Comparison Matrices of Semantic RESTful APIs Technologies

Antoine Cheron, Johann Bourcier, Olivier Barais, Antoine Michel

Comparison Matrices of Semantic RESTful APIs Technologies

Antoine Cheron[1][0000−0003−1857−6799], Johann Bourcier[2][0000−0003−2947−9150], Olivier Barais[2][0000−0002−4551−8562], and Antoine Michel[1]

[1] Fabernovel, 44-48 rue Saint-Lazare F-75009 Paris
[2] Univ Rennes, Inria, CNRS, IRISA 263 Avenue General Leclerc, F-35000 Rennes

Abstract. Semantic RESTful APIs combine the power of the REST architectural style, the Semantic Web and Linked Data. They picture a world in which Web APIs are easier to browse and more meaningful for humans while also being machine-interpretable, turning them into platforms that developers and companies can build on. We counted 36 technologies that target building such APIs. As there is no one-size-fits-all technology, they have to be combined. This makes selecting the appropriate set of technologies to a specific context a difficult task for architects and developers. So, how the selection of such a set of technologies can be eased? In this paper we propose three comparison matrices of Semantic RESTful APIs enabling technologies. It is based on the analysis of the differences and commonalities between existing technologies. It intends to help developers and architects in making an informed decision on the technologies to use. It also highlights the limitations of state-of-the-art technologies from which open challenges are derived.

Keywords: Hateoas· Semantic REST· Comparison· Linked Data· Web

1 Introduction

Today, RESTful APIs [19] have become the de-facto standard for building web applications. The main reason behind this popularity lies in the appropriate trade-off between the facility to build such applications and the benefits provided by this approach in such an opened large-scale distributed system: evolutivity, scalability and loose-coupling. However, 95% of APIs are not RESTful [13] as they claim.

Until today, no single standard has emerged to design truly RESTful APIs. Consequently, software architects are facing the challenge of selecting the right technologies for the design and implementation of these systems. Typically, a software architect has to select the right interface description language (IDL), interchange format and framework to ease the development of such APIs.

In addition, a new trend has recently emerged to create RESTful APIs that carry their own semantics, they are called Semantic RESTful APIs [15]. It is a vision that proposes to make fully REST-compliant APIs compatible with the Semantic Web [3] and Linked Data [4]. From our experience at FABERNOVEL, we found that building such APIs does not require much more effort than truly RESTful systems, whereas it offers great benefits, such as loose-coupling, automated API mash-ups [2], machine-interpretability and very powerful querying.

However, the design of semantic RESTful APIs considerably increases the complexity for the architect to choose the appropriate technology. Indeed, the specific criteria and properties to be taken into account are not explicit when choosing an IDL, an interchange format and a framework. 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 paper, 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 paper are:

- three comparison matrices of interchange formats, interface description languages and frameworks that help choosing 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.

Using these comparison matrices, we illustrate their usage on an industrial case and draw the outline of a research road-map to ease the adoption of Semantic RESTful APIs in the industry.

The remainder of this paper is organized as follows. Section 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 5 discusses the role of the existing frameworks to build Semantic REST APIs.

2 Background

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

Semantic RESTful services Combining REST with Semantic Web and Linked Data is a promising path since it enables the description of APIs that can change without breaking client applications. These APIs advertise their available state transitions, therefore enabling automatic composition to create high level services [1]. 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.

2.1 Selecting an API functionality level

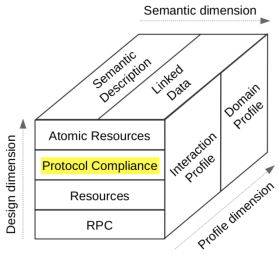
Today, Web systems offer a wide range of functionalities. For example, they may offer multiple media types or a single one, comply with the HTTP protocol or

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 [11, 15].

In companies, architects use them to decide features which 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 [6], which targets building REST APIs. However, we recommend using the WS3 maturity model [15] as it combines the models proposed by Richardson, and SoHA [18], and extends them with semantic and documentation constraints.

The WS3 maturity model In [15], 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. 1.

Fig. 1. WS3 Maturity Model (from [15])



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 verbs 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 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- and post-conditions, business constraints, etc.

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 [15], 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[3]. As another example, a system that supports HATEOAS and provides a swagger-like documentation along with the data is rated D3-S0-P2[4].

[3] D3-S0-P0: Atomic Resources Design, no Semantic Description, no Profile description
[4] D3-S0-P2: Atomic Resources Design, no Semantic Description, Domain Profile

2.2 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. Respecting this constraint may introduce negative properties in the API. Let us consider an API handling insurance contracts which 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 properties 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 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 3.1) and not linked to any WS3 levels.

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[5]. Our key findings are:

[5] https://github.com/AntoineCheron/comparison-matrices-semantic-rest-api-techno

- *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.
- *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 link 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 to Semantic REST APIs. They also highlight that these technologies are not yet mature and give a rough idea of the missing features.

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 helps implementing 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 [6] 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[7].

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 Interface Description Languages

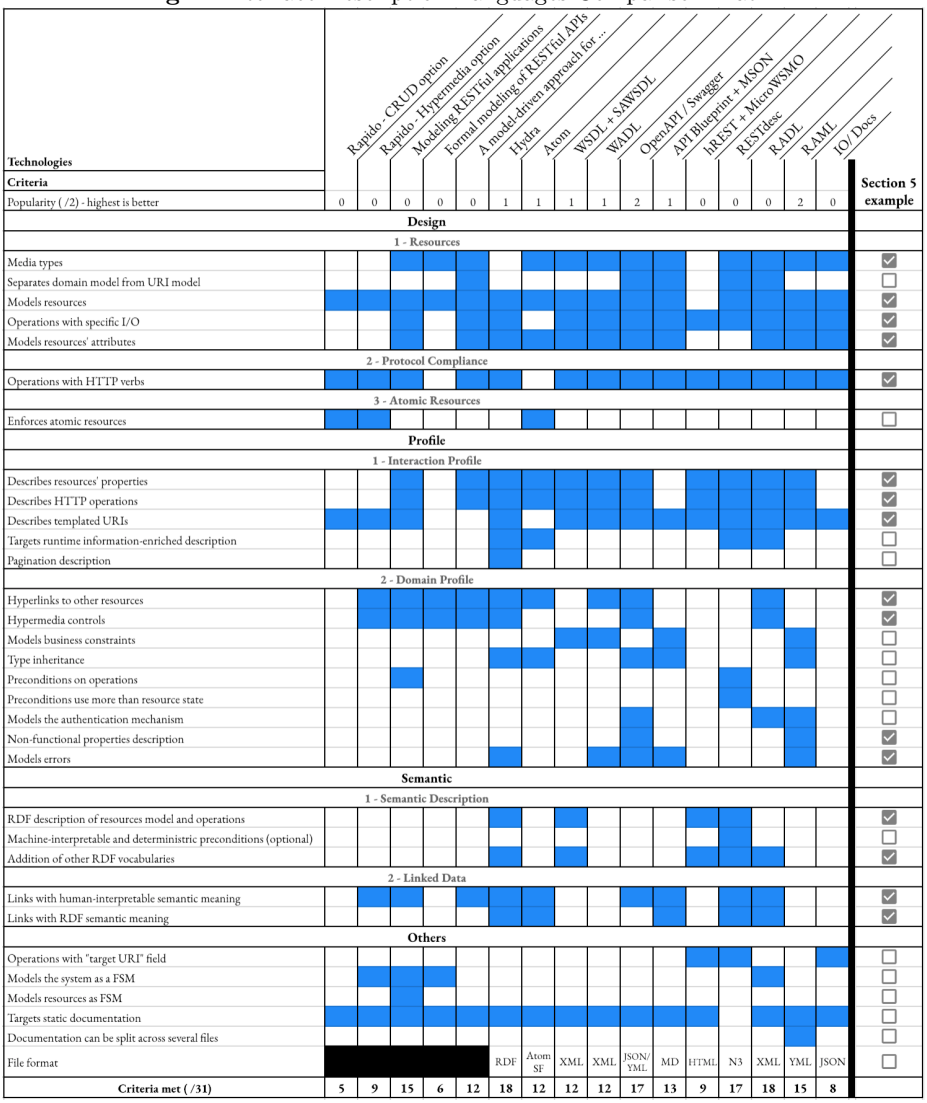
Interface Description Languages (IDLs) provide a vocabulary to document domain, functional and non-functional aspects of an API. We identified 16 candidates that are classified according to 31 criteria in Fig 2. Among them, 4 are meta-models from conference papers [7, 8, 16, 22]. The 11 others are open-source projects or W3C recommendations.

In [8] 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. [16] 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 details, which make them not machine-interpretable. In [22], 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. [7] propose a multi-layered model that separates the domain model from the URI model. However, resources have a fixed model, which prevent them from having one data model per state.

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

[6] http://amundsen.com/hypermedia/hfactor/
[7] https://github.com/AntoineCheron/comparison-matrices-semantic-rest-api-techno

Fig. 2. Interface Description Languages Comparison Matrix



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 to build 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). Ope-

nAPI 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, [16], 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.

3.4 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.

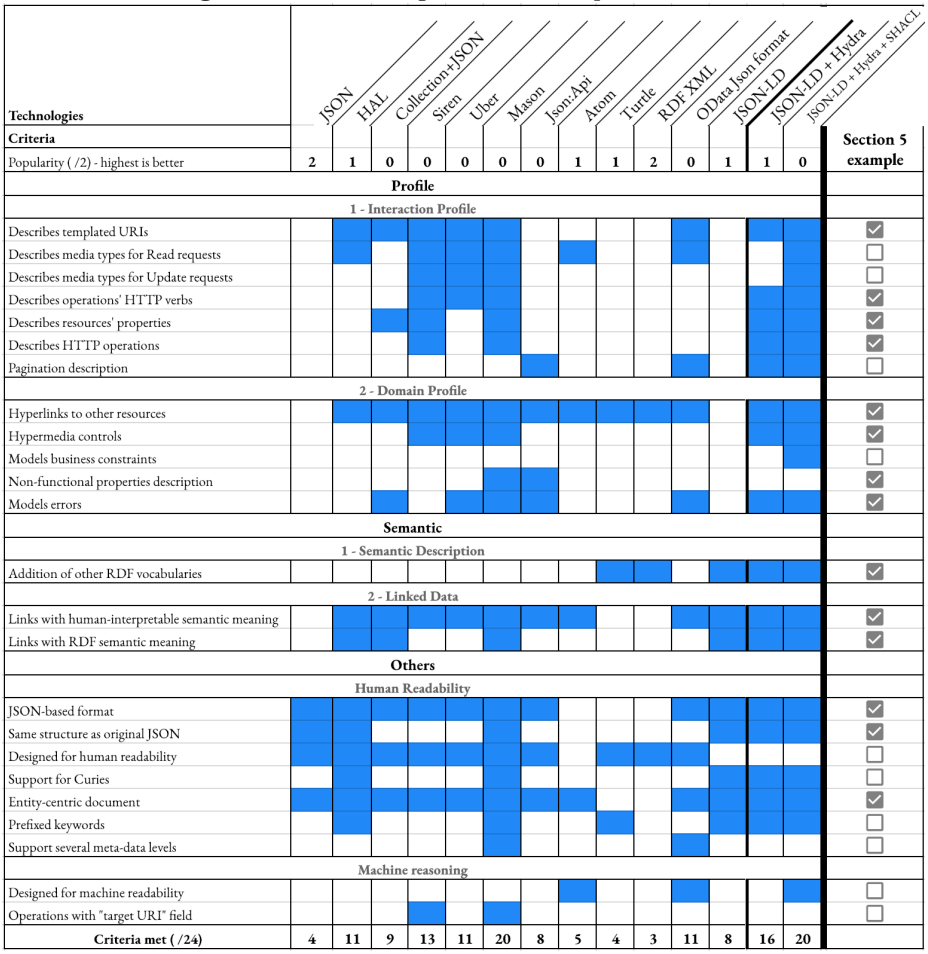
On the other side, when the system to be built have to support a hypermedia interchange format, none is considered as a standard today. We selected 11 candidate technologies, which are classified in Fig. 3 according to 24 criteria. JSON is included for comparison purposes.

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 a RDF format, we selected two vocabularies: Hydra and SHACL, a 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 allow building mature Semantic REST systems. However, this requires additional effort to find relevant vocabularies. On the other hand, non-RDF formats help building 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 advertise the state of the resource even though most scientific approaches we found describe REST APIs as state-machines.

Fig. 3. Data-interchange Formats Comparison Matrix



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 [14] 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 [9] 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. 4 according to 23 criteria.

Fig. 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	
Criteria									**Section 5 example**
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**	

Synthesis Despite the fact that only one framework enforces the *Atomic Resources* constraint, all frameworks allow to reach 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.

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.

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 an 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.

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 companys 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[8] middleware to easily query the whole IS [20]. 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[9] we needed the IDL to enable associating an operation to its own input and output data model.

[8] https://www.hypergraphql.org/
[9] 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.

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 Fig 2, 3, and 4. We then count the number of criteria that were provided by each existing technology. Results are presented in Fig 5, 6 and 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 criteria to be selected. Most of the time, missing features can be implemented afterwards, or proposed to the maintainers of the technologies.

Fig. 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**

Fig. 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**

Fig. 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**

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 setup both Hydra and OpenAPI. OpenAPI because it has 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.

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 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 [14] 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 the need to develop some features by hand. This is why we decided to go with Spring.

Easing the selection of the technologies We have developed an open-source web recommender system[10] in which the user selects the type of technology, the required criteria and finally *score* 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.

5 Discussion

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

First, there is no de-facto neither IETF or W3C standard Interchange Format for building Semantic REST APIs. The technology the more likely to become a standard is JSON-LD, for which a W3C Working Group is active. In addition, 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 will rely on formats that are likely to evolve, which will require additional effort and costs. This reduces the likelihood of framework editors to invest time in developing such features.

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

[10] https://antoinecheron.github.io/morice/

and useful features. Among the possible functionalities, we envision various tools to automate API testing, REST client generation, API gateways, middleware and smarter desktop REST clients. 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.

6 Related Work

In [21], and [15], authors justify the need to provide a semantic description of REST APIs to avoid that programmers who develop client applications have to understand in depth several APIs from several providers. Based on this motivation, they survey academic approaches to add semantic to such APIs description and technique to automatically compose restful services.

In [17] authors present a framework for REST-service integration based on Linked Data models. First, API providers semantically describe their REST services. Second, API consumers express data queries with SPARQL. Then, they use a middleware developed by the authors that automatically compose API calls to respond to data queries with a RDF graph. At the first step, authors needed to find and select a RDF-compatible Interface Description Language, which is precisely the kind of use-case our approach addresses. Because they could not find an existing one fitting their needs, they designed a new one by leveraging existing technologies such as MSM [12], Hydra, RAML and OpenApi.

In [20], Tuchinda *et al.* describe a programming-by-demonstration approach to build mashups by example. Instead of requiring a user to select and customize a set of widgets, the user simply demonstrates the integration task by example. Their approach addresses the problems of extracting data from web sources, cleaning and modeling the extracted data, and integrating the data across sources. It illustrates the benefits of getting meta-data on top of services to improve the definition of mashups and decrease the coupling between information system building blocks and the complexity of developing mature Semantic REST APIs. In [5], Duke *et al.* propose an approach to reduce the complexity for describing, finding, composing and invoking semantic rest services. They mainly provide an approach where they show how they can combine services when they get semantic information.

Other research efforts were done to lower the entry barrier for developing mature Semantic REST APIs. Among them is the semi-automatic annotation of web services as done by Patil *et al.* in [10]. Their contribution could help significantly increase the number of semantically described services, in case their work is open-sourced and updated to support nowadays popular technologies.

7 Findings Summary

In this paper, 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 paper, 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 conditions 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, REST client libraries can lower the coupling with the server and automate tasks such as login, and middleware can automatically create responses from the composition of several APIs.

References

1. Alarcon, R.e.a.: Rest web service description for graph-based service discovery. In: International Conference on Web Engineering. pp. 461–478. Springer (2015)
2. Benslimane, D., Dustdar, S., Sheth, A.: Services mashups: The new generation of web applications. IEEE Internet Computing **12**(5) (2008)
3. Berners-Lee, T.: The semantic web. Scientific American (2001)
4. Berners-Lee, T.: Linked data principles (07 2006), [Online]. Available: https://www.w3.org/DesignIssues/LinkedData.html. [Accessed: 15- Jan- 2019].
5. Duke, A.e.a.: Telecommunication mashups using restful services. In: Towards a Service-Based Internet. pp. 124–135. Springer, Berlin, Heidelberg (2010)
6. Fowler, M.: Richardson maturity model, [Online]. [Accessed: 15- Jan- 2019]. https://martinfowler.com/articles/richardsonMaturityModel.html.
7. Haupt, F., Karastoyanova, D., Leymann, F., Schroth, B.: A model-driven approach for rest compliant service. In: Proceedings of the IEEE International Conference on Web Services (ICWS 2014). pp. 129 – 136. IEEE (2014)
8. Mitra, R.: Rapido: a sketching tool for web api designers. World Wide Web Conference (2015)
9. Parastatidis, S., Webber, J., Silveira, G., Robinson, I.S.: The role of hypermedia in distributed system development. In: Proceedings of the First International Workshop on RESTful Design. pp. 16–22. ACM (2010)
10. Patil, A.A., Oundhakar, S.A., Sheth, A.P., Verma, K.: Meteor-s web service annotation framework. In: Proceedings of the 13th international conference on World Wide Web. pp. 553–562. ACM (2004)
11. Paulk, M.C.e.a.: Capability maturity model, version 1.1. IEEE software **10**(4), 18–27 (1993)
12. Pedrinaci, C., Domingue, J., et al.: Toward the next wave of services: Linked services for the web of data. J. ucs **16**(13), 1694–1719 (2010)
13. Rodríguez, C., et al.: Rest apis: A large-scale analysis of compliance with principles and best practices. In: Bozzon, A., Cudre-Maroux, P., Pautasso, C. (eds.) Web Engineering. pp. 21–39. Springer International Publishing, Cham (2016)

14. Salvadori, I.e.a.: A framework for semantic description of restful web apis. In: Web Services (ICWS), 2014 IEEE International Conference on. IEEE (2014)
15. Salvadori, I., Siqueira, F.: A maturity model for semantic restful web apis. In: 2015 IEEE International Conference on Web Services. pp. 703–710 (June 2015)
16. Schreier, S.: Modeling restful applications. In: Proceedings of the Second International Workshop on RESTful Design. pp. 15–21. WS-REST '11, ACM, New York, NY, USA (2011). https://doi.org/10.1145/1967428.1967434
17. Serrano, D., Stroulia, E., Lau, D., Ng, T.: Linked rest apis: A middleware for semantic rest api integration. In: Web Services (ICWS), 2017 IEEE International Conference on. pp. 138–145. IEEE (2017)
18. Soha (02 2010), [Online]. https://tinyurl.com/ya43vefk. [Accessed: 16- Jan- 2019].
19. T.Fielding, R.: Architectural Styles and the Design of Network-based Software Architectures. Ph.D. thesis, University of California, Irvine (2000)
20. Tuchinda, R., Knoblock, C.A., Szekely, P.: Building mashups by demonstration. ACM Trans. Web **5**(3), 16:1–16:45 (Jul 2011). https://doi.org/10.1145/1993053.1993058
21. Verborgh, R.e.a.: Survey of semantic description of REST APIs. In: REST: Advanced Research Topics and Practical Applications, pp. 69–89. Springer (2014)
22. Zuzak, I.e.a.: Formal modeling of restful systems using finite-state machines. In: Web Engineering. pp. 346–360. Springer, Berlin, Heidelberg (2011)