score superior to 1. We think that this feature can help practitioners in their selection.

3.5 Discussion

This section provides our perspective on why no standard solution exists to meet all the defined criteria and highlights the possibility of new research initiatives.

First, none of the existing interchange formats support all the criteria described above, making it likely that new formats will emerge. For this reason, frameworks supporting Semantic REST APIs have to rely on formats that are likely to evolve, which will require additional effort and costs. This reduces the likelihood of framework editors investing time in developing such features.

Figure 3.10 – Results presented by Morice, our web recommender system



To us, the second and also the most important reason is that the well-known and widely used tools do not rely on Semantic REST APIs to provide additional and useful features. Among the possible functionalities, we envision various tools to automate API testing, REST client generation, API gateways, middleware, and smarter REST clients. In particular, we study how to design smarter clients that self-adapt to evolving REST APIs at run-time in the next chapter. We believe that this limits the adoption of Semantic REST APIs because the cost of building these APIs is not perceived as offering a sufficient short-term return on investment.

In addition to the absence of tools leveraging the additional metadata of Semantic REST APIs, it is not clear for the developers how to build them. A maturity model is available but no concrete guidelines for their design. This is another reason why we study how to leverage contracts documentation, hypermedia controls, and semantics to ease the co-evolution of REST APIs and their clients in chapter 4.

3.6 Findings Summary

In this work, we have presented three comparison matrices that assist architects in choosing Semantic REST APIs enabling technologies that meet their needs. Through a real example, we

have illustrated how the use of these matrices simplifies the choice of these technologies. As stated in the work, technologies should be chosen not only according to the number of criteria they meet, but also according to the specific needs of the project. To facilitate this selection, we have developed an assistant available online.

We also pointed out some interesting features missing in current technologies. The description of constraints and preconditions indicating the availability of state transitions is ignored by IDLs, vocabularies, interchange formats, and frameworks. On the other hand, resource modeling as FSM is not available in most frameworks. More importantly, well-known tools do not take advantage of the power of Semantic REST APIs to provide additional and useful features.

Based on these findings, we identify areas for improvement in the tools around Semantic REST APIs that we believe can increase its adoption. By leveraging the semantic description and advertising of state transitions and non-functional properties, automated testing tools can become smarter, middleware can automatically create responses from the composition of several APIs and REST client libraries can lower the coupling with the server as we study in the next chapter.

Chapter 4

Evolvable-by-design: Robust web UI clients to evolving REST APIs

Foreword

This chapter presents the second contribution of this thesis. It tackles the second challenge presented earlier: the break-free co-evolution of REST APIs and their clients. First, it challenges the existing list of API evolutions by interviewing experienced developers. As a result, seven new kinds of evolutions are added to the existing taxonomy. Second, the chapter presents a novel approach to this challenge. The approach presents 7 requirements that enable the creation of REST APIs exposing the proper information enabling the design of Web User Interfaces able to automatically adapt to any evolution of the REST API at runtime, without modifying their code. Two evaluations are conducted. A quantitative study assesses the ability of a Web UI to evolve to 110 evolutions of the 29 types presented in chapter 2. The evaluation shows that the Web UI could automatically evolve to 27 out of the 29 types of evolution. In the qualitative evaluation, the approach is implemented in 5 open-source projects available on GitHub. The approach could successfully be implemented on the 10 types of evolutions available in these projects and proved to be effective.

4.1 Introduction

We saw in chapter 1 that the evolution of an API frequently introduces breaking changes that must be fixed by the developers of the client [21]. This problem has been widely addressed by the software engineering community. However, existing works are not well suited to Web REST APIs [153] and their user interfaces. Indeed, automatic program repair techniques are not effective with dynamically typed languages, adapters address few kinds of evolutions and

hide the problem instead of fixing it, code generation techniques are too complex to configure to obtain good results and evolution policies significantly constrain the evolution instead of searching for solutions to enable REST APIs to evolve without breaking clients.

In this chapter, we tackle the well-known problem of REST API evolution from a novel perspective by making the Web UI *evolvable-by-design*. The goal is to make it possible for Web UIs to evolve by themselves to typical API evolutions without requiring the developers' intervention.

We define *evolvable-by-design* as the ability of a UI client to adapt to the following REST API evolution types, at run time, without breaking and without changing their code:

1. Additions and deletions to the parameters, URLs, and fields of the operations and data structure in-use;
2. Modifications to the parameters, URLs, and fields of the operations and data structure in-use, that does not change their semantics;
3. Modifications of control-flow and the precondition of the operations;
4. Movement of API elements.

Therefore, **as the main contribution of this work, we investigate how to make Web user-interface clients evolvable-by-design.**

Currently, web user interfaces break when Web APIs evolve because the API contract is hard-coded. Meaning, the URLs, parameters' name, preconditions, and others are fixed in the UI code. Consequently, any modifications of these elements require updating the code. Instead, we argue that a new paradigm is necessary. **The UI code should mimic the developer's behavior: to interpret the API documentation in order to always build correct API interactions.** Therefore, functional descriptors will be coded instead of the very specific contract of the API that works with a single version of the API.

To make this possible, we saw that with the REST architectural style, Fielding recommends the use of hypermedia controls. However, our experiments proved to us that this is not sufficient. We therefore argue that the UI must access sufficiently detailed API documentation that can be interpreted based on functional descriptors. By API documentation, we mean both static documentation, such as an OpenApi or WSDL document, and dynamic (or contextual) documentation, for example in the form of hypermedia controls. We propose to follow the Semantic REST approach and rely on semantically rich API descriptors (static and dynamic). However, how should these two types of documentation be used in combination? Therefore, to answer the main research question we investigate **the required structural and contextual information about the API that should be available to the UI to enable the creation of *evolvable-by-design* web user interfaces.**

To answer this, we first revisit the existing taxonomy of evolutions that can be found in the world of Web APIs, which we detailed in 1.7.3. After interviewing ten professional web

developers, we identified seven new kinds of API evolutions that complement other studies [141, 140, 21]. For each type of evolution, we then identify and highlight the API documentation elements the client has to interpret for an automatic and rewriting-free evolution. Therefore, based on this study, we propose a new approach to enrich existing Web API description standards in order to allow the construction of a client called *evolvable-by-design*.

The remainder of this chapter is as follows. Section 4.2 discusses the motivation of this work. Section 4.3 challenges and completes the existing taxonomy of REST API evolutions. The details of our approach are given in section 4.4. Section 4.5 evaluates this approach, followed by the Section 4.6 which concludes the work.

4.2 Motivating Example

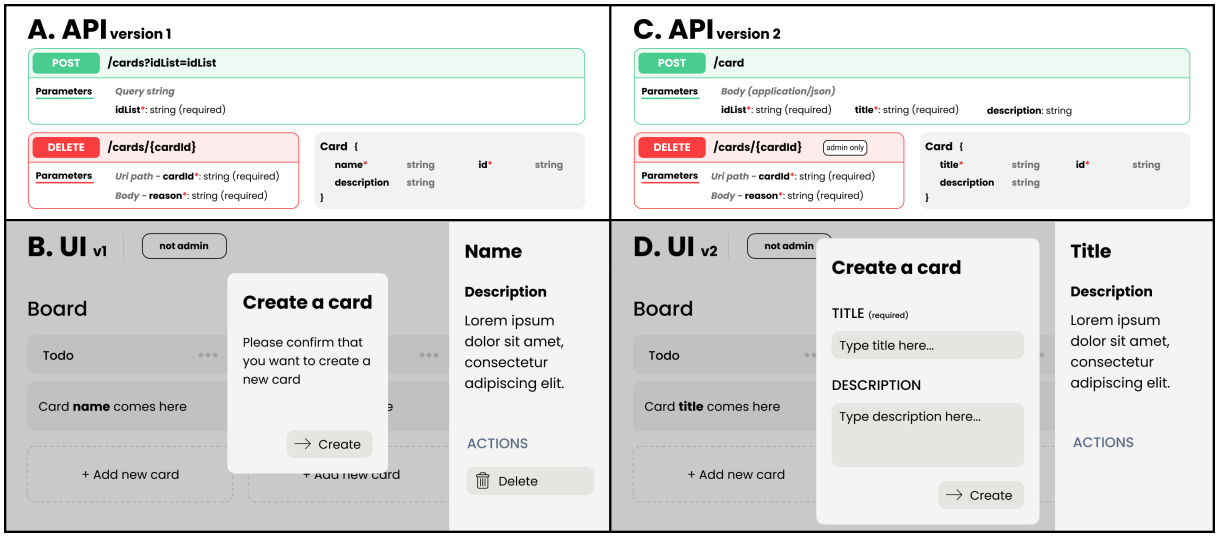


Figure 4.1 – Motivating Example

To motivate our work, this section introduces an example that illustrates the challenges that are faced with the maintenance of UIs client of REST APIs. We consider a Web Application like *Trello* [1], a collaboration tool that organizes projects into boards. It helps teams in managing their tasks on the Web by manipulating lists and cards similarly to moving sticky notes between columns on a blackboard. Here, the UI is designed to offer its users the following functionalities: (1) create task cards in lists, (2) modify a card's content, (3) move them between lists, (4) see the detailed information of a card and (5) delete cards. To access and modify the data, it uses an API. Then, when the API evolves, the UI will always have to use it properly to offer the aforementioned functionalities.

1. `https://trello.com`

Figure 4.1 presents the operations exposed by the API and the model of a `Card` on the top part (A and C), along with the resulting user interface on the bottom part (B and D). The operations to modify the cards were omitted.

As section A of Figure 4.1 shows, a card has a `name` and a `description` within the first version of the API. To create a card, the UI must send a POST request to `/cards?idList=idList` and it cannot provide a name or description at this step. Then, to delete a card, the UI sends a DELETE request to `/cards/{cardId}`. Accordingly, the UI displays lists as columns and the cards inside them (section B). A button to create a card is visible at the bottom of each list. Also, the button to delete a card is visible after the user clicks the card in order to see its details.

Then, the second version of the API comes with 4 breaking evolutions (section C). (1) `name` of `Card` is renamed into `title`, (2) the request to create a card must be sent to `/card`, (3) `idList` must be provided into the request body, along with (4) a `title` and a `description`. Also, only users that are administrators are now allowed to delete cards.

As a result, if the user interface is not updated, it would send requests to the wrong URL, omit required parameters, etc. The consequence is a broken UI. Thus, on a state-of-the-practice UI, a developer would have to update the code to support these evolutions. The modifications to do are the following: (1) all accesses to the title of the card would have to be updated, (2) the URL to create a card must be updated, (3) the `idList` parameter must be moved to the request body, (4) when the button to create a task is clicked, a form should require the user to input a title and a description, and (5) the button to remove a card must be displayed to administrators only. Next, the new code must pass the tests, be reviewed by other developers, and finally be redeployed in production, which is time-consuming and costly, as we mentioned before.

Instead, the code should mimic the developer's behavior when writing their UIs: referring to the API documentation. So, the code would not contain the URLs, the `name` or `title` keywords in the code neither manually build the API requests. It would also not have to implement the access-right management that the API already implements. Instead, it could rely on machine-interpretable semantics to find the proper information and URLs in the API documentation and on hypermedia controls, generated upon request, to get dynamic information such as the access rights and available operations.

We argue that UIs able to adapt to these kinds of evolutions without updating their code would make web applications significantly more robust. We refer to these kinds of UIs as *evolvable-by-design*, a concept that we introduced in this thesis. Thus, in the rest of the chapter, we present in detail our approach to enabling this property.

4.3 Seven new kinds of REST API evolutions

In section 1.7.3 we saw that [140, 141, 142] identified 22 kinds of REST API evolution that are listed in Table 1.1. However, these works did not study API evolutions developed and used inside companies, such as what Fabernovel does. Yet, these companies manage more complex access rights and more advanced business processes than public APIs. As a result, they often design CQRS REST APIs.

Thus, to challenge these results, we surveyed with a questionnaire nine professional Web developers who work in software firms and research laboratories.

We selected developers who work on Web applications from five different organizations and have at least one year of experience working with REST APIs. Six out of the nine respondents have five or more years of experience working with REST APIs. We asked nine questions. The first three to get information about the respondents. Then, one question asked them what evolutions they encountered. The response format is a free-text field to ensure that no bias is introduced. Finally, we asked five questions to challenge our intuitions. Each question asks them how likely are some kinds of evolutions to occur. The questionnaire and responses are available online [2].

After analyzing the results, we found that they reported that seven kinds of evolutions were not listed in [141, 140, 142]. Indeed, (23) the request method of an operation may change to denote a behavioral change; (24) the pre-condition may change to evolve the state machine of a resource or change access-rights; to achieve a business process (e.g. ordering a product), (25) the set or (26) the order of operations to execute sequentially may change, (27) any constraint of the input parameters are likely to change, not only the default value or upper bound (14 & 15), (28) parameters may be moved between the different parts of the HTTP request and (29) some values may be removed from the returned response. These additions to the state of the art are listed in Table 4.1. In this work, we replayed those newly identified API evolutions and observed that they are indeed breaking.

4.4 Approach

4.4.1 Approach Overview

Principle

This section details our approach to enable the design of *evolvable-by-design* user interfaces. To do so, rather than letting developers hard-code the contract with the API in the UI (i.e., to invoke the API by explicitly coding the URLs, keywords, formats, forms, etc.), we propose to shift the paradigm and code functional requirements towards the API. These requirements will

2. https://docs.google.com/spreadsheets/d/11dvV9vkvWeMd1pyGrA1PfW5TUan28GmPIa5yXoQgyzs/edit?usp=sharing

Table 4.1 – List of evolutions of RESTful APIs

From existing studies	
1	Add or Remove Parameter (✗ breaking)
2	Change Type of Parameter (✗ breaking)
3	Change Type of Return Value (✗ breaking)
4	Delete Method (✗ breaking)
5	Rename Method (✗ breaking)
6	Rename Parameter (✗ breaking)
7	Change Format of Parameter (✗ breaking)
8	Change Format of Return Value (✗ breaking)
9	Change XML Tag (✗ breaking)
10	Combine Methods (✗ breaking)
11	Split Method (✗ breaking)
12	Expose Data (✗ breaking)
13	Unsupport Request Method (✗ breaking)
14	Change Default Value of Parameter (✓ non-breaking)
15	Change Upper Bound of Parameter (✓ non-breaking)
16	Restrict Access to API (✓ non-breaking)
17	Move API elements (✗ breaking)
18	Rename API elements (✗ breaking)
19	Behavior change (✓ non-breaking)
20	Post condition change (✓ non-breaking)
21	HTTP header change (✗ breaking)
22	Error condition change (✓ non-breaking)
Additions of this work	
23	Request Method change (e.g. POST, PUT, etc.) (✗ breaking)
24	Precondition change (✓ non-breaking)
25	The order in which a set of operations must be played to achieve a business process changed (✓ non-breaking)
26	The set of operations to execute to achieve a business process changed (✓ non-breaking)
27	Change input parameter constraints (✓ non-breaking)
28	Move parameters (✗ breaking)
29	Remove returned value (✗ breaking)

be interpreted at run-time by the UI to create correct and up-to-date interactions with the API leveraging rich API documentation.

Figure 4.2 gives an example of such Web UI code. It displays the detail of a card and enables its deletion within the application illustrated in the motivation example in Fig. 4.1. This example is used to illustrate the requirements on the API side to enable an *evolvable-by-design* user interface. Therefore, this work does not focus on its implementation, but rather on the process followed by the UI designer and the API features that are therefore required.

```
1 const lib = new EvolvableByDesignLib(fetchApiDocumentation())
2 const DEL_SEMANTICS = '/dictionary#deleteAction' // OWL
3 function showCardDetailsComponent ({ card }) {
4   return <right-pane>
5     <h1>{lib.get('/dictionary#name').of(card)}</h1>
6     <h2>Description</h2>
7     <p>{lib.get('/dictionary#description').of(card)}</p>
8     <h2 class="grey">ACTIONS</h2>
9     <if test={lib.isOperationAvailable(DEL_SEMANTICS).on(card)}>
10      <pop-up-with-button buttonLabel="Delete"
11        formSchema={lib.getParamsSchema(DEL_SEMANTICS).of(card)}
12        onConfirm={(formValues) =>
13          CardService.delete(card, formValues)} />
14    </if>
15  </right-pane>
16 }
17 class CardService {
18   static delete (card, userInputs) {
19     lib.invokeOperation(DEL_SEMANTICS).on(card).with(userInputs)
20   }
21 }
```

Figure 4.2 – Web UI code example of the evolvable-by-design component of the right panel of the sections B and D of Figure 4.1. (Javascript and React v16)

This code is written by the UI developer to implement the custom UI visual components and navigation. To describe the aforementioned functional requirements, the developer uses machine-interpretable semantic descriptors designed with OWL (see lines 2, 5, 7, 9, 17 of Fig. 4.2). Then, a library leverages this semantics to browse the API documentation in order to figure out how to invoke it correctly. We provide an implementation of such a library [3]. It mimics what the developer normally does when he reads the API documentation to implement the code interacting with the API in the user interface. Yet, by doing this interpretation itself, at run-time, the UI becomes able to evolve without updating its code.

3. https://github.com/evolvable-by-design/pivo/tree/master/packages/pivo

Thus, when the UI needs to access a field on data that was already retrieved from the API, the library browses the API documentation to find the path of the field with the expected semantics. Then, it returns the value at this path. This is what happens in line 5 of Fig. 4.2. Hence, the developer is not required to update the code when the format of data changes.

To invoke an operation on the API, the developer should start by assessing its availability (line 9). Next, he can access the operation's parameters schema. Hence, he can pass them as an input of the visual component handling the operation, to let it generate a form for the parameters (line 11). This form is the only element that is generated on the UI. Then, when the user clicks the button to trigger the operation, the library builds the API request that complies with the documentation fetched earlier (lines 18, 19, and 20).

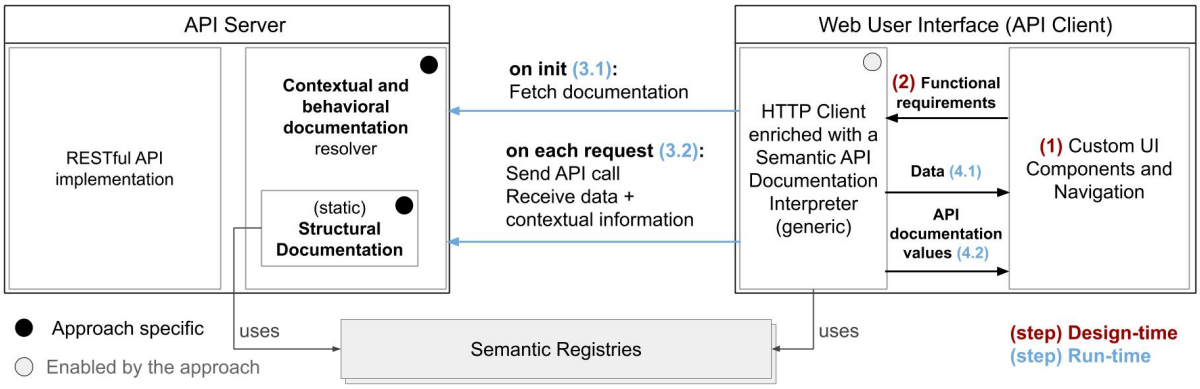


Figure 4.3 – Approach Overview

Therefore, to set up our evolvable-by-design approach, a web user interface must have a minimum set of information:

1. the detailed description of all the resources and operations exposed by the REST API, to be able to build syntactically correct requests (see R1)
2. semantic descriptors for all the resource properties, operations, and so on, to enable the UI to interpret the documentation based on the functional semantic descriptors (see R2)
3. explicit relationships between the properties and the object that they describe, and between the operations and the resource that they will apply on (see R3)
4. an API documentation that only contains the operations and resources that the connected user may be able to invoke, if available, to reflect the access rights and be able to hide the proper sections of the web applications such as an administration panel (see R4)
5. upon request, the list of the operations that are currently available on the returned resource, based on the current state of the resource. The goal is to remove the preconditions from the UI (see R5)

6. the default value of the operation's input parameters to make sure that the parameters that can not be known by the user will never be asked to the user (see R6)
7. given an operation that has multiple sets of input in the static documentation, when it is listed as available by the API, the set of input to use should be given in the same API response (see R7)

We will detail all the above necessary information and requirements in section 4.4.2 and 4.4.3.

Architecture

We now introduce an overview of the architecture of the approach in Figure 4.3. It illustrates the steps to display the end UI to a user. They are numbered in parenthesis in the following paragraph and on the figure accordingly.

First, the developer codes the *Custom UI Components and Navigation* similarly as Fig. 4.2. During this step, (2) he describes the functional requirements (lines 5, 7, 9, 11 and 19) to a *HTTP Client enriched with a Semantic API Documentation Interpreter* that is illustrated with the `lib` variable of type `EvolvableByDesignLib`. Then, at run-time, the documentation will be fetched when the UI loads in the browser (3.1). Next, the UI will interact with the API to get data and contextual information (3.2), such as the availability of an operation on a Card. Last, the HTTP client will return the expected data and API documentation values to the components in order to display the proper UI (4.1, 4.2).

Accordingly, in this section, we study what minimal set of documentation can be specified to counter the impact of the evolutions of APIs on Web UIs. Thus, how to ensure that UIs are *evolvable-by-design* and keep their interface and user experience consistent with how designers created them?

In the rest of this section, we detail the major components of this approach. Semantic registries are not detailed. Indeed, they are a core component of the Semantic Web [154] that is not specific to the approach.

4.4.2 Structural Documentation

A structural documentation describes the basic structure of an API, such as its functions and parameters. We identified that the structural documentation of the API must comply with the three following requirements to enable *evolvable-by-design* user interfaces.

In the following, we follow the original definition of a resource in the REST architecture [28]. Thus, a resource has a URI schema (e.g., `/card/{id}`) and a list of operations.

Describe the Resources in Detail (R1)

All resources, their URL schema, their operations, and their execution details along with the authentication mechanism must be documented in a machine-processable format.

This information is minimal to enable a machine to build syntactically correct requests and log a user into the system. Many technologies enable this, such as WSDL [4] and OpenApi [5]. Hence, in this subsection we focus on the peculiarities of this approach.

Operations description All operations must be described in detail: their URL schema, HTTP method, header, query, and body parameters along with the responses.

The documentation of the responses must describe the status codes, associated body, and headers schema. The potentially linked operations must also be documented. These include the operations that may be invoked on the resource itself, its relations with other resources (e.g. a friend), and control-flow information. The pre-conditions and post-conditions of the operations should not be described for two reasons: (i) the result of their resolution can be communicated directly, with less information than those necessary to describe them, as discussed in 4.4.3, and (ii) some organizations are reluctant to disclose such business-critical information.

Also, the parameters of an operation may vary depending on the type or state of the resource. In such a case, all alternatives must be documented and the alternative to use should be transmitted by the API at run-time with hypermedia controls. We discuss this point in 4.4.3.

Parameters description It must cover three cases.

— *The user can input the value himself/herself:* for this purpose, a precise description of the input type, including the data constraints, is sufficient. Existing machine-interpretable data constraint languages should be used, such as JSON Schema [6] or SHACL [7].

— *The user cannot input the value himself/herself and there are several relevant values in the context:* the input should be linked to a search function hiding the technical complexity from the user perspective. An example is given on Figure. 4.4.

— *The user cannot input the value himself/herself and there is only one relevant value in the context:* the server must provide the value to the UI at run-time through hypermedia controls as detailed in the subsection 4.4.3.

4. `https://www.w3.org/TR/wsdl.html`
5. `https://github.com/OAI/OpenAPI-Specification/`
6. JSON Schema Homepage - `https://json-schema.org/`
7. SHACL Specification - `https://www.w3.org/TR/shacl/`

```yaml
operationId: createProject
parameters:
  collaborators:
    x-research-function-operationId: searchUsers
```

Figure 4.4 – OpenApi documentation example mentioning a search function

```yaml
schema:
  x-@type: '/dictionary#Project'
  type: object
  properties:
    id:
      x-@id: '/dictionary#projectId'
      x-@type: 'http://purl.org/ontology/mo/uuid'
links:
  delete:
    x-@relation: 'https://schema.org/DeleteAction'
    parameters: ...
```

Figure 4.5 – OpenApi documentation example with semantic descriptors

Describe the Semantics (R2)

Each operation, property, and relation between operations must be described with machine-interpretable semantics using the W3C standard OWL [8]. These descriptors enable the UI to interpret the functional requirements and read the documentation as exemplified in Fig. 4.2.

Four types of semantic descriptors must be differentiated: (i) the meaning, (ii) the type, (iii) the format, and (iv) the relation. Indeed, two versions of an API may share the meaning of the *startTime* property with the same semantic descriptor, *https://vocab.ex#startTime*. However, one may format it as a timestamp and the other one as an ISO-8601 string. Consequently, the UI needs to be aware of this variation to evolve its behavior.

Lines 2, 6, 7 and 10 of Figure 4.5 give an example of such documentation.

OWL enables the machine-interpretable expression of complex relations between terms, such as the *sameAs* property stating that two URI share the same semantics. Consequently *https://my-vocab.com/Project* can be used in place of *https://schema.org/Project*. Thus, this *sameAs* property limits potential breaking evolutions. To illustrate this, Figure 4.6 gives an

8. https://www.w3.org/TR/owl2-overview/

```
@prefix schema_org: <https://schema.org/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .

this:Project a rdfs:Class;
    rdfs:label "Project"@en ;
    owl:sameAs schema_org:Project;
    rdfs:comment """An enterprise that is planned
        to achieve a particular aim."""@en ;
```

Figure 4.6 – Semantic vocabulary example of a Project

example of a Project ontology.

Moreover, many vocabularies are available in semantic registries online, such as *Schema.org* [9] and *Linked Open Vocabularies* [10]. Also, numerous Semantic Web and Linked Data works leverage machine-interpretable semantics, as detailed in 1.5.

Explicit Objects and Links Affiliation (R3)

Is required to determine if the data of an object or linked resource can be directly affiliated to the parent object or it is completely different data. Hence, each object of a document, no matter its depth, and each link must explicitly state the affiliation of its data. This is necessary to address the evolutions “change format of return value" (2) and “move API elements" (17).

An example of such object is given with `details` at the lines 3 to 5 of Fig. 4.7. The represented API response gives us the `id` and `title` of a **`Project`** along with contextual information that we discuss in the next subsection. Here, the `details` object is used to ease human readability. Yet, all the properties that it contains (only `title` here) belong to the project, not the `details` object itself.

With links, the case is very similar. Consider a **`Card`** resource with one analytic: its `creationDate`. Then, this field is moved to an **`Analytic`** resource. Hence, a link from **`Card`** to **`Analytic`** is added. Yet, it is required to distinguish this link to data describing the card from links to completely different resources such as the list of all cards.

As a result, we distinguish objects and links in which data are affiliated to the parent object from those affiliated to themselves.

9. `https://schema.org/`
10. `https://lov.linkeddata.es/dataset/lov`

4.4.3 Contextual and Behavioral Documentation

Among the contextual and behavioral documentation, we distinguish two categories. First, the structural documentation that is adapted to the user's access rights (R4). Second, information that should be provided by the server along with the resource representation (the data) in the response to each request (R5, R6, and R7).

Provide a WYSIWYG Documentation (R4)

All operations, resources, and properties that the user is not authorized to invoke or see should not be included. Thus, the documentation must be customized for each user that requests it.

This is required to adapt the UI to the user's access rights without adding descriptors to the documentation.

List the operations available on the returned resource with hypermedia controls (R5)

— The available resource's state transitions. For example, the operation to archive a project if it's not yet archived.

— The operations to access related resources. For example, given a user, his/her list of pending tasks.

— Business processes control-flow. For example, the next operation to invoke to continue the product ordering process.

Consequently, all the operations that are not available in the current application context should not be listed.

Give the default value of the operation's input parameters within the hypermedia controls (R6)

Amongst the default values to provide as contextual information are the parameters that can take only one relevant value in the context and that the user cannot know. For example, in the response of an API detailing a project, if the operation to create a new task into it is available, the *parent project id* of the future task should be provided by the server.

An example API response providing few data describing a `Project` (lines 2 to 5) along with contextual information (lines 6 to 14) is given in Figure 4.7.

```
1 {
2   "id": "project-1",
3   "details": {
4     "title": "Big Bang"
5   },
6   "contextual-information": [
7     {
8       "relation": "/ontology#createTask",
9       "link-id": "/doc#operations/createTask",
10      "inputModel": "/doc#models/createTaskType1",
11      "parameters": {
12          "parent-project-id": "project-1"
13      }
14    }
15 ]}
```

Figure 4.7 – An example of an API response that details a project's data and contextual information

Reference the operation's input model to use when multiple options are listed in the structural documentation (R7)

It should be done using a similar mechanism used to comply with R6. An example is given at the line 10 of Fig. 4.7.

4.4.4 HTTP Client enriched with a Semantic API Documentation Interpreter

We explained how the developer can leverage machine-interpretable semantics to describe functional requirements and design an *evolvable-by-design* client in 4.4.1. Hence, the UI developer can code every API call with a generic mechanism. This mechanism will interpret the API documentation and leverage the semantic descriptors to create the interaction with the API at run-time. This logic can be abstracted to enrich an HTTP client with a semantic API documentation interpreter. Doing so moves the complexity of the approach inside a library so that the UI developer can focus on the business logic.

We provide an open-source reference implementation of such semantic HTTP client, named Pivo, to ease the development of *evolvable-by-design* UIs. It is developed in TypeScript and is

framework-agnostic. It is available on GitHub [11] and NPM, and is open for contributions. In addition, we provide a documentation with more details on the approach, tutorials, and two UIs with a traditional and *evolvable-by-design* implementation.

Using this client instead of a traditional HTTP client imposes to implement the web UI differently, as exemplified in Fig. 4.2. Indeed, it should not implement direct calls to the API but functional requirements instead. Yet, most visual components and all the navigation logic can be kept unchanged.

This trade-off is necessary to design *evolvable-by-design* UIs. However, we argue that developers will be able to adapt to this new paradigm relatively quickly since it is not a radical change or a new language for writing UIs. Moreover, the long-term benefits surpass the entry cost by an important factor. We will further evaluate the cost of adopting our approach in section 4.5.

4.4.5 Synthesis and Discussion

Table 4.2 synthesizes all the elements that must be documented into the API to enable the design of *evolvable-by-design* user interfaces.

Reuse and limitations of the state-of-the-art

Our approach is designed to maximize the reuse of state-of-the-art technologies. Indeed, most of the detailed resource description (R1) can be described with OpenApi or WSDL and the semantic description (R2) with OWL. Also, (R5), (R6), and (R7) are common hypermedia-controls [29]. However, even if possible, not all developers specify this information in their APIs, and none combine them all. This work also identifies new requirements that OpenApi and WSDL must meet, namely: affiliation information (R3) and a WYSIWYG documentation (R4), which are necessary. In addition, we pinpointed the three types of parameters and accurately pointed out how to manage them in 4.4.2. Thus, this combination of existing technologies and the new requirements we add in this approach enable the powerful design of *evolvable-by-design* user interfaces.

Limitations of the approach

Our approach uses the semantics of the functionality and data exposed by the API as the central element to enable the UI to interpret the API documentation. Thus, the evolutions of the API that change the semantics of the functionalities or data cannot be handled. Hence, two of the changes listed in Table 1.1 cannot be handled by our approach: “combine methods" (10) and “split method" (11). Indeed, by combining or splitting methods, the API will then expose different functionalities than before, which changes the initial semantics. Nonetheless,

11. `https://github.com/evolvable-by-design/pivo`

Table 4.2 – Requirements of the approach on the API documentation

Structural Documentation	
R1	**Describe the Resources in Detail**: all resources, their operations and their execution details along with the authentication mechanism must be documented in a machine-processable format. Parameters that expect a value that the user can not know must be provided a default value or point to a research-function.
R2	**Describe the Semantics**: each operation, property and relation between operations should be described with machine-interpretable semantics
R3	**Explicit Objects and Links Affiliation**: each object and link of a document must explicit the affiliation of its data
Contextual Information	
R4	**Provide a WYSIWYG Documentation**: all operations, resources and properties that the user is not authorized to invoke or see should not be included
R5	**List the operations available on the returned resource with hypermedia controls**
R6	**Give the default value of the operation's input parameters within the hypermedia controls**
R7	**Reference the operation's input model** to use when multiple options are listed in the structural documentation

these 2 kinds of evolutions are seldom. They represented only 3 out of the 303 evolutions (less than 1%) observed by Li et al. in [141]. In addition, according to Roy Fielding, the author of the REST architecture, it is normal to manually modify the clients when the semantics of the interface change. Indeed, in an interview in December 2014, to the question "Does that mean as long as I use the REST style I am free and clear of versioning issues?", he answered "No. It is always possible for some unexpected reason to come along that requires a completely different API, especially when the semantics of the interface change or security issues require the abandonment of previously deployed software." [12].

Another limitation of our approach relates to the deletions of data and operations on the API. To minimize the documentation effort to enable *evolvable-by-design* UI, we do not encourage documentation of deleted items. Thus, a UI unable to find an operation or data will be unable to determine if it is due to (i) a deletion or (ii) the user not being allowed to access it. Accordingly, it is impossible for the UI to display a detailed message in such a case. Yet, if necessary, it is

12. Roy Fielding on Versioning, Hypermedia, and REST: `https://www.infoq.com/articles/roy-fielding-on-versioning/`

easy to document such information later on.

The approach presented in this work aims at allowing web UIs to evolve by themselves to typical API evolutions. However, not all web UIs should be made fully flexible to adapt to all the API side evolutions. Indeed, web UIs designers may want to control how their UIs can evolve. It would be relevant to investigate this question in future work as many situations can be encountered. Therefore, we recommended that web UI designers supervise them by adding a test suite to the UI client. They can for example expose an API endpoint that informs the UI of the API version to use. When a new API version is available, they can run the system test suite and leverage it to select the API evolutions to support and delay others until the next evolution of the code. To make this selection, the API endpoint returning the version to use may recommend different versions depending on the features. To us, this is possible because, in today's practices, API providers usually leave several REST API versions online for a moment in order to give their clients some time to adapt their code. In addition, web UI designers can leverage the machine-interpretable semantics exposed by the API. The richer it gets, the more accurate the UI can be made to select the kind of evolutions to support.

Moreover, while we recommend accompanying a UI client with a test suite, the evolution of the test suite must be handled as well. For example, an evolution of a regulation or new parameters added to an operation may require evolving the test suite accordingly. In those cases, automatic evolution of the UI is feasible with our approach but it will break the test suite. However, test maintenance is out of the scope of this work.

4.5 Evaluation

This section presents the evaluations we performed, regarding completeness, feasibility, and applicability in a real-world context.

The section is divided into four parts. First, Section 4.5.1 benchmarks our approach to all types of API evolution through a case study rigorously created for this purpose. It covers quantitative analysis. Section 4.5.2 evaluates the approach on five use-cases found on GitHub. It covers qualitative analysis. Then, Section 4.5.3 discusses the results, and finally Section 4.5.4 presents the threats to validity of our evaluation.

4.5.1 Benchmark

The first part of our evaluation performs a quantitative analysis. To do so, we use a benchmark that covers exhaustively all API evolutions. This benchmark is built to answer the following two research questions:

RQ1

Can a user interface autonomously adapt to all evolutions of a REST API that complies with the approach, without changing its code, at run-time? This aims to investigate the applicability and feasibility of our approach. It also evaluates its robustness w.r.t. REST API evolutions, in particular breaking evolutions.

RQ2

Does the implementation of an *evolvable-by-design* Web UI require additional development effort in terms of LOC? This aims to investigate the trade-off of our approach regarding the development effort of user interfaces.

Data set

To evaluate the approach, we needed a web application composed of a Web UI and a REST API that implemented all sorts of evolutions from Table 1.1, at least once. Unfortunately, we could not find such kind of open-source project and our industrial partner could not provide one for privacy reasons.

We developed a realistic web application imitating the project management software Jira [13]. With this application, multiple users collaborate on projects. A user can create public or private projects, invite other users to collaborate on the project, archive, delete or add tasks to it. The tasks have several operations and well-defined state transitions: they must be archived to be removable and only the tasks in a certain state can be completed, which can be customized by the users.

Figure 4.8 is a screenshot of the application displaying the details of a task.

To implement this application, **we developed three software components: (1) a REST API that exposes up to 28 operations and implements 110 evolutions split into 16 versions, (2) an *evolvable-by-design* UI, and (3) a state-of-the-practice UI where the contract with the API is hardcoded. We developed both UIs to be identical from the user perspective.** To demonstrate that the approach is feasible with modern technologies, the UIs are implemented in JavaScript with the React framework. The API server uses NodeJS and the documentation follows the OpenAPI Specification 3.0.0 that we extended to

13. Atlassian Jira - Home page - `https://www.atlassian.com/software/jira`

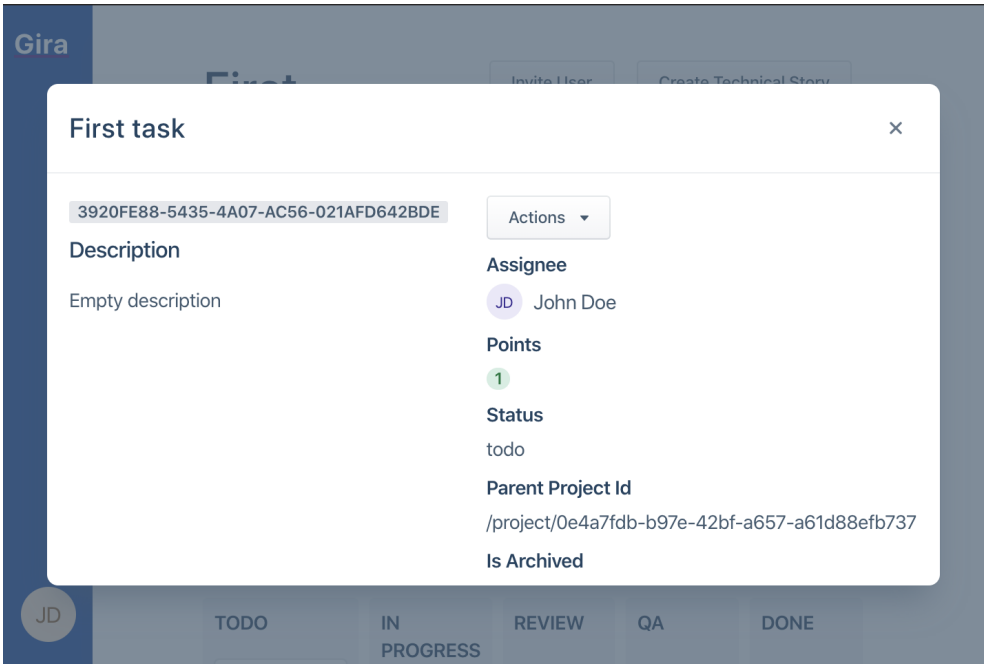


Figure 4.8 – Screenshot of the application developed to evaluate the approach

support OWL semantic annotations. Contextual documentation elements are transferred with hypermedia controls in the response body, using a custom format.

In order to ensure the reproducibility and transparency of this research, the three artifacts along with the documentation of all evolutions are publicly available online on GitHub [14]. Thus, the development history of each artifact and evolution can be observed in detail.

Also, the code of the *evolvable-by-design* UI includes a *library* folder that contains all the code interpreting the documentation of the API at run-time, which aims to meet the *evolvable-by-design* property. We later extracted this code in an open-source library that we named Pivo and uploaded to NPM [15]. Therefore, developers or scientists can handily reproduce the experiment or create their own.

Experimental Protocol

Here, we describe the experimental protocol used to evaluate the approach w.r.t. the research questions.

To test at least one variant of all kinds of evolutions listed in the taxonomy presented in Table 1.1, I created a REST API with 16 versions that implement 110 evolutions, including 59 breaking evolutions. For example, evolutions 5 and 7 of Table 1.1 were respectively applied 9

14. Evolutions: `https://bit.ly/2JlNg7E` - Clients: `https://bit.ly/2JlNT10` - Server: `https://bit.ly/2VhKZgf`
15. `https://www.npmjs.com/package/@evolvable-by-design/pivo`

Table 4.3 – Overall evaluation results

Client	**Breaking Evolutions**	**Non-Breaking Evolutions**	**LoC Updated**
Traditional Client	0/59	0/51	420
Evolvable-by-Design Client	57/59	43/51	98

and 3 times. I followed the same methodology as I do when developing web applications for Fabernovel clients.

As a second step, I implemented the first version of the two user interfaces, the traditional and evolvable-by-design user interfaces.

Then, for each upgrade of the API, I manually evolved the code of the “traditional UI" to implement the evolutions. On the *evolvable-by-design* UI, I refreshed the page and manually verified that all evolutions were automatically integrated while not introducing bugs. Otherwise, I updated the code.

Therefore, for each upgrade of the API, we count:

— *The evolutions that are automatically supported by the UIs without causing bugs* by manually testing each feature of the interface.

— *The lines of code that are changed to support breaking evolutions* by summing up the difference between additions and deletions of each commit related to this topic.

— *The lines of code that are changed to support non-breaking evolutions* by summing up the difference between additions and deletions of each commit related to this topic.

All the code of the traditional UI is included. On the *evolvable-by-design* UI, Pivo is the only component that is excluded because it is a generic library that is reused among projects.

Observed results

Table 4.3 summarizes the observed results of the evaluation, and more details are given in Table 4.4.

Table 4.4 – Detailed results of the case study [N/C: non-concerned]

API Version	1.0.0	2.0.0	3.0.0	4.0.0	5.0.0	5.1.0	6.0.0	7.0.0	7.1.0	8.0.0	8.0.1	9.0.0	9.1.0	9.2.0	9.3.0	10.0.0	Total
Breaking Evolutions automatically handled by the client																	
Total breaking evolutions	0	6	0	1	2	0	14	19	1	3	0	2	0	2	0	9	**59**
Traditional Client	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0/59**
Evolvable-by-design Client	0	6	0	1	2	0	14	19	1	3	0	0	0	2	0	9	**57/59**
Non-Breaking Evolutions automatically handled by the client																	
Total non-breaking evolutions	0	0	4	14	0	20	3	0	1	1	1	3	2	0	2	0	**51**
Traditional Client	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0/51**
Evolvable-by-design Client	0	0	4	14	0	20	0	0	1	1	1	0	0	0	2	0	**43/51**
Updated lines of code to support breaking evolutions																	
Traditional Client	N/C	16	N/C	4	2	N/C	70	28	1	28	N/C	44	N/C	23	N/C	14	**230**
Evolvable-by-design Client	N/C	0	N/C	0	0	N/C	0	0	0	0	N/C	7	N/C	0	N/C	0	**7**
Updated lines of code to support non-breaking evolutions																	
Traditional Client	1623	N/C	2	71	N/C	13	41	N/C	10	10	1	19	13	N/C	10	N/C	**190**
Evolvable-by-design Client	1395	N/C	5	12	N/C	0	28	N/C	5	0	0	11	30	N/C	0	N/C	**91**

RQ1 - Can a UI autonomously adapt to all evolutions of a REST API that complies with the approach, without changing its code, at run-time?

We observe that **the evolvable-by-design UI** can evolve itself to 57 out of the 59 breaking evolutions that are tested, which **addresses 27 out of the 29 (93%) evolutions of Table 1.1**. On the other hand, 43 out of the 51 non-breaking evolutions are also addressed **while the traditional UI has been affected by the evolutions**. Indeed, all evolutions required to update its code. As a result, the breaking changes indeed broke it, and some of the non-breaking changes produced irrelevant behaviors, for example on precondition change.

The two breaking evolutions that the UI is not able to address with the documentation are the combination and the split of methods. It confirms the limitations set out in 4.4.5.

On the other hand, among the non-breaking evolutions, the only kind of evolutions that the UI is not able to evolve to is the addition of methods and data. Because this is not related to the maintenance of an existing collaboration between the UI and the server, this property was not expected. Yet, for this very specific case, the UI can use the documentation to generate UI elements necessary to automatically integrate the new features. However, a designer will always have to refine the design to smoothly integrate them.

From this research question, we learn that **this semantic-based approach enables the automatic evolution of the UI, at run-time, to all evolutions that do not impact the semantics of the API**, which include 27 out of the 29 kinds of evolutions. Unfortunately, the combination and split of methods cannot be managed by this approach. Nonetheless, it still outperforms the traditional UI that did not evolve to any of the 29 kinds of evolutions, and hence, required manual adaptation.

RQ2 - Does the implementation of an evolvable-by-design Web UI require additional development efforts in terms of LOC?

First, we observe in Table 4.4 that the first implementation of the *evolvable-by-design* UI required less code than the traditional version (1395 vs 1623). Yet, we argue that the development effort is similar. Indeed, the higher level of abstraction needed to implement the *evolvable-by-design* version requires more cognitive effort than a copy of the contract into the code.

Then, every evolution of the API required an update of the traditional UI. On the other hand, on the *evolvable-by-design* UI, the code had to be updated only for the two unsupported breaking evolutions. In the end, 7 lines of code were updated on the *evolvable-by-design* UI and 230 on the traditional UI.

Regarding the integration of the new features of the API, we observe that they require more lines of code for the traditional UI than for the *evolvable-by-design* UI (190 vs 91).

We justify these results for three reasons: (1) the *evolvable-by-design* UI generates the form for the operations from small components (e.g. the email input field) and hence maximize abstraction and reuse while the other UI implements one component per operation. Also, it (2) validates the input from a generic code leveraging the data constraint language instead of implementing this logic for each input, and (3) it does not implement the access logic nor the conditions determining the availability of the operations.

As a result, from this research question, we learn that, **the first implementation of an *evolvable-by-design* UI is equivalent to a traditional UI. However, for the fewer code to write, its implementation requires anticipating unforeseen evolutions, which demands additional cognitive effort and developing new skills.** Yet, from these results we argue that the development of an *evolvable-by-design* UI reduces the long-term effort to evolve the UI for the next versions of the API, as it does not require to be maintained along with every API evolution. Thus, we observe that the overall development effort is greatly reduced in the long term.

4.5.2 Use cases

This section presents the evaluation we performed on five use cases that covers a qualitative analysis of our approach. The goal is to compare how our approach can make traditional UIs that break due to API evolutions robust, i.e., evolvable-by-design. The evaluation is built to answer the following research question.

RQ3

Can our approach be implemented on real-world and already existing web user interfaces? This aims to investigate the applicability and feasibility of our approach on already existing web applications, that provide new API evolution examples. It also evaluates the cost of implementing our approach in real-world projects.

We therefore selected five projects on GitHub that were evolved and adapted by maintainers to support the updates of the REST APIs they use. We then implemented our approach in each project, before its adaptation to each REST API evolution. Finally, we replayed the evolutions to evaluate the effectiveness of the approach.

Data set

For this evaluation, we needed software projects that meet the following requirements.

— Versioned with git

— Contains a web user interface written in JavaScript

— Uses a REST API that evolved, forcing maintainers to evolve the code of the web user interface to support the evolutions

In order to select the use cases for this evaluation, we followed the following steps:

1. We extracted the 395 APIs that have more than one version available in the API Guru OpenApi directory [16].
2. We removed from the list the APIs that are not implemented in at least one public GitHub repository written in JavaScript. To do so, we used the Search feature of the GitHub REST API. Unfortunately, it searches for a textual query into the README of the repositories, not into the code, and we could not find a REST API offering this feature. We thus searched for repositories written in JavaScript, with a query containing the name of the API along with the API keyword itself. For example, for the Google Drive API, we queried *https://api.github.com/search/repositories?q=Google Drive API + language:js*. At this step, we kept 92 APIs.
3. We listed the repositories using these APIs. We observed that each API is used in 1 to 10 057 repositories for a total of 14 394 repositories.

16. `https://apis.guru/openapi-directory/`

4. We cloned a maximum of 200 repositories per API. In the end, we cloned 2 778 repositories.
5. We analyzed the repositories to keep the ones where we could find usage of the API URL in the code, using the `git log -source -all -S 'API-URL'` command. At this step, we found 183 repositories.
6. We analyzed the repositories to keep the ones where more than one version of the API is used and where these several versions are found in different commits. At this step, we found 60 repositories.
7. As the last step, we manually analyzed the 60 repositories to verify that they meet the aforementioned requirements. In the end, we selected 4 of them. Most were rejected because they do not contain a user interface or do not implement evolutions, either because they already use the last version of the API, or leave their client broken. Indeed, a significant part of the analyzed projects were projects used to learn to program.

We automated steps 1 to 6 with scripts written in JavaScript. Then, to ensure the transparency of this research, we make them available online [17].

In addition, one coworker of ours, who teaches computer science to Master students could share with us a project that they developed. We use it as a fifth use-case.

Table 4.5 presents the main characteristics of the five selected projects. They represent small to medium projects covering few types, yet, real-world API evolutions. This is expected because they use public APIs that evolve rarely, or have a too-short life cycle to have been evolved many times. On the opposite, private APIs (i.e. internal to companies) evolve more frequently but their sources are closed. The next subsections detail each of the five use cases, i.e., projects, and summarize the work done on each one of them. Then, the last subsection presents the results of this empirical evaluation.

In order to ensure the reproducibility and transparency of this research, all the material is available online [18]. A central document details what was done on each repository, the links to the commits implementing our approach along with the instructions explaining how to test the evolutions. All the repositories were forked and all the work done on them is split into commits with explicit messages into a single branch named `evolvable-by-design`. In addition, the README was completed with all the details of the work done.

Use case 1 – "Dialog Flow"

Dialog Flow is a web application offering a conversational chat bot. It is **implemented as a single page application with the React library and comes with a**

17. https://github.com/evolvable-by-design/research/tree/master/utils/search-github-apps-with-api-evolutions
18. Empirical evaluation material: `https://github.com/evolvable-by-design/research/tree/master/experiments/evaluation-on-use-cases`

Table 4.5 – Artifacts selected for the empirical evaluation

GitHub name	**Size** (in lines)	**Number of evolutions**	**Evolution types**
lauchness/dialog-flow	721	1	3
IronistM/pagespeed_api_apps_script	63	2	3 and 5
danielsinclairtill/spaghetti-makes-me-moody	2 827	3	1 and 23
georgejacobt/utify	2 529	2	1
cbdt/Projet-Simba	1 663	14	1, 3, 5, 6, 10, 17, 26, 28 and 29

dedicated web server offering a REST API for the web UI. The application **uses the Google Dialogflow REST API**. In the beginning, this API was called from the web UI but a modification of the app moved this call to the server. Hence, the web UI now calls the server that calls the Google Dialogflow API on its own.

We found **one REST API evolution in this project**. Indeed, developers changed the model of the data returned by one API endpoint. At first, given `intentResponseContent` the object returned by the Google Dialogflow API, the API endpoint returned `"intentResponse": intentResponseContent` . After the evolution, it returns the `intentResponseContent` value directly, hence exposing the data upper in the document hierarchy. This change has been done when the usage of the Dialogflow API moved from the web UI to the server. We therefore had to split this into two steps: (1) move the call to the server, (2) make the evolution.

To implement our approach in this project, we had to go through four preliminary steps. We (1) forked the repositories, (2) moved the Dialogflow API call to the server, (3) created an OpenApi documentation with semantic descriptors, and (4) modified the web UI to use the API of the server. Then we could implement the approach in a single commit. **It required to add and update 43 lines of code to the project. We then replayed the API evolution and could observe that the web UI was indeed able to handle the evolution without modifying its code.**

Amongst the 43 added and updated lines of code, 8 adds the semantic vocabulary used in the web UI, 17 modify the existing code to use the approach to query the server API and read the result and 18 are used to fetch the API documentation on startup to display a “Loading" text during this operation and instantiate the library.

The metrics measured to evaluate the approach on this use case are presented in Table 4.6.

Use case 2 – "Pagespeed api apps script"

The project of this use case is a small script designed for Google Apps Script that **calls the Google PageSpeedInsight REST API** to **get the overall performance score of a website**. Hence, it returns only one figure. It is implemented in JavaScript and does not have a web user interface. Yet, the approach is applicable here because the types of changes applied here do not require a user to input data.

We found two REST API evolutions in this project. Indeed, the base URL and the type of the returned value of the endpoint used on the Google PageSpeedInsight API changed from version 1 to version 2. In the URL, the change is the version number, from *www.googleapis.com/pagespeedonline/v1* to */v2*. On the other hand, the response from the REST API has been completed. As a result, to retrieve the score from the response the path changed from `response.score` to `response.ruleGroups.SPEED.score`.

To implement our approach in this project, we had to go through four preliminary steps. We (1) forked the repositories, (2) created a branch before the evolutions, (3) adapted the application to run locally instead of in Google Apps Script, and (4) created a mock server for both versions of the PageSpeedInsight API along with the v1 documentation, because it is not online anymore. Then we could implement the approach in a single commit. **It required to add and update 15 lines of code to the project. We then replayed the API evolutions and could observe that the script was indeed able to handle the two evolutions without modifying its code.**

Amongst the 15 lines of code added or updated on the project, 5 are used to fetch the API documentation and instantiate the library, and 10 to update the existing code in order to use the library to make the call and read its result. The modifications are shorter here because the semantic vocabulary is not placed in a dedicated file but used as a string in the code, the architecture is simpler and fewer data are manipulated.

The metrics measured to evaluate the approach on this use case are presented in Table 4.6.

Use case 3 – "Spaghetti makes me moody"

Spaghetti makes me moody is a web application that tracks and stores users' accounts along with journal entries. It then **uses the Google Cloud Platform's Natural Language API** to infer and display the estimated mood, tags, and themes found within these journal entries. To achieve this, **the application is composed of a web user interface and a dedicated web server, both implemented in JavaScript. Within this use case, we study the evolutions of the REST API offered by the dedicated server.**

We found three REST API evolutions in this project. The first two are of the same type, they both add parameters to existing API operations. In the first case, a *historyData* field was added to the input of the user creation operation. Its purpose is to save in the database

the journal entries and corresponding Language API analysis that the user did before creating his account. The second case is the addition of the *username* and *password* parameters to the operation used to analyze the entry, in order to store the result in the history. The developers took this decision instead of implementing a standard authentication mechanism. The third REST API evolution is the modification of the HTTP verb used for the create user operation, from POST to PUT.

We implemented our approach in this project one evolution at a time. We thus explain the work done in the next three sections.

Evolution 1: addition of a historyData parameter To implement our approach for this evolution, we had to go through the same preliminary steps as for the previous use-cases and also had to create a Google Language API key. **Then, we could implement our approach in a single commit, in 57 lines of code. We then replayed the API evolution and could observe that the user interface behave as expected without changing its code.**

Amongst the 57 lines of code, 16 set the library up, 11 add semantic vocabulary to the project and 30 served to adapt the code to use our approach. This figure is different from the previous use cases because a custom solution is necessary for each use case in order to follow the architecture in place. In addition, we had to implement something new in this use case. For the first time, we had to create an object containing all the context of the component calling the library and to semantically describe its fields. This context contained the user information along with the component inputs and state. We thus learned from this use case that the component calling our library should provide a context to our library in order to let it browse the context and find potential values for the parameters of the operation. This is particularly relevant when user information is necessary, such as the username.

Evolution 2: addition of the username and password parameters Because the second evolution is of the same type as the first one, we applied the same logic to the operation analyzing the journal input instead of the user creation operation. **This time it modified or added 38 lines.** Among them, 9 added vocabulary into the application, and 29 modified the code to leverage our approach.

Evolution 3: request method change, from POST to PUT **This last evolution concerned the same operation as the first evolution. We thus did not need to adapt the user interface code because it already was. Hence, modifying the API documentation was sufficient.**

The metrics measured to evaluate the approach on this use case are presented in Table 4.6.

Use case 4 – “Utify"

Utify is a web application offering a music and entertainment channels integrator. It was developed by a single developer. Functionally, it broadcasts a keyword search to multiple entertainment data sources including YouTube, Spotify, and Pandora, and combines their results in a single web page. To achieve this, Utify is composed of a web user interface and a dedicated web server offering a REST API for the web UI. Thus, this is the server that does the REST API calls to YouTube, Spotify, and Pandara. About their implementation, both the server and the user interface are written in JavaScript. **Within this use case, we study the evolutions of the REST API offered by the dedicated server** because this is the only REST API that evolved during the project life cycle.

We found two REST API evolutions in this project. Indeed, the developer added a new required parameter (evolution of type 1 from Table 1.1) to the server's REST API two times. The first addition was the *userId* to the search request. The server used this data to retrieve the user's OAuth2 token in order to call the YouTube API. The second change was the addition of a tag to the videos' metadata in order to classify them and create categories on the web UI.

We implemented our approach in this project one evolution at a time. We thus explain the work done in two sections.

Evolution 1: addition of a userId parameter This first change can not be managed by our approach at first. Indeed, the evolution is the addition of a *userId* parameter to the API endpoint */api/search*. This parameter is used to identify the user making the request, because no standard authentication mechanism is used. The problem can hence be seen from two different angles.

First, it can be considered that because the authentication mechanism is not standard, it has not been possible to implement it in our library. Thus, the authentication can not be performed automatically and the generated requests will fail. Yet, this interpretation is limited. While it is true for this very use case, a more general interpretation is possible and is detailed in the next paragraph.

Second, the added parameter is an example of a parameter that the user can not know. Hence, it should be provided by the API, either directly into the documentation, or within hypermedia controls. Yet in this project, this is not possible because, to provide this information in the documentation or in hypermedia controls, the user context must be known. Here, this is not possible because the parameter is the element enabling the retrieval of the user context.

To overcome this issue, we implemented a standard authentication mechanism into the project using the *AuthenticationService* offered by our library.

Hence, to implement our approach in this project, we also had to go through four preliminary

steps. We (1) forked the project, (2) added our YouTube API key in the configuration, (3) created an evolvable-by-design branch, (4) standardized the authentication mechanism. Then we could implement our approach in a single commit. **It required to add and update 46 lines of code. We then replayed the API evolution and could observe that the web UI was indeed able to handle the evolution without modifying its code.**

Amongst the 46 lines of code added or updated on the project, 27 enable to fetch the API documentation, instantiate the library and display a loader while doing so; 7 add the semantic vocabulary to the application and 12 to update the existing code in order to use the library to make the call.

Evolution 2: addition of a tag parameter This change corresponds to the addition of a new feature to the web UI: sorting content per category. As aforementioned, this kind of evolution is not supported because it requires designers to think about how to modify the user experience to add this feature. Otherwise, the approach would fall into the category of automatically generated web user interfaces. Yet, with our approach, we are able to keep the user interface working with the evolved API. As a counterpart, the user experience will not be optimal but sufficient to make the web UI usable.

Before implementing our approach for this evolution, **we replayed the 16 commits between the two evolutions. We could notice that it did not break the feature using our approach. Then, we added the operation to the API documentation and implemented our approach in 18 lines. We then replayed the API evolution and could observe that the web UI was indeed able to handle the evolution without modifying its code.** At last, we applied the changes of the original commit on the user interface and verified that the software still worked, which it did.

Amongst the 18 lines to add our approach, 2 added semantic vocabulary entries and 16 updated the existing code in order to use the library to make the call to the API.

The metrics measured to evaluate the approach on this use case are presented in Table 4.6.

Use case 5 – “Simba"

Simba is a web application offering a meeting organization tool where the participants can vote on their preferred time slot. It is similar to *doodle.com*. Simba was developed by three developers. In addition to voting, users can indicate meal preferences and post comments. To achieve this, Simba is composed of a web user interface communicating with a dedicated backend through a REST API. Here we study the evolutions of this REST API. The particularity of this project is that it is another developer that developed the second version of the REST API, without communicating with the first authors.

We found fourteen REST API evolutions in this project, of nine different types. In-

deed, the developers added parameters (no.1 of Fig 1.1), changed the types of returned values (no.3), renamed a method (no.5), renamed parameters (no.6), combined methods (no.10), moved API elements (no.17), changed the set of operation to execute to achieve a business process (no.26), moved parameters (no.28) and removed returned value (no.29). For example, to comment on a poll, it was required to create a user and then comment by invoking two operations. A JSON structured as `{ "content": "the-comment" }` had to be sent to *POST /api/polls/slug/comments/{userId}*. With the v2 of the API, a single call to *POST /api/poll/comment/slug* is necessary, and the `username` should be included in the JSON of this operation invocation.

Before implementing our approach in this project, we documented and semantically annotated both APIs. Then we added a toggle on the web UI to easily switch between the two versions of the API. Next, we added all the semantic vocabulary used on the project into the web UI. Finally, **we added and configured our library into the web UI, which added 33 lines of code.**

Implementing our approach was very similar for 10 out of the 14 evolutions. First, we used our library to find the API operation with semantic descriptors. Next, we semantically annotated the data available on the web UI, and we used these annotated data to invoke the operation. Finally, we managed the API response either by keeping the response body as is when it did not change between versions, or by leveraging our library to retrieve the data from the response, based on its semantics, when the format of the returned data changed. **On average, it took 15 lines of code per operation to make evolvable-by-design.** For the two evolutions modifying a process, for example, to vote or comment a poll, we had to implement a function to follow the process leveraging our library. It took an additional 10 lines of code, once.

Handling the deletion of a parameter was different from the rest. Here, the deletion was linked to the deletion of a feature: the support of meal preferences. Then, the relevant manner to handle this change was to hide all the user interface elements related to this feature when it is not available on the REST API. Hence, on the web UI, we implemented a function that determines if the feature is available or not. If so, the UI elements are displayed, otherwise, they are not. This was relatively easy to implement, it took a total of 28 lines of code.

Among the changes, we were not able to automatically handle the *combine methods* change, as expected and already discussed in 4.4.

The metrics measured to evaluate our approach on these use cases are presented in Table 4.6.

Summary of the results

Table 4.6 summarizes the results of this experiment.

On the one hand, we observe that it took **on average 20 lines of code to fetch the API documentation and set up our library** from the web user interface, and it took **on average**

Table 4.6 – Detailed results of the experimental evaluation on five real world use cases

Name	**Lines of code**	**Evolutions**	**Types of evolutions**	**Commits including evolutions**	**Lines to evolve without approach**	**Lines to implement the approach**	**Developers**	**Broke tests?**	**Evolutions handled without updating the client code?**
Dialog Flow	721	1	Change return type (3)	1	2	43	1	No tests	Yes
Page Speed API Apps Script	63	2	Change return type (3) Change URL (5)	1	2	15	1	No tests	Yes
Spaghetti Makes Me Moody	2 827	3	Add parameter (1) Request method change (23)	Evol. 1: several Evol. 2: 1 Evol. 3: 1	Evol. 1: 4 Evol. 2: 22 Evol. 3: 1	Evol. 1: 57 Evol. 2: 38 Evol. 3: 0	4	No tests	Yes
Utify	2 529	2	Add parameter (1)	1 per evolution	Evol. 1: 7 Evol. 2: 2	Evol. 1: 46 Evol. 2: 18	1	No tests	Partial
Simba	1 663	14	1, 3, 5, 6, 10, 17, 26, 28 and 29	1 per evolution	Evol. 1&2: 10 Evol. 3: 15 Evol. 4: 2 Evol. 5: 66 Evol. 6: 5 Evol. 7: 9 Evol. 8: 8	Evol. 1&2: 56 Evol. 3: 15 Evol. 4: 15 Evol. 5: 28 Evol. 6: 24 Evol. 7: 15 Evol. 8: 19	3	No tests	Partial: all except the change of type combine methods

16 lines of code to apply our approach per API operation. On the other hand, it took **on average 7.5 lines of code to evolving the web UI without our approach**.

RQ3 - Can our approach be implemented on real-world and already existing web user interfaces?

We observe that:

1. We always could apply our approach on the 9/10 studied types of evolutions that we claim to support in Section 4.4
2. We always handled the evolutions when the system matched the requirements that we described in Section 4.4
3. The additional work to do on the web user interface to make it evolvable-by-design does not induce drastic changes. Indeed, it costs on average 20 lines of code for the setup and an additional 8.5 lines of code per API operation used.

After implementing our approach on five use-cases, we conclude that **the approach can be applied to real-world web applications. Furthermore, the additional lines of code to produce do not induce drastic changes.** Yet, we acknowledge that an additional design and cognitive effort is necessary to implement the approach. Indeed, it is less straightforward than making REST API calls following a well-defined REST API documentation. However, **we argue on the long-term benefit that will remove the burden of manual migration.**

4.5.3 Discussion

From our experiments, we observed that this approach comes with its own advantages and shortcomings.

Among the advantages, **it enables the design of web UIs that can evolve at run-time, automatically, to 27/29 known kinds of API evolution** while the traditional UI evolves to 0/29. Likewise, for non-breaking evolutions, on the benchmark, our approach could adapt to 51/59 evolution while the traditional UI evolved to 0/59.

In addition, **we were able to implement the approach in all the use cases we studied**. In addition, we consider that the effort to modify pre-existing use cases has remained low. Indeed, it took from 15 to 57 lines per change, where **on average 15 were necessary to modify an existing code invoking a REST API operation to make it evolvable-by-design**. The rest of the added code concerned the setup of the library and semantic vocabulary.

It is worth noting the possibility to make only certain areas of a web user interface evolvable-by-design, as the experiment on the use-cases proved. Hence, **a progressive migration can be achieved** and developers can decide to not make critical sections of their application evolvable-by-design at first.

Finally, to us, **the main benefit of the approach is on the long term with future evolutions**. Indeed, as shown with the third evolution of the third use-case, when a type of change is repeated several times it is not necessary to make the code evolvable-by-design again. Thus, over time, the more the API evolves, the more time development teams will save. Hence, web user interface developers can focus on valuable tasks and API providers gain freedom to evolve their API.

However, our approach also comes with shortcomings.

First, **an effort must be done by the API provider to document their API and follow the requirements presented in Section 4.4**. While providing rich API documentation is considered a good practice, making this a requirement is an impediment to the adoption of our approach. Indeed, for example, we were not able to make the web UI evolvable-by-design for one change of the fourth use case at first, because the API did not meet the requirements. Indeed, it used a non-standard authentication mechanism and thus could not provide hypermedia controls

for values that the user can not know. Therefore, it has been necessary to modify the API in order to use a standard authentication mechanism.

Also, our approach may bring new sources of mistakes. **A careful attention must be paid by developers to the following points, even if they are requirements of the approach.**

i) Replacing a semantic descriptor with another one can break the UI while the API did not change. For example, if *schema.org/Project* is replaced with *other.vocab/Project* and no *owl:sameAs* attribute links the two.

ii) The interface may ask the user for inputs that he/she cannot input. For example, if an input is required in a specific format such as a UUID, but no research function is documented. This would make the UI unusable. Yet, no exception would be raised by the software.

Moreover, producing a very detailed and semantically annotated API documentation is not trivial. Indeed, during our experiments, we sometimes had to fix the API documentation. Hence, **tools are necessary to accompany API designers producing documentations enabling our approach**.

In addition to the advantages and drawbacks, we consider it interesting to discuss the reasons why the number of lines of code differs between changes. In fact, three main factors influence this difference. First, the number of parameters sent to the API plays a role, we usually use one line to send a parameter with its semantic descriptor. Second, the dependency injection technology and the internal architecture of the code also play a role. Finally, the web UI sometimes just needs to execute an action, or to fetch data, or to fetch a data first and then execute actions on it. Each adaptation requires a different amount of code.

Finally, using our approach on a legacy code could require specific refactoring. Indeed, all the navigation within the data model is done asynchronously for all paths. Therefore, whether the data is present in memory or accessible as a result of a set of requests to the server is hidden from the developer. While this strong assumption may require a larger refactoring when using our approach on some existing Web applications (i.e. with synchronous calls, for example, as in use case 5), it is also a common pattern in advanced Web development. For example, JSON API specification allows nested complex types in attributes of an individual resource, but the complex type can not be another resource [19]. This pattern allows the architect to change the granularity of the objects sent by the back-end. This refactoring cost is thus induced by our approach but can also be considered as a cost related to the application of a classical pattern increasing the code quality on the client-side.

19. Clarification on resources composed of other resources: `https://bit.ly/3mJOKqz`

4.5.4 Threats to validity

This section discusses threats to validity [155].

Internal Validity

The first internal validity threat is w.r.t. the RQ1. I created the API evolutions and implemented the two user interfaces of the first experiment. This process presents a potential bias. To mitigate this, I developed both UIs with the same rigor and best practices that I follow when developing web applications for big corporations at Fabernovel, where more than 100 web and mobile developers work. Moreover, the second experiment does not present this bias and confirms that the approach indeed works for a sub-part of the evolutions (9 out of the 29). In addition, all the material is available online and can be cross-checked. Other researchers can therefore challenge these results by building new user interfaces on top of the same REST API.

Also, we tested our approach against 10 out of the 29 types of evolution over five real-world use cases. While all successful, we could not test the other types of evolution on real-world use cases. Hence, they were tested on a case study including two UIs in RQ1 for exhaustiveness purposes. Moreover, each kind of API evolution has been tested more than once to test scenarios that he meets on a daily basis. In particular, the breaking API evolutions but also non-breaking API evolutions.

External Validity

External validity first relates to the number of participants and the technologies involved. Indeed, we tested the approach over one case study and five use cases. Also, we implemented the servers with a single set of technologies. Although we cannot generalize our results, we are confident that they can be beneficial to any API and Web UI implementation technology. Indeed, on the server side, we did not use any language construct or feature specific to JavaScript. Similarly, the code leveraging the documentation on the user interface is written in JavaScript, which is the only programming language supported by web browsers. Therefore, it can already be used with other frameworks such as Angular [20] and VueJS [21].

External validity also relates to the lines of code (LoC) being used as the metric regarding maintenance effort. Therefore, the results can not be generalized as the modified lines of code are not statistically representative of the provided development effort. We used the LoC metric to provide an intuition of the development effort.

20. `https://angular.io/`
21. `https://vuejs.org`

Conclusion Validity

We tested our approach on our own over six web applications. From our experimentation, the results of the approach seem good and promising. However, a similar experimentation should be done on other case studies with developers, excluding ourselves, using different technologies and languages to confirm these encouraging results.

Moreover, we proposed an approach in which adoption strongly depends on the developers' understanding and engagement. It is therefore necessary to design an experiment where developers will implement UI clients with the proposed approach to assess their interest. This is left as future work. In this work, we solely evaluated the feasibility and applicability of the proposed approach on real-world use cases.

4.6 Conclusion

This work proposed an approach that enables the design of *evolvable-by-design* user interfaces that are clients of REST APIs. They are user interfaces that evolve automatically to the changes of the APIs, at run-time, and without modifying their code. We proposed to decrease the coupling between the client and its API by relying on the ability to have semantically rich REST API documentation. In particular, we studied what structural and contextual information should be documented in the APIs for web user interfaces to be *evolvable-by-design*.

Our approach was evaluated over both a quantitative and qualitative study. Indeed, we first evaluated it over a case study with an API that implements 110 evolutions through 16 versions. We compared the effort required to evolve a traditional client with an *evolvable-by-design* client. It showed that the UI could evolve to 27/29 kinds of evolutions at run-time without modifying its code, which represents 99% of real-world evolutions and outperforms traditional user interfaces that can not evolve without modifying their code. Moreover, while the design of an *evolvable-by-design* user interface required a slightly superior effort at first, it proved to significantly reduce the maintenance effort in the long term. Then, we implemented our approach in five use-cases that are real-world open-source web applications found on GitHub. These use-cases implement 20 REST API evolutions of 10 different kinds. As a result of this experiment, we could successfully implement our approach for the 20 evolutions. It thus proved that our approach can progressively be implemented in real-world web applications at a reasonable cost. Indeed, it took on average 15 lines of code to make a REST API operation invocation evolvable-by-design.

Chapter 5

Conclusions and perspectives

In this thesis, we first reviewed how the World Wide Web evolved. We saw that the Web is made of many applications and services that communicate with each other using REST APIs. We therefore explored the main research surrounding REST APIs such as how contracts are expressed, how some tasks could be automated with Semantic Web technologies, how they evolve and how to ease their co-evolution.

We thus identified two pain points that developers face when working with REST APIs. First, the many available technologies to design, document, and implement REST APIs make their selection a time-consuming and error-prone task. Second, there is no satisfying solution for the break-free co-evolution of REST APIs. Indeed, the state of the practice is to limit evolutions because most of them are breaking clients. Developers create new versions of their APIs to avoid breakage but are however reluctant to create many of them because otherwise they would lose clients.

So, as a first contribution of this thesis, we proposed three comparison matrices to assist developers in the selection of the right technologies for the system to design. After the identification of candidate technologies, we read their documentation. Then, we designed a set of criteria to compare them and we finally produced the comparison matrices. We then showcased how these matrices can be used through an example. To facilitate selection, we have developed an assistant available online [1]. From this work, we learned that whereas there are many technologies to design Semantic REST APIs, it is not clear how to actually implement them and how to leverage the additional metadata to solve real-world problems. We advocated for additional studies that specify what metadata should be exposed to enable new usages such as automated service testing or break-free co-evolution.

As a second contribution of this thesis, we proposed a novel approach to the co-evolution of REST APIs and their clients. We argued that clients of REST APIs should not be evolved manually but should be able to adapt to typical REST API evolutions, at run-time, without breaking, and without changing their code. To enable this, the clients of REST API need rich documentation. We therefore studied what structural and contextual information should be documented in the APIs for Web user interfaces to be *evolvable-by-design*. We gave an open-

1. `https://antoinecheron.github.io/morice/step/1`

source implementation of a library enabling this on the client-side along with an open-source example of an evolvable-by-design Web User Interface with its evolving REST API. Our approach was evaluated over both a quantitative and qualitative study. With the quantitative study, we demonstrated that our UI could evolve to 27/29 kinds of evolutions at run-time without modifying its code. The qualitative study implemented the approach on the five real-world open-source Web applications that we could find on GitHub. These use-cases implement 20 REST API evolutions of 10 different kinds. As a result of this experiment, we could successfully implement our approach for the 20 evolutions. With the two experiments, we could prove the feasibility of the approach.

With these two contributions, we tried to address the major scientific challenges faced by REST API providers throughout the entire life cycle of their REST APIs. We could also identify areas for future research.

Perspectives

The next question to tackle is how to drive adoption of the proposed co-evolution approach? To achieve this, the approach must be easy to understand, implementable with reasonable effort, and attractive.

To test the required effort and attractiveness, as a first perspective we propose to lead a crossover study (see 5). Depending on the results, if the approach is complex to understand or not attractive enough, communication and teaching efforts will be necessary but are out of the scope of scientific research. To us, the last thing to address is the ease of implementation. On the one hand, it requires advanced user experience studies to design the client-side library's API, which are related to industrialization and engineering. On the other hand, we identify interesting research topics. To us, one of the major challenges is the selection and manipulation of ontologies. So, as a second perspective (see 5) we propose to explore solutions to propose ontology recommendation from the code editor, leveraging on machine learning techniques to drive standardization and obtain good precision and to also leverage both the code and API documentation.

Now, if we imagine that developers use our approach, REST APIs would be semantically described and Web applications would be evolvable-by-design. In such a context, we see interesting topics to explore. A first topic is the use of GraphQL [2] to query a set of Semantic REST APIs with no prior knowledge of them (see 5). A second topic is the description and partial automation of functional tests (see 5).

The following sections detail the four perspectives.

2. `https://graphql.org/`

First perspective: crossover study

As a first perspective, we propose to extend this work with a crossover developers study. This work proposed a new approach to the co-evolution of REST APIs and their clients, that enables clients to evolve to 27 out of the 29 known kinds of REST API evolution at run-time without changing their code. We therefore studied the effectiveness of the approach and its applicability to real-world applications.

A natural next step is to test the potential adoption and the extent to which the proposed approach may prove attractive to web developers. For this purpose, we propose to lead a crossover developers study, which is a variant of within-subject design in which all participants are exposed sequentially to all variances of the task. With such a study, we would be able to compare the effort required to implement and maintain the client of a REST API with and without our approach.

We believe that additional and extensive user studies should be conducted to minimize bias, as the library that will be given to developers to test the approach can strongly influence the results.

We started the design of such an evaluation. The material is available online [3] and may be re-used.

Second perspective: ontology recommendation within the code editor

As a second perspective, we propose to study how to ease the manipulation of Semantic Web / Linked Data ontologies by developers. In order to design evolvable-by-design clients, developers often manipulate ontologies. The developers of the REST API do so when they document the REST API and the developers of the clients use ontologies to describe the functional requirements. The easy selection and manipulation of ontologies offering the best coverage for a particular domain is therefore crucial to drive adoption. Moreover, this subject raises interesting research opportunities.

To date, the Linked Open Vocabularies platform hosts 757 ontologies [4]. Selecting ontologies, identifying gaps, and enriching them is therefore cumbersome for developers. The idea is thus to help the developer select the most relevant ontologies, or terms when several ontologies are required, from his code editor. To achieve this, we propose to design an extension to code editors and Integrated Development Environments (IDE) that would open with a shortcut and let the developer search for semantic terms. The goal is to ease ontology selection, improve reuse and standardization.

Previous research proposed several ontology search engines and recommendation systems [156,

3. `https://github.com/evolvable-by-design/research/tree/experiment/crossover-developer-study/experiments/crossover-developers-study`

4. Source: `https://lov.linkeddata.es/dataset/lov/vocabs` - Accessed: 06/30/2021

157, 158, 159, 160]. Extensions to these works explored how to apply them to specific domains such as Internet of Things [161]. However, it is widely pointed out that classical ontology model is not sufficient to deal with imprecise and vague knowledge strongly characterizing some real-world applications. There are thus many proposals for fuzzy extensions to ontologies that intend to deal with such vague and imprecise information [162]. However, to the best of our knowledge, no technique proposes to leverage the entire API description or the code of an application to improve recommendations. We therefore propose to explore this. In addition, we believe that machine learning techniques can help improve the precision of the recommendations, reuse, and standardization. Besides, we would like to explore which architecture would be relevant to make recommendations in real-time from the code editor.

Third perspective: easing the design of GraphQL API mashups

In recent years, a new architectural pattern emerged. Instead of directly calling several REST APIs from a web client (most often a web or mobile application), some developers started using a sort of proxy server that is the single API called by the web client. This proxy is in charge of analyzing the query and then calling the proper REST APIs in order to produce the desired effect and build the response. The goal is to reduce querying times by factoring several queries into a single one and reducing complexity for the front-end developer by moving the orchestration logic to this sort of proxy server. The most popular technology used to achieve this in 2021 is GraphQL [5], a graph query language designed by Facebook engineers. In practice, the client sends to the server a representation of the data that it expects to receive as a response, within a format very close to JSON. The server that supports GraphQL should however be implemented manually. This is partly because the REST API contracts are hardcoded, and also because a new interface must be designed specifically for very few clients based on existing REST APIs, with the objective to make the interface as easy to understand and manipulate as possible for the developers of the clients.

The third perspective of this work is therefore to ease the design of this component. We observe that the creation of such a component starts by defining the schema of the data to expose. Then, developers write software components that will fetch multiple REST APIs upon request and then gather the data into a single response. We argue that developers should not write those connectors manually. Instead, they should rely on model-driven approaches. In practice, they should specify which data should be exposed and in which REST APIs this data is. Then, the connectors that fetch multiple REST APIs and intelligently factor the responses should either be generated or generic.

So, a first idea is to design a DSL for this purpose. In practice, the API designer would select the REST APIs to use. Next, the DSL editor would retrieve the documentation of all these

5. `https://graphql.org`

APIs. Then, the designer would use the DSL to specify the composition to achieve. We would like to explore how to design such a DSL leveraging on the semantic description of Semantic REST APIs to manage orchestration and composition at run-time. We will base our research on known bridges between GraphQL and RDF [6]. Existing works propose to query stored RDF graphs with GraphQL, extend GraphQL with JSON-LD, and transform GraphQL queries into SparQL queries.

A first challenge of this work is to adapt the existing approaches regarding GraphQL to become compatible with Semantic REST APIs and the approach we proposed as a second contribution of this thesis. A second challenge is the design of the DSL itself. In addition, we propose to have the DSL editor fetch the documentation of the REST APIs that the API designer wants to use. So as a third challenge, we will have to find techniques to make this information easy to manipulate. Ideally, we would like to make accurate recommendations from the editor by reusing the ontology recommendation engine described in our second perspective. However, we do not know if this is valuable for the API designer or not so we may have to find a different technique.

Fourth perspective: a Domain-Specific Language for scenario-based functional testing of Semantic REST APIs

As a fourth perspective, we envision a Domain Specific Language (DSL) that developers of services would use to describe the scenarios to test. The goal is to enable them to describe the functionalities to test without worrying about the technical details. To select the functionalities to test, we would reuse the ontology recommendation engine described in our second perspective. From the description of the scenario to test, the engine would generate the API requests and then take care of orchestration and consistency. In addition, thanks to the availability of data schema descriptions in the ontologies or API description, we could automatically leverage property-based testing techniques. With such a Domain-Specific Language, we envision that domain experts would be enabled to write functional test scenarios.

In contrast to the popularity of REST, systematic testing of REST APIs has not attracted much attention so far. The only model-driven approach to the test of REST APIs that we could find proposes to both generate the REST API code and a test suite [74]. Thus, to the best of our knowledge, no DSL address the scenario-based functional testing of existing REST APIs or generates the API requests. Moreover, we believe that the presence of semantic descriptors into the documentation can be helpful to drive consistency and minimize errors during the orchestration of multiple API interactions.

Bibliography

[1] M. Lanthaler, "Third generation web apis," Ph.D. dissertation, Ph. D. dissertation, Institute of Information Systems and Computer Media ..., 2014.

[2] I. Salvadori and F. Siqueira, "A maturity model for semantic restful web apis," in *2015 IEEE International Conference on Web Services*. IEEE, 2015, pp. 703–710.

[3] R. Cailliau and D. Connoly, "A little history of the world wide web," https://www.w3.org/History.html.

[4] "Most popular websites 1996 - 2019," https://www.youtube.com/watch?v=2Uj1A9AguFs, accessed: 2021-09-12.

[5] G. Thies and G. Vossen, "Web-oriented architectures: On the impact of web 2.0 on service-oriented architectures," in *2008 IEEE Asia-Pacific Services Computing Conference*. IEEE, 2008, pp. 1075–1082.

[6] R. Taelman, M. Vander Sande, and R. Verborgh, "Graphql-ld: linked data querying with graphql," in *ISWC2018, the 17th International Semantic Web Conference*, 2018, pp. 1–4.

[7] A. Martin-Lopez, "Ai-driven web api testing," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, 2020, pp. 202–205.

[8] S. Segura, J. A. Parejo, J. Troya, and A. Ruiz-Cortés, "Metamorphic testing of restful web apis," *IEEE Transactions on Software Engineering*, vol. 44, no. 11, pp. 1083–1099, 2017.

[9] J. Wang, X. Bai, L. Li, Z. Ji, and H. Ma, "A model-based framework for cloud api testing," in *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, vol. 2. IEEE, 2017, pp. 60–65.

[10] A. Arcuri, "Restful api automated test case generation with evomaster," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 1, pp. 1–37, 2019.

[11] T. Espinha, A. Zaidman, and H.-G. Gross, "Web api growing pains: Loosely coupled yet strongly tied," *Journal of Systems and Software*, vol. 100, pp. 27–43, 2015.

[12] D. Dig and R. Johnson, "How do apis evolve? a story of refactoring," *Journal of software maintenance and evolution: Research and Practice*, vol. 18, no. 2, pp. 83–107, 2006.

[13] J. Li, Y. Xiong, X. Liu, and L. Zhang, "How does web service api evolution affect clients?" in *2013 IEEE 20th International Conference on Web Services*, 2013, pp. 300–307.

[14] J. Henkel and A. Diwan, "Catchup! capturing and replaying refactorings to support api evolution," in *Proceedings of the 27th international conference on Software engineering*, 2005, pp. 274–283.

[15] T. McDonnell, B. Ray, and M. Kim, "An empirical study of api stability and adoption in the android ecosystem," in *2013 IEEE International Conference on Software Maintenance*. IEEE, 2013, pp. 70–79.

[16] M. Monperrus, "Automatic software repair: a bibliography," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, pp. 1–24, 2018.

[17] B. Dagenais and M. P. Robillard, "Semdiff: Analysis and recommendation support for api evolution," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 599–602.

[18] N. Forsgren and D. Smith, "Accelerate State of DevOps 2019," 2019. [Online]. Available: https://cloud.google.com/devops/state-of-devops/

[19] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Addison-Wesley Professional, 1994. [Online]. Available: http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/ref=ntt_at_ep_dpi_1

[20] A. Bennaceur and P. Inverardi, "Automated synthesis of connectors to support software evolution," Jan. 2012, http://ercim-news.ercim.eu/en88/. [Online]. Available: https://hal.inria.fr/hal-00662058

[21] M. Fokaefs, R. Mikhaiel, N. Tsantalis, E. Stroulia, and A. Lau, "An empirical study on web service evolution," in *2011 IEEE International Conference on Web Services*. IEEE, 2011, pp. 49–56.

[22] P. Leitner, A. Michlmayr, F. Rosenberg, and S. Dustdar, "End-to-end versioning support for web services," in *2008 IEEE International Conference on Services Computing*, vol. 1. IEEE, 2008, pp. 59–66.

[23] Adidas. (2018) Adidas - rules for extending. [Online]. Available: https://adidas.gitbook.io/api-guidelines/general-guidelines/rules-for-extending

[24] Zalando. (2020) Zalando restful api and event scheme guidelines. [Online]. Available: https://opensource.zalando.com/restful-api-guidelines/

[25] T. Espinha, A. Zaidman, and H.-G. Gross, "Web api growing pains: Stories from client developers and their code," in *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE, 2014, pp. 84–93.

[26] Y. Syu, S.-P. Ma, J.-Y. Kuo, and Y.-Y. FanJiang, "A survey on automated service composition methods and related techniques," in *2012 IEEE Ninth International Conference on Services Computing*. IEEE, 2012, pp. 290–297.

[27] R. Kanagasabai *et al.*, "Semantic web service discovery: state-of-the-art and research challenges," *Personal and ubiquitous computing*, vol. 17, no. 8, pp. 1741–1752, 2013.

[28] R. T. Fielding, *Architectural styles and the design of network-based software architectures*. University of California, Irvine Irvine, 2000, vol. 7.

[29] A. Cheron, J. Bourcier, O. Barais, and A. Michel, "Comparison matrices of semantic restful apis technologies," in *International Conference on Web Engineering*. Springer, 2019, pp. 425–440.

[30] T. J. Berners-Lee, "The world-wide web," *Computer networks and ISDN systems*, vol. 25, no. 4-5, pp. 454–459, 1992.

[31] ——, "Information management: A proposal," Tech. Rep., 1989.

[32] H. W. Lie and B. Bos, "Cascading style sheets, level 1," 1996.

[33] "What actually is a java applets vulnerability?" https://security.stackexchange.com/questions/195619/what-actually-is-a-java-applets-vulnerabilitys, accessed: 2021-09-15.

[34] M. J. Taylor, J. McWilliam, H. Forsyth, and S. Wade, "Methodologies and website development: a survey of practice," *Information and software technology*, vol. 44, no. 6, pp. 381–391, 2002.

[35] J. J. Garrett *et al.*, "Ajax: A new approach to web applications," 2005.

[36] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall, "A note on distributed computing," in *International Workshop on Mobile Object Systems*. Springer, 1996, pp. 49–64.

[37] T. Berners-Lee, J. Hendler, and O. Lassila, "The semantic web," *Scientific american*, vol. 284, no. 5, pp. 34–43, 2001.

[38] S. A. McIlraith, T. C. Son, and H. Zeng, "Semantic web services," *IEEE intelligent systems*, vol. 16, no. 2, pp. 46–53, 2001.

[39] M. A. Jadhav, B. R. Sawant, and A. Deshmukh, "Single page application using angularjs," *International Journal of Computer Science and Information Technologies*, vol. 6, no. 3, pp. 2876–2879, 2015.

[40] A. Biørn-Hansen, T. A. Majchrzak, and T.-M. Grønli, "Progressive web apps: The possible web-native unifier for mobile development," in *International Conference on Web Information Systems and Technologies*, vol. 2. SCITEPRESS, 2017, pp. 344–351.

[41] "History of front-end frameworks," https://blog.logrocket.com/history-of-frontend-frameworks/, accessed: 2021-04-30.

[42] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," *Present and ulterior software engineering*, pp. 195–216, 2017.

[43] R. Perrey and M. Lycett, "Service-oriented architecture," in *2003 Symposium on Applications and the Internet Workshops, 2003. Proceedings.* IEEE, 2003, pp. 116–119.

[44] D. Krafzig, K. Banke, and D. Slama, *Enterprise SOA: service-oriented architecture best practices.* Prentice Hall Professional, 2005.

[45] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano, "Devops," *Ieee Software*, vol. 33, no. 3, pp. 94–100, 2016.

[46] S. Sohan, C. Anslow, and F. Maurer, "Automated example oriented rest api documentation at cisco," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP).* IEEE, 2017, pp. 213–222.

[47] J. Yu, B. Benatallah, F. Casati, and F. Daniel, "Understanding mashup development," *IEEE Internet computing*, vol. 12, no. 5, pp. 44–52, 2008.

[48] B. Meyer, "Applying'design by contract'," *Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[49] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins, "Making components contract aware," *Computer*, vol. 32, no. 7, pp. 38–45, 1999.

[50] A. Beugnard, J.-M. Jézéquel, and N. Plouzeau, "Contract aware components, 10 years after," *arXiv preprint arXiv:1010.2822*, 2010.

[51] B. Rubinger and T. Bultan, "Contracting the facebook api," *arXiv preprint arXiv:1009.3715*, 2010.

[52] J. Gregorio, "The atom publishing protocol," 2005.

[53] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana *et al.*, "Web services description language (wsdl) 1.1," 2001.

[54] B. Elvesæter, A.-J. Berre, and A. Sadovykh, "Specifying services using the service oriented architecture modeling language (soaml)-a baseline for specification of cloud-based services." in *CLOSER*. Citeseer, 2011, pp. 276–285.

[55] F. H. Vogt, S. Zambrovski, B. Gruschko, P. Furniss, and A. Green, "Implementing web service protocols in soa: Ws-coordination and ws-businessactivity," in *Seventh IEEE International Conference on E-Commerce Technology Workshops*. IEEE, 2005, pp. 21–26.

[56] S. Ran, "A model for web services discovery with qos," *ACM Sigecom exchanges*, vol. 4, no. 1, pp. 1–10, 2003.

[57] H. Cao, J.-R. Falleri, and X. Blanc, "Automated generation of rest api specification from plain html documentation," in *International Conference on Service-Oriented Computing*. Springer, 2017, pp. 453–461.

[58] M. J. Hadley, "Web application description language (wadl)," 2006.

[59] C. Pautasso, "Bpmn for rest," in *International Workshop on Business Process Modeling Notation*. Springer, 2011, pp. 74–87.

[60] C. Pautasso, A. Ivanchikj, and S. Schreier, "Modeling restful conversations with extended bpmn choreography diagrams," in *European Conference on Software Architecture*. Springer, 2015, pp. 87–94.

[61] C. Pautasso, "Composing restful services with jopera," in *International conference on software composition*. Springer, 2009, pp. 142–159.

[62] ——, "Restful web service composition with bpel for rest," *Data & Knowledge Engineering*, vol. 68, no. 9, pp. 851–866, 2009.

[63] D. Bonetta, A. Peternier, C. Pautasso, and W. Binder, "S: a scripting language for high-performance restful web services," *ACM Sigplan Notices*, vol. 47, no. 8, pp. 97–106, 2012.

[64] D. Chappell, "Introducing odata," *Data Access for the Web, The Cloud, Mobile Devices, and More*, pp. 1–24, 2011.

[65] "Social web protocols working group note," https://www.w3.org/TR/social-web-protocols/, accessed: 2021-06-09.

[66] T. J. Mowbray and R. C. Malveau, *CORBA design patterns.* John Wiley & Sons, Inc., 1997.

[67] J. Snell, D. Tidwell, and P. Kulchenko, *Programming web services with SOAP: building distributed applications.* " O'Reilly Media, Inc.", 2001.

[68] H. Ed-Douibi, J. L. C. Izquierdo, A. Gómez, M. Tisi, and J. Cabot, "Emf-rest: generation of restful apis from models," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, 2016, pp. 1446–1453.

[69] F. Haupt, D. Karastoyanova, F. Leymann, and B. Schroth, "A model-driven approach for rest compliant services," in *2014 IEEE International Conference on Web Services.* IEEE, 2014, pp. 129–136.

[70] M. Polák and I. Holubová, "Rest api management and evolution using mda," in *Proceedings of the Eighth International C* Conference on Computer Science & Software Engineering*, 2015, pp. 102–109.

[71] B. Terzic, V. Dimitrieski, S. Kordic, G. Milosavljevic, and I. Lukovic, "Microbuilder: A model-driven tool for the specification of rest microservice architectures," in *International Conference on Information Society and Technology*, 2017, pp. 179–184.

[72] S. Pérez, F. Durao, S. Meliá, P. Dolog, and O. Díaz, "Restful, resource-oriented architectures: a model-driven approach," in *International Conference on Web Information Systems Engineering.* Springer, 2010, pp. 282–294.

[73] F. Valverde and O. Pastor, "Dealing with rest services in model-driven web engineering methods," *V Jornadas Científico-Técnicas en Servicios Web y SOA, JSWEB*, pp. 243–250, 2009.

[74] V. Schreibmann and P. Braun, "Model-driven development of restful apis." in *WEBIST*, 2015, pp. 5–14.

[75] R. C. da Cruz Gonçalves and I. Azevedo, "Restful web services development with a model-driven engineering approach," in *Code Generation, Analysis Tools, and Testing for Quality.* IGI Global, 2019, pp. 191–228.

[76] C. Zolotas, T. Diamantopoulos, K. C. Chatzidimitriou, and A. L. Symeonidis, "From requirements to source code: a model-driven engineering approach for restful web services," *Automated Software Engineering*, vol. 24, no. 4, pp. 791–838, 2017.

[77] M. N. Michuki, "Exposing rich update operations via rest apis."

[78] B. Adida, M. Birbeck, S. McCarron, and S. Pemberton, "Rdfa in xhtml: Syntax and processing," *Recommendation, W3C*, vol. 7, p. 41, 2008.

[79] D. Beckett, T. Berners-Lee, E. Prud'hommeaux, and G. Carothers, "Rdf 1.1 turtle," *World Wide Web Consortium*, pp. 18–31, 2014.

[80] M. Sporny, D. Longley, G. Kellogg, M. Lanthaler, and N. Lindström, "Json-ld 1.0," *W3C recommendation*, vol. 16, p. 41, 2014.

[81] S. Bechhofer, F. Van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, L. A. Stein *et al.*, "Owl web ontology language reference," *W3C recommendation*, vol. 10, no. 02, 2004.

[82] T. Berners-Lee, "Linked data," https://www.w3.org/DesignIssues/LinkedData.html, accessed: 2021-06-16.

[83] J. Kopeckỳ, T. Vitvar, C. Bournez, and J. Farrell, "Sawsdl: Semantic annotations for wsdl and xml schema," *IEEE Internet Computing*, vol. 11, no. 6, pp. 60–67, 2007.

[84] M. Maleshkova, J. Kopeckỳ, and C. Pedrinaci, "Adapting sawsdl for semantic annotations of restful services," in *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer, 2009, pp. 917–926.

[85] R. Khare and T. Çelik, "Microformats: a pragmatic path to the semantic web," in *Proceedings of the 15th international conference on World Wide Web*, 2006, pp. 865–866.

[86] J. Kopeckỳ, K. Gomadam, and T. Vitvar, "hrests: An html microformat for describing restful web services," in *2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, vol. 1. IEEE, 2008, pp. 619–625.

[87] A. P. Sheth, K. Gomadam, and J. Lathem, "Sa-rest: Semantically interoperable and easier-to-use services and mashups," *IEEE Internet Computing*, vol. 11, no. 6, pp. 91–94, 2007.

[88] M. Lanthaler and C. Gütl, "Hydra: A vocabulary for hypermedia-driven web apis," in *LDOW*, 2013.

[89] D. Martin, M. Paolucci, S. McIlraith, M. Burstein, D. McDermott, D. McGuinness, B. Parsia, T. Payne, M. Sabou, M. Solanki *et al.*, "Bringing semantics to web services: The owl-s approach," in *International Workshop on Semantic Web Services and Web Process Composition*. Springer, 2004, pp. 26–42.

[90] J. De Bruijn, C. Bussler, J. Domingue, D. Fensel, M. Hepp, M. Kifer, B. König-Ries, J. Kopecky, R. Lara, E. Oren *et al.*, "Web service modeling ontology (wsmo)," *Interface*, vol. 5, no. 1, p. 50, 2005.

[91] R. Verborgh, T. Steiner, D. Van Deursen, J. De Roo, R. Van de Walle, and J. G. Vallés, "Capturing the functionality of web services with functional descriptions," *Multimedia tools and applications*, vol. 64, no. 2, pp. 365–387, 2013.

[92] R. Verborgh, T. Steiner, D. Van Deursen, S. Coppens, J. G. Vallés, and R. Van de Walle, "Functional descriptions as the bridge between hypermedia apis and the semantic web," in *Proceedings of the third international workshop on restful design*, 2012, pp. 33–40.

[93] I. Rauf and I. Porres, "Towards behaviorally enriched semantic restful interfaces using owl2," in *International Conference on Web Engineering*. Springer, 2011, pp. 407–410.

[94] D. John and M. Rajasree, "Restdoc: Describe, discover and compose restful semantic web services using annotated documentations," *International journal of Web & Semantic Technology*, vol. 4, no. 1, p. 37, 2013.

[95] M. Lanthaler and C. Gütl, "A semantic description language for restful data services to combat semaphobia," in *5th IEEE International conference on digital ecosystems and technologies (IEEE DEST 2011)*. IEEE, 2011, pp. 47–53.

[96] ——, "On using json-ld to create evolvable restful services," in *Proceedings of the Third International Workshop on RESTful Design*, 2012, pp. 25–32.

[97] R. Verborgh, A. Harth, M. Maleshkova, S. Stadtmüller, T. Steiner, M. Taheriyan, and R. Van de Walle, "Survey of semantic description of rest apis," in *REST: Advanced Research Topics and Practical Applications*. Springer, 2014, pp. 69–89.

[98] A. A. Patil, S. A. Oundhakar, A. P. Sheth, and K. Verma, "Meteor-s web service annotation framework," in *Proceedings of the 13th international conference on World Wide Web*, 2004, pp. 553–562.

[99] I. L. Salvadori and F. Siqueira, "A framework for semantic description of restful web apis," in *2014 IEEE International Conference on Web Services*. IEEE, 2014, pp. 630–637.

[100] M. Maleshkova, C. Pedrinaci, and J. Domingue, "Supporting the creation of semantic restful service descriptions," 2009.

[101] M. Klusch, "Semantic web service coordination," in *CASCOM: Intelligent Service Coordination in the Semantic Web*. Springer, 2008, pp. 59–104.

[102] H. Dong, F. K. Hussain, and E. Chang, "Semantic web service matchmakers: state of the art and challenges," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 7, pp. 961–988, 2013.

[103] M. Klusch, P. Kapahnke, S. Schulte, F. Lecue, and A. Bernstein, "Semantic web service search: a brief survey," *KI-Künstliche Intelligenz*, vol. 30, no. 2, pp. 139–147, 2016.

[104] J. Garofalakis, Y. Panagis, E. Sakkopoulos, and A. Tsakalidis, "Contemporary web service discovery mechanisms," *J. Web Eng.*, vol. 5, no. 3, pp. 265–290, 2006.

[105] K. Sycara, M. Paolucci, J. Soudry, and N. Srinivasan, "Dynamic discovery and coordination of agent-based semantic web services," *IEEE Internet computing*, vol. 8, no. 3, pp. 66–73, 2004.

[106] G. C. Hobold and F. Siqueira, "Discovery of semantic web services compositions based on sawsdl annotations," in *2012 IEEE 19th International Conference on Web Services*. IEEE, 2012, pp. 280–287.

[107] A. Günay and P. Yolum, "Semantic matchmaking of web services using model checking," in *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 1*. Citeseer, 2008, pp. 273–280.

[108] K. Li, H. Li, and J. Chen, "Semantic web service discovery algorithm based on constraint extraction and structure analysis," *International Journal of u- and e- Service, Science and Technology*, vol. 9, pp. 21–30, 2016.

[109] M. Klusch, B. Fries, and K. Sycara, "Automated semantic web service discovery with owls-mx," in *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, 2006, pp. 915–922.

[110] A. Bernstein and C. Kiefer, "Imprecise rdql: towards generic retrieval in ontologies using similarity joins," in *Proceedings of the 2006 ACM symposium on Applied computing*, 2006, pp. 1684–1689.

[111] E. Toch, A. Gal, I. Reinhartz-Berger, and D. Dori, "A semantic approach to approximate service retrieval," *ACM Transactions on Internet Technology (TOIT)*, vol. 8, no. 1, pp. 2–es, 2007.

[112] M. Klusch, "Overview of the s3 contest: Performance evaluation of semantic service matchmakers," in *Semantic web services*. Springer, 2012, pp. 17–34.

[113] F. Casati, S. Ilnicki, L. Jin, V. Krishnamoorthy, and M.-C. Shan, "Adaptive and dynamic service composition in eflow," in *International Conference on Advanced Information Systems Engineering*. Springer, 2000, pp. 13–31.

[114] F. Lécué, E. Silva, and L. F. Pires, "A framework for dynamic web services composition," in *Emerging Web Services Technology, Volume II*. Springer, 2008, pp. 59–75.

[115] E. G. da Silva, L. F. Pires, and M. van Sinderen, "A-dynamicos: a flexible framework for user-centric service composition," in *2012 IEEE 16th International Enterprise Distributed Object Computing Conference*. IEEE, 2012, pp. 81–92.

[116] K. Fujii and T. Suda, "Semantics-based dynamic web service composition," *International Journal of Cooperative Information Systems*, vol. 15, no. 03, pp. 293–324, 2006.

[117] S. Kona, A. Bansal, M. B. Blake, and G. Gupta, "Generalized semantics-based service composition," in *2008 IEEE International Conference on Web Services*. IEEE, 2008, pp. 219–227.

[118] B. Srivastava and J. Koehler, "Web service composition-current solutions and open problems," in *ICAPS 2003 workshop on Planning for Web Services*, vol. 35, 2003, pp. 28–35.

[119] E. Sirin, B. Parsia, D. Wu, J. Hendler, and D. Nau, "Htn planning for web service composition using shop2," *Journal of Web Semantics*, vol. 1, no. 4, pp. 377–396, 2004.

[120] M. Pistore, A. Marconi, P. Bertoli, and P. Traverso, "Automated composition of web services by planning at the knowledge level," in *IJCAI*, vol. 19, 2005, pp. 1252–1259.

[121] S. McIlraith and T. C. Son, "Adapting golog for composition of semantic web services," *Kr*, vol. 2, no. 200, p. 2, 2002.

[122] S. R. Ponnekanti and A. Fox, "Sword: A developer toolkit for web service composition," in *Proc. of the Eleventh International World Wide Web Conference, Honolulu, HI*, vol. 45, 2002.

[123] M. Klusch, A. Gerber, and M. Schmidt, "Semantic web service composition planning with owls-xplan, agents and the semantic web," in *2005 AAAI Fall Symposium Series, Arlington, Virginia, USA, 4th-6th November*, 2005.

[124] O. Hatzi, D. Vrakas, M. Nikolaidou, N. Bassiliades, D. Anagnostopoulos, and I. Vlahavas, "An integrated approach to automated semantic web service composition through planning," *IEEE Transactions on Services Computing*, vol. 5, no. 3, pp. 319–332, 2011.

[125] J. Fernández-Olivares, T. Garzón, L. Castillo, Ó. García-Pérez, and F. Palao, "A middleware for the automated composition and invocation of semantic web services based on temporal htn planning techniques," in *Conference of the Spanish Association for Artificial Intelligence*. Springer, 2007, pp. 70–79.

[126] F. Lécué and A. Léger, "A formal model for semantic web service composition," in *International semantic web conference*. Springer, 2006, pp. 385–398.

[127] D. V. McDermott, "Estimated-regression planning for interactions with web services." in *AIPS*, vol. 2, 2002, pp. 204–211.

[128] J. Rao and X. Su, "A survey of automated web service composition methods," in *International Workshop on Semantic Web Services and Web Process Composition*. Springer, 2004, pp. 43–54.

[129] H. Q. Yu, S. Dietze, N. Li, C. Pedrinaci, D. Taibi, N. Dovrolls, T. Stefanut, E. Kaldoudi, and J. Domingue, "A linked data-driven & service-oriented architecture for sharing educational resources," 2011.

[130] S. Stadtmüller and A. Harth, "Towards data-driven programming for restful linked data," in *Workshop on programming the semantic web (iswc'12)*, 2012.

[131] A. Khalili, A. Loizou, and F. van Harmelen, "Adaptive linked data-driven web components: Building flexible and reusable semantic web interfaces," in *European semantic web conference*. Springer, 2016, pp. 677–692.

[132] D. Serrano, E. Stroulia, D. Lau, and T. Ng, "Linked rest apis: a middleware for semantic rest api integration," in *2017 IEEE International Conference on Web Services (ICWS)*. IEEE, 2017, pp. 138–145.

[133] R. Taelman, M. Vander Sande, and R. Verborgh, "Bridges between graphql and rdf," in *W3C Workshop on Web Standardization for Graph Data. W3C*, 2019.

[134] T. Durieux, Y. Hamadi, and M. Monperrus, "Fully automated html and javascript rewriting for constructing a self-healing web proxy," in *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, 2018, pp. 1–12.

[135] P. Kaminski, M. Litoiu, and H. Müller, "A design technique for evolving web services," in *Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research*. IBM Corp., 2006, p. 23.

[136] R. D. Banker, G. B. Davis, and S. A. Slaughter, "Software development practices, software complexity, and software maintenance performance: A field study," *Management science*, vol. 44, no. 4, pp. 433–450, 1998.

[137] I. Koren and R. Klamma, "The exploitation of openapi documentation for the generation of web frontends," in *Companion Proceedings of the The Web Conference 2018*. International World Wide Web Conferences Steering Committee, 2018, pp. 781–787.

[138] R. Hervás and J. Bravo, "Towards the ubiquitous visualization: Adaptive user-interfaces based on the semantic web," *Interacting with Computers*, vol. 23, no. 1, pp. 40–56, 2011.

[139] C. Rodríguez, M. Baez, F. Daniel, F. Casati, J. C. Trabucco, L. Canali, and G. Percannella, "Rest apis: a large-scale analysis of compliance with principles and best practices," in *International conference on web engineering*. Springer, 2016, pp. 21–39.

[140] S. Sohan, C. Anslow, and F. Maurer, "A case study of web api evolution," in *2015 IEEE World Congress on Services*. IEEE, 2015, pp. 245–252.

[141] J. Li, Y. Xiong, X. Liu, and L. Zhang, "How does web service api evolution affect clients?" in *2013 IEEE 20th International Conference on Web Services*. IEEE, 2013, pp. 300–307.

[142] S. Wang, I. Keivanloo, and Y. Zou, "How do developers react to restful api evolution?" in *International Conference on Service-Oriented Computing*. Springer, 2014, pp. 245–259.

[143] D. Benslimane, S. Dustdar, and A. Sheth, "Services mashups: The new generation of web applications," *IEEE Internet Computing*, vol. 12, no. 5, 2008.

[144] R. e. a. Alarcon, "Rest web service description for graph-based service discovery," in *International Conference on Web Engineering*. Springer, 2015, pp. 461–478.

[145] M. C. e. a. Paulk, "Capability maturity model, version 1.1," *IEEE software*, vol. 10, no. 4, pp. 18–27, 1993.

[146] M. Fowler, "Richardson maturity model," [Online]. [Accessed: 22-Jun-2021]. https://martinfowler.com/articles/richardsonMaturityModel.html.

[147] "Soha," 02 2010, [Online]. https://tinyurl.com/ya43vefk. [Accessed: 16- Jan- 2019].

[148] R. Mitra, "Rapido: a sketching tool for web api designers," in *Proceedings of the 24th International Conference on World Wide Web*, 2015, pp. 1509–1514.

[149] S. Schreier, "Modeling restful applications," in *Proceedings of the second international workshop on restful design*, 2011, pp. 15–21.

[150] I. Zuzak, I. Budiselic, and G. Delac, "Formal modeling of restful systems using finite-state machines," in *International Conference on Web Engineering*. Springer, 2011, pp. 346–360.

[151] S. Parastatidis, J. Webber, G. Silveira, and I. S. Robinson, "The role of hypermedia in distributed system development," in *Proceedings of the First International Workshop on RESTful Design*. ACM, 2010, pp. 16–22.

[152] R. Tuchinda, C. A. Knoblock, and P. Szekely, "Building mashups by demonstration," *ACM Transactions on the Web (TWEB)*, vol. 5, no. 3, pp. 1–45, 2011.

[153] M. Masse, *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. " O'Reilly Media, Inc.", 2011.

[154] J. Seidenberg and A. Rector, "Web ontology segmentation: analysis, classification and use," in *Proceedings of the 15th international conference on World Wide Web*, 2006, pp. 13–22.

[155] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[156] M. M. Romero, J. M. Vázquez-Naya, C. R. Munteanu, J. Pereira, and A. Pazos, "An approach for the automatic recommendation of ontologies using collaborative knowledge," in *International Conference on Knowledge-Based and Intelligent Information and Engineering Systems*. Springer, 2010, pp. 74–81.

[157] M. d'Aquin, C. Baldassarre, L. Gridinoc, M. Sabou, S. Angeletou, and E. Motta, "Watson: Supporting next generation semantic web applications," 2007.

[158] C. Anutariya, R. Ungrangsi, and V. Wuwongse, "Sqore: A framework for semantic query based ontology retrieval," in *International Conference on Database Systems for Advanced Applications*. Springer, 2007, pp. 924–929.

[159] J. Z. Pan, E. Thomas, and D. Sleeman, "Ontosearch2: Searching and querying web ontologies," *Proc. of WWW/Internet*, vol. 2006, pp. 211–218, 2006.

[160] E. Ashley-Dejo, S. Ngwira, and T. Zuva, "A survey of context-aware recommender system and services," in *2015 International Conference on Computing, Communication and Security (ICCCS)*. IEEE, 2015, pp. 1–6.

[161] N. Kolbe, S. Kubler, J. Robert, Y. Le Traon, and A. Zaslavsky, "Linked vocabulary recommendation tools for internet of things: a survey," *ACM Computing Surveys (CSUR)*, vol. 51, no. 6, pp. 1–31, 2019.

[162] F. Zhang, J. Cheng, and Z. Ma, "A survey on fuzzy ontologies for the semantic web," *The Knowledge Engineering Review*, vol. 31, no. 3, pp. 278–321, 2016.

List of publications

[1] A. Cheron, J. Bourcier, O. Barais, and A. Michel, "Comparison matrices of semantic restful apis technologies," in *International Conference on Web Engineering*. Springer, 2019, pp. 425–440.

[2] A. Cheron, J. Bourcier, O. Barais, D. E. Khelladi, and A. Michel, "[rejected] evolvable-by-design: Robust web ui clients to evolving rest apis," *Journal of Systems and Software*, vol. none, no. none, 2021.

DOCTORAT BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

Titre : Conception, maintenance et évolution non-cassante des API REST

Mot clés : Web, API, REST, co-évolution logicielle, comparaison, Web sémantique

Résumé : Le World Wide Web est principalement composé de deux types de composants applicatifs : les applications et les services. Les applications, qu'elles soit des applications mobiles ou Web, c-a-d ayant vocation à être utilisées depuis un navigateur, ont en commun qu'elles sont une sorte de texte à trous et communiquent avec les services pour personnaliser l'application pour chaque utilisateur. C'est donc le service qui possède et gère les données. Pour rendre possible cette communication, les services rendent disponible des APIs suivant l'architecture REST.

La gestion du cycle de vie d'une API REST est alors un élément central du développement de systèmes dans le Web. La première étape de ce cycle de vie est la définition des exigences d'une API (fonctionnalité et propriétés logicielles). Ensuite, les technologies qui permettront de la concevoir, l'implémenter et la documenter sont choisies. Elle est ensuite implémentée et documentée puis mise en ligne. Dès lors, des applications peuvent l'utiliser. S'ensuit alors une phase de maintenance et d'évolution de l'API, dans laquelle les bogues sont corrigés et les fonctionnalités évolues pour s'adapter aux changements des attentes de ses utilisateurs.

Dans cette thèse, nous passons en revue les méthodes et technologies qui accompagnent le développeur au cours de ce cycle de vie. Nous identifions deux challenges restant sans réponse.

Premièrement, il existe de nombreuses technologies pour créer et documenter une API. Choisir les technologies les plus pertinentes pour un projet est une tâche difficile. Comme première contribution de cette thèse, nous établissons des critères permettant de comparer ces technologies. Ensuite, nous utilisons ces critères pour comparer les technologies existantes et proposons ainsi trois matrices de comparaison. Enfin, pour simplifier cette selection, nous avons développé un assistant open-source disponible sur le Web, qui guide le développeur dans son choix.

Le second challenge que nous avons identifié est lié à la maintenance et l'évolution des APIs REST. La littérature existante ne permet pas de faire évoluer une API REST librement, sans risquer de rendre inopérantes les applications qui s'en servent (leurs clients). La seconde contribution de ce travail est une nouvelle approche à la co-évolution des APIs REST et de leurs clients. Nous avons identifié qu'en suivant 7 règles régissant la documentation de l'API et les données qu'elles renvoient en répondant à ses clients, il est possible de créer des interfaces utilisateur Web capable de s'adapter à la majorité des évolutions des APIs REST sans produire de bogues, ni les casser et sans même nécessiter la modification de leur code source.

Title: Design, maintenance and break-free evolution of REST APIs

Keywords: Web, API, REST, software co-evolution, comparison, Semantic Web

Abstract: The World Wide Web is mainly composed of two types of application components: applications and services. Applications, whether they are mobile or Web applications, i.e. intended to be used from a browser, have in common that they are a kind of text to holes and communicate with the services to customize the application for each user. It is therefore the service that owns and manages the data. To make this communication possible, the services offer APIs following the REST architecture.

The management of the life cycle of a REST API is then a central element of the development of systems on the Web. The first step in this life cycle is the definition of the requirements of an API (functionality and software properties). Then, the technologies that will allow it to be designed, implemented, and documented are chosen. It is then im-

plemented and documented and put online. From then on, applications can use it. Then follows a phase of maintenance and evolution of the API, in which bugs are fixed and functionalities evolve to adapt to the changes of its users' expectations.

In this thesis, we review the methods and technologies that accompany the developer during this life cycle. We identify two open challenges.

First, there are many technologies for creating and documenting an API. Choosing the most relevant technologies for a project is a difficult task. As a first contribution of this thesis, we establish criteria to compare these technologies. Then, we use these criteria to compare existing technologies and propose three comparison matrices. Finally, to simplify this selection, we have developed an open-source wizard available on the Web, which guides the developer in his choice.

The second challenge we have identified is related to the maintenance and evolution of REST APIs. The existing literature does not allow a REST API to evolve freely, without the risk of breaking the applications that use it (their clients). The second contribution of this work is a new approach to the co-evolution of REST APIs and their clients. We have identified that by following 7 rules governing the documentation of the API and the data they return in response to its clients, it is possible to create Web user interfaces capable of adapting to the majority of evolutions of REST APIs without producing bugs, nor breaking them and without even requiring the modification of their source code.