Evolvable-by-design clients of REST APIs: automated run-time evolution with no code change

Abstract—**Web APIs have their own characteristics (heterogeneity, use of a dynamically typed language on the client side, cost of redeployment, lack of access to the API implementation) which tend to make the classical co-evolution techniques between a client and an API inapplicable. In practice, this leads to numerous guides to help API designers respect best practices regarding the evolution of their Web APIs. These best practices are generally conservative techniques that strongly limit the evolution of an API. In this article, we revisit this challenge of supporting Web API evolution without breaking clients. First, we propose an extended version of a taxonomy of Web API evolution. We also propose to decrease the coupling between the client and its API by relying on the ability to have semantically rich API descriptors in the world of Web APIs. In particular, we explore what structural and contextual information should be added in these API descriptors in order to design 'evolvable by design' clients (i.e., that would be able to adapt to API evolution, at run-time, without a developer's intervention). We validate our approach through a case study that imitates Jira and implements 110 API evolutions. As a result, with our approach leveraging on machine-interpretable semantic documentation, we were able to design a client application that adapts to 25 out of the 27 kinds of API evolutions, at run-time, without updating its code.**

Index Terms—**Software Evolution, Evolvable by Design, REST API, User Interface Evolution, Web API Documentation**

I. Introduction

When developers evolve an API, all the clients of this API must also be maintained [1]. Indeed, the evolution of an API frequently introduces breaking changes that must be fixed by developers [2]. This problem has been widely addressed by the software engineering community [3]–[6]. In particular, co-evolving the client of a library that evolved. However, these works are not well suited for Web APIs and the user interfaces (UIs) that use them, which have some peculiarities. Among the important characteristics of the Web world, we can cite:

- A strong technological heterogeneity between the UI and the server;
- The common use of dynamically typed programming language to develop the UI parts, that make static analysis work more complex;
- The absence of access to the source code of the APIs server. We usually have, at best, the new interface contract defined in standards such as OpenAPI, at worst the evolution of the Web API documented on a web page;
- The cost of a UI redeployment within industrial projects [7]. This creates a barrier to automatic co-evolution and automatic program repair techniques that need to modify the UI's code when the API evolves, which requires application redeployment.

In this context, existing works proposed adapter generation techniques to maintain backward compatibility with the initial API [8]. These techniques remain effective in the field of Web APIs but do not allow the UI to handle the addition of required parameters to existing methods of the API. Other works have proposed static analysis techniques to observe changes in the library code in order to use automatic client code repair techniques [5]. However, this is ineffective with dynamically typed client code. Moreover, such techniques that modify the client code require a complete redeployment of the UI. This operation remains costly in many modern software companies. It was reported that a redeployment takes from one week up to six months in 66% of the corporations [7].

Thus, the classical techniques to avoid breaking the UI when the Web API evolves are not fully adapted to the peculiarities of this domain [9]. The result is a huge number of guides provided by the major Web companies to manage the evolutions of their Web APIs. Generally, they drastically reduce the accepted evolutions [10], [11].

In this paper, we tackle the well-known problem of REST API[1] evolution from a novel perspective by making the Web UI *evolvable-by-design*. The aim is for Web UI to be able to evolve on their own to typical API evolutions without the intervention of developers.

We define *evolvable-by-design* as the ability of a UI to evolve along the four following properties of API evolution, at run time, 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.

To do so, we propose to decrease the coupling between the Web UI and its Web API by relying on the ability to have semantically rich API descriptors in the world of Web APIs. Thus, as a main contribution of this paper, we investigate **What structural and contextual information about the API should be available to the UI to enable the creation of *evolvable-by-design* user interfaces?**

To answer this, we first revisit the existing taxonomy of evolutions that can be found in the world of Web APIs. After conducting an interview with a CTO of ANONYM

[1]In the rest of the paper, for simplicity, we refer to REST API as API.

company that developed 200 APIs in the last ten years, we identified five new kinds of API evolutions that complement others studies [2], [12], [13]. 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*.

We validate our approach on an application that imitates the project management software Jira. Its API counts up to 28 operations and implements 110 evolutions, including 59 breaking changes. We compare the ability of two UIs to follow the evolutions of the API at run time without changing their code. One UI was built with traditional methods while the other leverages our approach and is *evolvable-by-design*. As a result, the former could adapt to 0/110 evolutions. The latter could adapt to 100/110 evolutions including 57/59 breaking changes. Furthermore, the implementation of the first version of the *evolvable-by-design* UI client required less code than the traditional version (1395 LoC vs 1623 LoC).

The remainder of this paper is as follows. Section II discusses the motivation of this work. The details of our approach are given in section III. Section IV evaluates this approach. Section V presents the related work, followed by the Section VI which concludes the paper.

II. Motivating Example and Background

A. 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* [2], a collaboration tool that organizes projects into boards. It helps teams in managing their tasks on the Web by manipulating lists and cards similarly as 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 evolve, the UI will always have to use it properly in order to offer the aforementioned functionalities.

Figure 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 UI on the bottom part (B and D). The operations to modify cards were omitted.

As the section A of Figure 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, and the button to delete a card is visible after the user clicks it to see its details.

[2]https://trello.com

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 access 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 redeployed in production, which is time-consuming and costly, as we mentioned before.

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 paper. Thus, in the rest of the paper we present in details our approach, that enables this property.

B. REST APIs Evolutions

As observed by Sohan et al., who studied public, widely-used, REST APIs, they tend to evolve frequently [13], particularly the young APIs that companies keep internally. Hence, little research studied how REST APIs evolve. A first work [12] identified 16 changes in 2013 and was extended in 2015 [13] with 6 other patterns. Another work was done by Wang et al. in 2014 [14]. Table I lists these 22 evolutions where the breaking and non-breaking changes are differentiated. Non-breaking changes relate to behavioral element. While they don't break at compile time, they may create inconsistent behaviors. For example, 26 (*"The set of operations to execute to achieve a business process changed (non-breaking)"*) can lead the ordering process of an e-commerce website to become impossible to complete.

However, none of these works studied API evolutions developed and used in companies. Yet, these companies manage more complex access-rights and more advanced business processes than public APIs. Thus, to challenge these results, we interviewed the Chief Technical Officer (CTO) of ANONYM, a company that developed more than 200 APIs, web and mobile applications for more than 100 companies in the last 10 years. He reported that 5 kinds of changes were not listed. These additions to the state of the art are listed in Table I. He reported that: (23) the request method of an operation

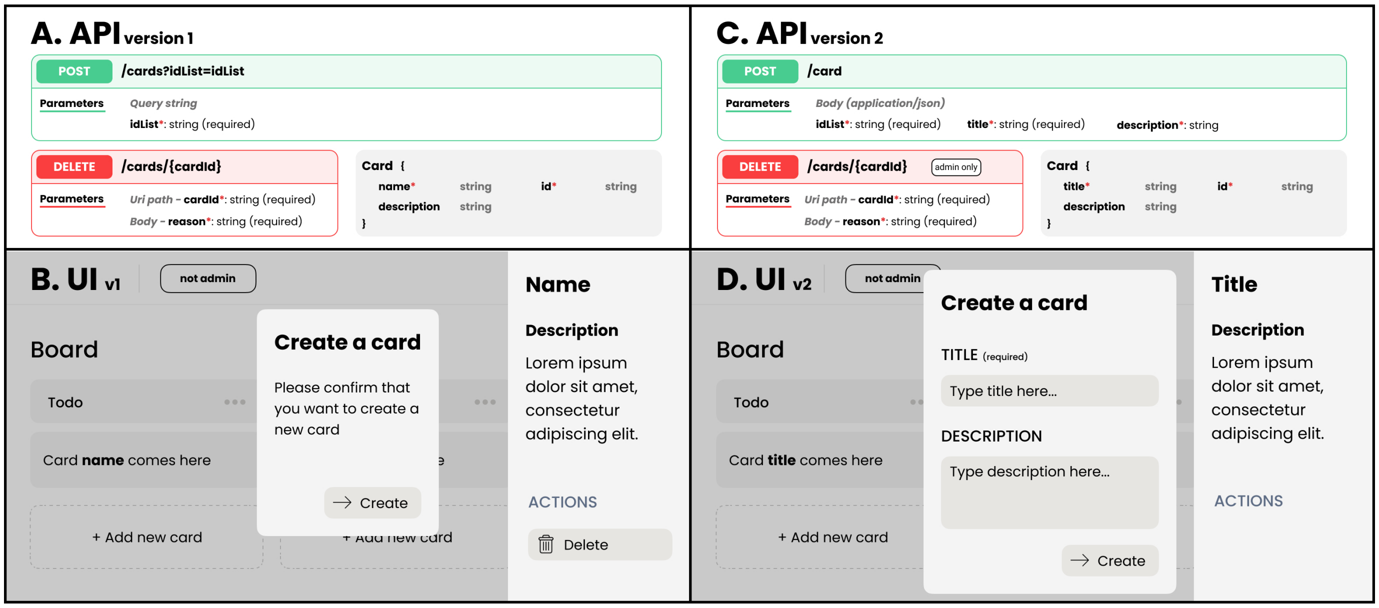

A. API version 1
POST /cards?idList=idList
Parameters Query string
idList*: string (required)
DELETE /cards/{cardId}
Parameters Uri path - cardId*: string (required)
Body - reason*: string (required)
Card { name* string id* string description string }
C. API version 2
POST /card
Parameters Body (application/json)
idList*: string (required) title*: string (required) description*: string
DELETE /cards/{cardId} admin only
Parameters Uri path - cardId*: string (required)
Body - reason*: string (required)
Card { title* string id* string description string }
B. UI v1
not admin
Board
Todo
Card name comes here
+ Add new card
Create a card
Please confirm that you want to create a new card
Create
Name
Description
Lorem ipsum dolor sit amet, consectetur adipiscing elit.
ACTIONS
Delete
D. UI v2
not admin
Board
Todo
Card title comes here
+ Add new card
Create a card
TITLE (required)
Type title here...
DESCRIPTION
Type description here...
Create
Title
Description
Lorem ipsum dolor sit amet, consectetur adipiscing elit.
ACTIONS

Fig. 1. Motivating Example

TABLE I
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)

may change to denote a behavioral change; (24) the precondition 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; and (27) any constraint of the input parameters are likely to change, not only the default value or upper bound (14 & 15). In the evaluation, we replayed those new identified API evolutions, and we observed that they are indeed breaking, either the software or some features (see section IV).

III. Approach

A. Approach Overview

This section details our approach to enable the design of evolvable-by-design UIs. To do so, rather than letting developers hard coding 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 be interpreted at run-time by the UI to create correct and up-to-date interactions with the API leveraging a rich API documentation.

Figure 2 gives an example of such Web UI code. It displays the detail of a card and enable its deletion within the application illustrated in the motivation example on Fig. 1. This example is not given to illustrate how evolvable-by-design UIs must be implemented as this is out of the scope of this work. In fact, while we implemented with React, other frameworks can be used. However, the functional logic behind it remains the same. We explain it in the following.

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 use machine-interpretable semantic descriptors designed with OWL (see lines 2, 5, 7, 9, 17 of Fig. 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 library[3]. It does exactly what the developer does when he reads the API documentation to

[3]URL omitted for the double-blind review

```
 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, approach)} />
14     </if>
15   </right-pane>}
16
17 class CardService {
18   static delete (card, userInputs, evolvable) {
19     lib.invokeOperation(DEL_SEMANTICS).on(card).with(userInputs)
20   }
21 }
```

Fig. 2. Web UI code example of the evolvable-by-design component of the right panel of the sections B and D of Figure 1. (Javascript and React v16)

implement, in the UI, the code interacting with the API. Yet, by doing this interpretation itself, at run-time, the UI becomes able to evolve without updating its code.

Thus, when the UI needs to access a field on a 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 at line 5 of Fig. 2. Hence, the developer is not required to update the code when the format of a data changes.

To invoke an operation on the API, the developer should start by assessing its availability (line 7). 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 9). 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 10, 11 and 17).

We now introduce an overview of the approach architecture in Figure 3. It illustrates the steps to display the end UI to a user. They are numbered into parenthesis in the following paragraph and on the figure accordingly.

First, the developer codes the *Custom UI Components and Navigation* similarly as Fig. 2. During this step, (2) he describes the functional requirements (lines 5, 7, 9 and 17) 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 (5).

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 [15] that is not specific to the approach.

B. 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* UIs.

In the following, we follow the original definition of a resource in the REST architecture [16]. Thus, a resource has an URI schema (e.g., `/card/{id}`) and a list of operations.

*1) **Detailed Resource Description (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.

a) Operations description: all operations must be described in details: their URL schema, HTTP verb, header, query and body parameters along with the responses.

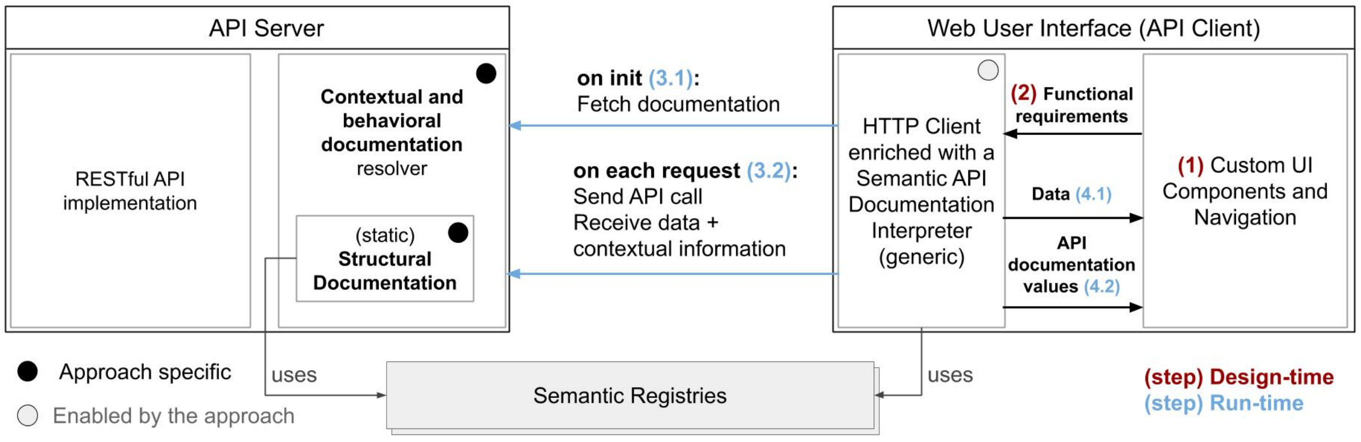
The documentation of the responses must describe the status codes and associated body and headers schema. The potentially linked operations must also be documented. These include the operations that may be invoked on the resource itself, its relations with other resources (e.g. a friend) and control-flow information. The preconditions and post-conditions of the operations must 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 III-C, 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 III-C4.

b) Parameters description: 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].

[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/



API Server
RESTful API implementation
Contextual and behavioral documentation resolver
(static) Structural Documentation
on init (3.1): Fetch documentation
on each request (3.2): Send API call Receive data + contextual information
Web User Interface (API Client)
HTTP Client enriched with a Semantic API Documentation Interpreter (generic)
(2) Functional requirements
Data (4.1)
API documentation values (4.2)
(1) Custom UI Components and Navigation
uses
Semantic Registries
uses
● Approach specific
○ Enabled by the approach
(step) Design-time
(step) Run-time

Fig. 3. Approach Overview

```
operationId: inviteCollaboratorsToProject
parameters:
  collaborators:
    x-research-function-operationId: searchUsers
```

Fig. 4. OpenApi documentation example mentioning a research function

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

*2) **Semantic Description (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 on Fig. 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 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 6 gives an example of a Project ontology.

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

```
 1 schema:
 2   x-@type: '/dictionary#Project'
 3   type: object
 4   properties:
 5     id:
 6       x-@id: '/dictionary#projectId'
 7       x-@type: 'http://purl.org/ontology/mo/uuid'
 8 links:
 9   delete:
10     x-@relation: 'https://schema.org/DeleteAction'
11     parameters: ...
```

Fig. 5. OpenApi documentation example with semantic descriptors

```
@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 ;
```

Fig. 6. Semantic vocabulary example of a Project

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 [17]–[26].

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

*3) **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 if is a completely different data. Hence, each object of a document, no matter its depth, and each link must explicit 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. 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 object and links which data are affiliated to the parent object from those affiliated to themselves.

C. 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).

*1) **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.

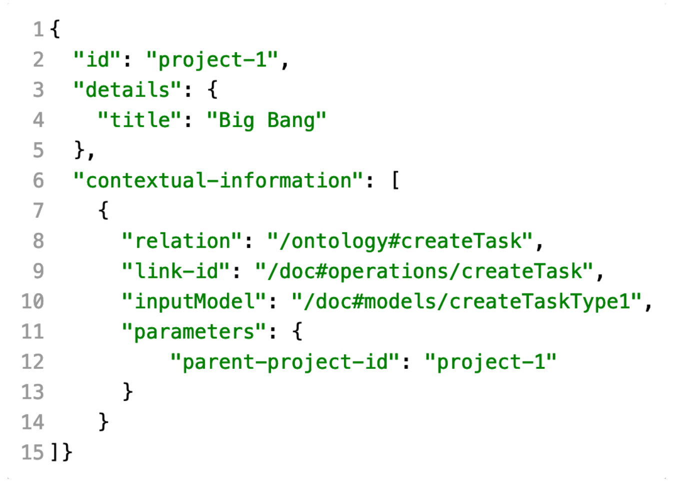
This is required to adapt the UI to the user's access rights without adding descriptors to the documentation.

*2) **All the relevant operations related to the returned resource (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, given a product ordering process, the next operation to invoke to continue the process.

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

*3) **The default value of the operation's input parameters (R6)**:* amongst the default values to provide 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.



```
 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 ]}
```

Fig. 7. An API response example that details a project data and contextual information

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 7.

*4) **The reference to the operation's input model to use when multiple options are listed in the structural documentation (R7)**:* with a similar mechanism used to comply with R6. An example is given at the line 10 of Fig. 7.

D. 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 III-A. 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. 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

[11]https://anonymous.4open.science/r/459cc9bf-9a40-4ddd-a8bd-072cbca0f91b/

TABLE II
REQUIREMENTS OF THE APPROACH ON THE API DOCUMENTATION

Structural Documentation	
R1	**Detailed Resource Description**: 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	**Semantic Description**: 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	**WYSIWYG Documentation**: all operations, resources and properties that the user is not authorized to invoke or see should not be included
R5	**All the relevant operations** related to the returned resource
R6	**The default value of the operation's input parameters**
R7	**The reference to the operation's input model** to use when multiple options are listed in the structural documentation

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 IV.

E. Synthesis and Discussion

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

a) Reuse and limitations of the state-of-the-art: our approach is designed to maximize reuse of state-of-the-art technologies. Indeed, most of the detailed resource's description (R1) can be done with OpenApi or WSDL and the semantic description (R2) with OWL. Also, (R5), (R6) and (R7) are common hypermedia-controls [27]. However, even if possible, not all developers specify this information along their APIs, and none combine them at all. This work further identifies new requirements that OpenApi and WSDL must be extended with, 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 III-B1b. Thus, this combination of existing technologies and the new requirements we add in the current approach enables the powerful design of *evolvable-by-design* UIs.

b) 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 I 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, 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 [12].

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

IV. EVALUATION

This section presents the empirical evaluation we performed. The goal is to study to what extents the approach presented in Section III answers the principal question of the introduction: *What structural and contextual information about the API should be available to the UI to enable the creation of evolvable-by-design UIs?*

Hence, this evaluation is built on 2 research questions.

a) 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 changes.

b) RQ2: Does the implementation of an *evolvable-by-design* Web UI require additional development efforts?

This aims to investigate the trade-off of our approach in terms of development efforts of user interfaces.

A. Data set

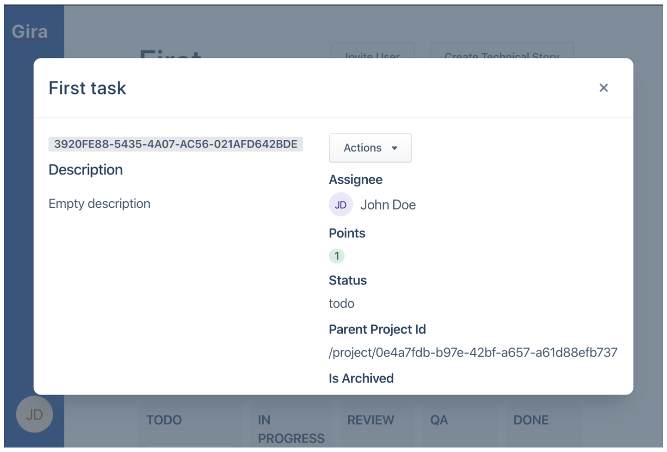
To evaluate the approach, we needed a web application composed of a Web UI and a REST API that implemented all sorts of evolutions from Table I, at least once. Unfortunately, we could not find such kind of open-source project and our industrial partner could not provide one for privacy reasons.

Then, we developed a web application imitating the project management software Jira[12]. 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.

The Figure 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 in 16 versions, (2) an evolvable-by-design UI and (3) a state-of-the-practice UI where the contract with the API is hard coded. 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

[12]Atlassian Jira - Home page - https://www.atlassian.com/software/jira



First task

3920FE88-5435-4A07-AC56-021AFD642BDE

Actions

Description

Empty description

Assignee

JD John Doe

Points

1

Status

todo

Parent Project Id

/project/0e4a7fdb-b97e-42bf-a657-a61d88efb737

Is Archived

Fig. 8. Screenshot of the application developed to evaluate the approach

documentation follows the OpenAPI Specification 3.0.0 that we extended to 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 [13]. 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. Hence, this code can be reused for other UI projects with minimal effort. Therefore, developers or scientists can handily reproduce the experiment or create their own.

B. Experimental Protocol

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

In order to test at least one variant of all kinds of evolutions listed in the taxonomy presented in Table I, we implemented 110 evolutions of the API, including 59 breaking changes and distributed them into 16 versions of the API. For example, 5 and 17 were respectively applied 9 and 3 times.

For each upgrade of the API, we manually evolve the code of the ”traditional UI” to implement the evolutions. On the *evolvable-by-design* UI, we refresh the page and manually verify that all evolutions were automatically integrated while not introducing bugs. Otherwise, we update the code.

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

a) the changes that are automatically supported by the UIs without causing bugs: by manually testing each feature of the interface.

b) the lines of code that are changed to support breaking changes: by summing up the difference between additions and deletions of each commit related to this topic.

[13]Evolutions: https://bit.ly/2UUpvGP - Clients: https://bit.ly/3c1kwcX - Server: https://bit.ly/2XipgqA

TABLE III
OVERALL EVALUATION RESULTS

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

c) the lines of code that are changed to support non-breaking changes: by summing up the difference between additions and deletions of each commit related to this topics.

All the code of the traditional UI is included. On the *evolvable-by-design* UI, the http client enriched with a semantic API documentation interpreter is the only component that is excluded because it is a generic library.

C. Observed results

Table III summarizes the observed results of the evaluation, and more details are given in Table IV.

a) 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 changes that are tested, which **addresses 25 out of the 27 (93%) evolutions of Table I**. On the other hand, 43 out of the 51 non-breaking changes are also addressed **while the traditional UI is unable to evolve to any breaking or non-breaking change**.

The two breaking changes 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 III-E.

On the other hand, among the non-breaking changes, the only kind of changes 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.

By leveraging this latter capacity, the UI would also be able to evolve to the two breaking changes not addressed, with a trade-off on the ergonomics.

From this research question we learn that **this semantic-based approach enables the automatic evolution of the UI, at run-time, to all changes that does not impact the semantics of the API**, which include 25 out of the 27 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 27 kinds of evolutions, and hence, required manual adaptation.

b) RQ2: Does the implementation of an evolvable-by-design Web UI require additional development efforts?

TABLE IV
DETAILED RESULTS OF THE EVALUATION [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 Changes automatically handled by the client																	
Total breaking changes	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 Changes automatically handled by the client																	
Total non-breaking changes	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 changes																	
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 changes																	
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**

First, we observe in Table IV 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 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 to update 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 changes. 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 neither 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 changes, which demands additional cognitive effort and developing new skills. Yet, from these results we argue that the development of an *evolvable-by-design* UI greatly reduces the effort to evolve the UI for the next versions of the API. Thus, we observe that the overall development effort is greatly reduced in the long term.

D. 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 25/27 known kinds of API evolution while the traditional UI evolves to 0/27. Likewise, for non-breaking changes, our approach evolves to 51/59 while the traditional UI evolve to 0/59.

However, the approach brings new sources of mistake. A careful attention must be paid by developers to the following points, even if they are requirements of the approach.

a) 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.

b) 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.

E. Threats to validity

1) Internal Validity: The two UIs used for the evaluation were implemented by one author of this paper. It introduces a bias. Thus, to limit its impact, he developed both UIs with the same rigor, and best practices that he follows to develop web applications for big corporations at ANONYM company, where more than 100 web and mobile developers work. 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.

2) External Validity: relates to the number of participants and the technologies involved. Indeed, we tested the approach over one case study and implemented the server and UIs 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 NodeJS. Similarly, the code leveraging the documentation 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[14] and VueJS[15].

[14]https://angular.io/
[15]https://vuejs.org

3) Conclusion Validity: we tested our approach on our own over a single case-study, which is a project management tool similar to Jira. From our experimentation the results of the approach seem good and promising. However, a similar experimentation should be done on several case studies with tens of developers, excluding ourselves, using different technologies and languages to confirm these encouraging results.

V. State of the art & Related Work

In order to face the problems of API evolution, many techniques have been proposed. The two most pragmatic still widely used in the industry limits the evolution. Indeed, a first approach aims to keep several versions of its Web API online [9]. This approach proposes to use mechanisms to declare the future deprecations as part of the API. For example, the ”Chain of Adapters” introduced in [28] 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 [10], [11].

The issue of co-evolving an API and its client has been widely studied by the software maintenance community. 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 [1], [4]–[6]. As a representative solution, [29] proposed an Eclipse plugin to automatically update the client code when the API changes. These approaches have several limitations with respect to the evolution of a Web API. Firstly, the Web world is highly heterogeneous in terms of programming language; 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. To fill this last gap, several approaches have proposed to automatically synthesize an adapter [8], [30] 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.* [2] 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 approach. Another approach from Leitner et. al. uses a proxy to route API calls to the requested version. VRESCo [31] uses plain string version names such as latest. In [32], 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.

Fully synthesizing the UI based on API description seems to be a relevant approach to cope with Web API evolution challenge. Then, the Swagger editor[16] is an example of such approach. However, code generators proved to be complex to customize and evolve to integrate specific ergonomics project [33]. Compared to generative approach, we use an interpreter within the client to adapt to the Web API evolution.

No matter whether one chooses a generative or interpreter-based approach, an important question remains what information the Web API must provide in order to allow the client to adapt its API. Hervas et. al. [34] propose to use Semantic Web technologies in order to gain semantic information about the user context, to adapt the UI. However, API evolutions are not considered in this work. It may be used to extend this work by inferring the content and components to use for unforeseen additions to APIs. Another approach is LD-Reactor [20]. It builds a component-based UI from a configuration indicating which Linked Data-sets to retrieve. Thus, the UI is aware of data requirements. Similarly to our approach, designers can precisely design UI components. However, the use of SPARQL endpoints through RESTful API prevent clients to adapt to API evolution because graph, database-like, queries are sent to one single endpoint per API. Koren *et al.* propose a graphical tool to build a UI with already implemented web components from an OpenAPI documentation [35]. Thus, designers can design high-quality components but the interface must be generated each time the API evolves.

To the best of our knowledge, no existing approach proposed to keep clients compliant with evolving APIs at run-time. Our approach proposes this and let clients be *evolvable by design* in interpreting semantically-annotated API documentation.

VI. Conclusion

This paper proposed an approach that enables the design of *evolvable-by-design* user interfaces client of REST APIs. They are UIs that evolve automatically to the changes of the APIs, at runtime, and without modifying their code. We first extended the existing taxonomy of Web API evolutions by five new changes. After that, we proposed to decrease the coupling between the client and its API by relying on the ability to have semantically rich REST API documentations. In particular, we studied what structural and contextual information should be documented in the APIs for clients to be *evolvable-by-design*.

Our approach was evaluated over 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 25/27 kinds of evolutions at runtime without modifying its code, which represents 99% of real-world evolutions, which outperforms traditional UI clients. 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.

[16]Swagger Editor - https://editor.swagger.io

References

[1] 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.

[2] 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.

[3] T. Espinha, A. Zaidman, and H. Gross, "Web API growing pains: Loosely coupled yet strongly tied," *J. Syst. Softw.*, vol. 100, pp. 27–43, 2015. [Online]. Available: https://doi.org/10.1016/j.jss.2014.10.014

[4] 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.

[5] 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.

[6] 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.

[7] N. Forsgren, D. Smith, J. Humble, and J. Frazelle, "Accelerate State of DevOps 2019," 2019. [Online]. Available: https://cloud.google.com/devops/state-of-devops/

[8] A. Bennaceur, P. Inverardi, V. Issarny, and R. Spalazzese, "Automated Synthesis of CONNECTors to support Software Evolution," Jan. 2012, http://ercim-news.ercim.eu/en88/. [Online]. Available: https://hal.inria.fr/hal-00662058

[9] 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.

[10] Adidas. Adidas - rules for extending. [Online]. Available: https://adidas.gitbook.io/api-guidelines/general-guidelines/rules-for-extending

[11] Zalando. Zalando restful api and event scheme guidelines. [Online]. Available: https://opensource.zalando.com/restful-api-guidelines/

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] R. T. Fielding, *Architectural styles and the design of network-based software architectures*. University of California, Irvine Irvine, 2000, vol. 7.

[17] S. A. McIlraith, T. C. Son, and H. Zeng, "Semantic web services," *IEEE intelligent systems*, vol. 16, no. 2, pp. 46–53, 2001.

[18] F. Giunchiglia, M. Yatskevich, and P. Shvaiko, "Semantic matching: Algorithms and implementation," in *Journal on data semantics IX*. Springer, 2007, pp. 1–38.

[19] L. Li and I. Horrocks, "A software framework for matchmaking based on semantic web technology," *International Journal of Electronic Commerce*, vol. 8, no. 4, pp. 39–60, 2004.

[20] 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.

[21] M. Paolucci, T. Kawamura, T. R. Payne, and K. Sycara, "Semantic matching of web services capabilities," in *International semantic web conference*. Springer, 2002, pp. 333–347.

[22] K. Sycara, M. Paolucci, A. Ankolekar, and N. Srinivasan, "Automated discovery, interaction and composition of semantic web services," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 1, no. 1, pp. 27–46, 2003.

[23] A. P. Sheth, K. Gomadam, and A. H. Ranabahu, "Semantics enhanced services: Meteor-s, sawsdl and sa-rest," *Bulletin of the Technical Committee on Data Engineering*, vol. 31, no. 3, p. 8, 2008.

[24] L. Obrst, "Ontologies for semantically interoperable systems," in *Proceedings of the twelfth international conference on Information and knowledge management*, 2003, pp. 366–369.

[25] T. Heath and C. Bizer, "Linked data: Evolving the web into a global data space," *Synthesis lectures on the semantic web: theory and technology*, vol. 1, no. 1, pp. 1–136, 2011.

[26] P. Jain, P. Hitzler, A. P. Sheth, K. Verma, and P. Z. Yeh, "Ontology alignment for linked open data," in *International semantic web conference*. Springer, 2010, pp. 402–417.

[27] 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.

[28] 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.

[29] 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.

[30] 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

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.