Development of a tool allowing to create and use JSON schemas so as to enhance the validation of existing projects

CHARLES-ELIE SIMON
Development of a tool allowing to create and use JSON schemas so as to enhance the validation of existing projects

CHARLES-ELIE SIMON

Master’s Thesis at KTH Information and Communication Technology and Applidium
Supervisor: Emilie Paillous
Examiner: Johan Montelius

TRITA-ICT-EX-2017:82
Abstract

A mobile application is typically divided into two sides that communicate with each other: the front-end (i.e. what the user can see and interact with on the phone) and the back-end (the hidden ”server” side, which processes requests from the front-end). Ways to improve their production cycle are constantly investigated by corporations such as Applidium, which is a French startup company specialized in mobile applications. For instance, the firm often has to deal with external back-ends that are not properly documented, which makes the development of products intricate. Furthermore, test and documentation files for certain parts of projects are manually written, which is time consuming, and are all largely based on the same information (back-end descriptions). Hence, this information frequently finds itself scattered in different files, sometimes in different versions. Having identified issues that most regularly disrupt the work of the company’s employees, a certain number of goals to solve these are set, such as, notably, centralizing all back-end-related information into one authoritative source, and automatizing the generation of test and documentation files. A tool (in the form of a web application) allowing users to describe back-ends, called Pericles, is then proposed as the outcome of the master thesis, to deal with the described problems and materialize the defined objectives. Finally, a qualitative evaluation is performed through a questionnaire designed to assess how users feel the tool helps them in their work, which constitutes the metric for this project. The evaluation suggests that the implemented tool is relevant with respect to the fixed goals, and allows to infer its propensity to help Applidium’s developers and project managers by making the development and validation of projects easier.

Keywords: Application programming interface, Representational state transfer, JSON schema, Resource, Route
# Contents

1 Introduction .................................................. 3  
1.1 Background ............................................. 3  
1.2 Problem ............................................... 4  
1.3 Purpose ............................................... 4  
1.4 Goals ................................................ 5  
1.5 Method ................................................. 6  
1.6 Delimitations ......................................... 6  

2 Background .................................................. 7  
2.1 JSON and JSON Schema .................................. 7  
2.2 The HTTP Protocol ...................................... 8  
2.3 What is an API? ......................................... 13  
2.4 Existing work ........................................... 19  

3 Implemented solution ........................................... 22  
3.1 Overview .............................................. 22  
3.2 Software architecture .................................... 22  
3.3 Implementation ......................................... 29  
3.4 Timings and project organization ...................... 40  
3.5 Current limits .......................................... 43  
3.6 Benefits and sustainability .............................. 44  

4 Evaluation .................................................... 46  

5 Conclusion ................................................... 50  

Appendices ....................................................... 55  
A JSON Schemas for the validation API .................. 55  
A.1 In HTTP requests ....................................... 55  
A.2 In HTTP responses ...................................... 56  
B Planned timings at the beginning of the project ....... 58  
C Screenshots .................................................. 59  
D Summary in French ......................................... 62
1 Introduction

This section gives an overview of the problem that is tackled as part of this master thesis, and details the approach taken to solve it.

1.1 Background

Applidium is a startup located in Paris and designs mobile applications which are tailor-made according to the needs of clients. The company works both with iOS and Android, and is reknown for its work for clients in the transport (RATP, for instance, which manages the metro network in Paris) and entertainment (Canal+, which is the French equivalent of HBO) industries.

The design of a mobile application is usually divided between the back-end and the front-end. Roughly, the front-end of a mobile application consists in what the user can see and interact with on his phone (the mobile interface). The back-end is embodied by all the components that are hidden to the user and that aim at, notably, serving its requests (the server part). The technology used by the front-end and the back-end to communicate with each other is HTTP. Basically, users issue requests by interacting with the front-end. These requests are then transmitted to the back-end, and processed. The answers are then forwarded by the back-end to the front-end to make them accessible to the user. The team at Applidium often works with both described aspects (front-end and back-end) during projects, thus overseeing the design of the application in its entirety.

The work of back-end developers at Applidium consists in building APIs. API stands for Application Programming Interface. As suggested by the name, it corresponds to a set of services that software offers to other pieces of software, and can be related to the notion of interface that exists in Java, for instance.

The two development teams at Applidium (one dedicated to back-end development, and the other to the front-end) thus work in synergy. The aim for the back-end team is to provide APIs that can be consumed easily by the front-end developers and cover all the needs for the application, while respecting the good practices of API design.

This master thesis was realized as part of a 6-month internship at Applidium. During this internship, part of the time was dedicated to working on projects for Applidium’s clients, and the other part resulted in the project that is described in this report. The knowledge and experience acquired while participating in real-world projects for Applidium was put to use to build the presented solution and increase its relevance, as will be further explained later in this document.
1.2 Problem

There are constantly several projects concurrently in development at Applidium, and the company’s engineers are always investigating solutions to improve the different aspects of the software development cycle and enhance the management of projects.

For instance, back-ends for the mobile applications at Applidium are not always developed by the company’s internal team dedicated to back-end design. There are projects where the web services are provided by another company, and are used by the front-end developers at Applidium. While the back-end developers at Applidium have in mind to follow the REST principles and good practices for web API design as much as possible, and to provide accurate and exhaustive documentation, that is not always the case for ”external” back-ends. This is a problem that is commonly encountered by the front-end developers at Applidium, and which makes developing laborious.

Another issue is related to the current format used to describe the data embedded in HTTP messages in the documentations for back-ends at Applidium. This information is used by front-end developers to know what to include in requests and what to expect in responses. As of now, the description is done by giving examples, which may not be representative of all possible instances. This problem is commonly encountered in external APIs as well.

Finally, developers at Applidium use a number of tools that allow to test and document the different parts of a project. This raises two more issues. Firstly, testing, and writing documentation files is mandatory and essential, especially when working in a team. However, this currently takes a lot of time and is prone to lead to mistakes because all documentation and configuration files (for tests) are manually written. Secondly, the content for all these different files is largely based on the same information: the description of the API(s) for the project. This means that ultimately, as time goes by, different versions of the same information, that may not be synchronized and without an authoritative source, may coexist.

In this report, we investigate the following questions: How can the validation and development of projects at Applidium be optimized through the creation of an internal tool? What features should the tool incorporate?

1.3 Purpose

The purpose of this master thesis project is to help the developers (front-end and back-end) and project managers at Applidium by making the development and validation of existing and future projects at the company easier.
1.4 Goals

As evoked in section 1.2, the expected outcome of this master thesis project is meant to be a tool that finds its use both when the back-end for a mobile application is internally developed by Applidium, and when the back-end is being, or was developed externally.

In the context of internally developed back-ends

- The system should be, above all, a tool made for API description. The aim is for this system to constitute the authoritative source for information when it comes to an API at Applidium. This way, information is centralized, verified as correct only once, instead of being spread in multiple files (possibly in different versions),

- An important feature is to include the possibility to specify the structure of resource representations that are included in HTTP requests and responses. The format chosen by Applidium to this end is JSON Schema. Furthermore, as shown in section 2.4.2, many tools exist online that each provide different features related to JSON Schema. There is a will to gather all these functionalities in one place for more convenience,

- The solution is designed to support the automatic generation of exports (documentation files, and configuration files for tests) based on information entered in the tool as a means to save time, synchronize content and avoid mistakes in the obtained files. Any kind of service creating value out of API descriptions can be imagined to be plugged to the system, which illustrates its modularity.

In the context of external back-ends

There are two cases to distinguish:

- If the back-end was already developed, the intention behind the system is to help to document the API in case the provided documentation is incomplete or inaccurate. Besides providing a way to describe (and document) APIs, there is also a will for the solution to be able to act as a proxy, intercepting HTTP requests and responses between machines at Applidium and the external back-end. Ultimately, the ambition is to create an algorithm to automatically describe APIs based on the intercepted HTTP requests and responses and thus to dynamically provide descriptions that are actually based on the API’s implementation,
If the back-end is going to be or is being developed, the objective is for the system to help specify the API (by describing it) and to act as a "contract" between the (external) back-end developers and Applidium (e.g. during workshops between Applidium's developers and project managers, and the service provider), so that it is feasible to easily compare what was specified and what was actually implemented. The solution is therefore envisioned as a tool featuring a user-friendly interface, that allows to describe APIs quickly and conveniently, and in which information is saved as data structures (and not as text in a file).

1.5 Method

The first phase of the project consisted in gathering and understanding the issues encountered by developers and project managers at Applidium, and ushered in section 1.2. Actually working for clients with the team was really helpful for that matter, as highlighted in section 1.1.

This master thesis proposes a solution to the engineering problem described in section 1.2. The goals for the system detailed in section 1.4 were shaped empirically, based on observation gathered during that first phase.

Finally, the solution is evaluated regarding how users feel it helps them in their work, which constitutes the metric for this master thesis project. The evaluation method is thus qualitative, and data is collected using a questionnaire (online form).

1.6 Delimitations

The outcome of this master thesis should be a web application developed using Ruby On Rails and JavaScript, and aims at being used by people with different levels of technical knowledge.

Furthermore, the system will be used in the context of already established workflows at Applidium, which tools like Postman, WireMock, or Markdown are part of. For instance, the documentations for APIs at Applidium are written using Markdown [1], because it allows to generate HTML pages from plain text (using a certain syntax) that can then be reached online through a browser. This constitutes a constraint.

Finally, this master thesis project is the starting point of a much larger and ambitious project at Applidium with the same aim. This means that certain design choices that were made for the solution may not seem to fully make sense within the scope of the project as it is presented in this report, but will in the long run.
2 Background

This section provides technical details so that the technologies evoked in section 1 and later in this report can be better apprehended. Its content revolves around the notion of API, which is central in this master thesis. APIs are first presented generally, so that the meaning of API in the context of Applidium (and therefore of this master thesis project) can eventually be explained.

Finally, the section presents existing work related to the questions stated in section 1.2.

2.1 JSON and JSON Schema

This section aims at describing JSON and JSON Schema.

2.1.1 What is JSON ?

JSON (JavaScript Object Notation) is a format defined by RFC 7159 [2] that allows to exchange data which is very popular in web services because it is lightweight and can easily be read by humans. Here is an example of a JSON instance, that describes an employees array which contains 3 employees (taken from [3]):

```json
{
    "employees":[
        {
            "firstName":"John", "lastName":"Doe"
        },
        {
            "firstName":"Anna", "lastName":"Smith"
        },
        {
            "firstName":"Peter", "lastName":"Jones"
        }
    ]
}
```

2.1.2 Using JSON Schema to describe the structure of JSON instances

JSON Schema is a vocabulary that describes the structure of JSON instances (or examples) [4]. It is used to verify that JSON instances respect the structure described in the JSON Schema that corresponds to them. Here is an example of a JSON Schema:

```json
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "type": "object",
    "properties":{
        "employees":{
            "type": "array",
            "items":{
                "type": "object",
                "properties":{
                    "firstName": {"type": "string"},
                    "lastName": {"type": "string"}
                }
            }
        }
    }
}
```
This JSON schema validates the JSON instance displayed in section 2.1.1, because the instance is indeed a JSON object with a key `employees`, whose value is an array of objects that have a first name and a last name (both strings). However, the schema potentially validates an infinity of other JSON instances.

### 2.2 The HTTP Protocol

This section will focus on HTTP 1.1, as defined by RFC 2616 [5], since it is, at the time of writing, the most commonly used version on the internet.

#### 2.2.1 General considerations

HTTP is an application protocol (as defined in the OSI model) that is widely used to communicate on the world wide web, and which is therefore central to its operation.

First, HTTP is based on the client-server architecture. This means that participants are divided between servers, who provide a service, and clients, who request the services provided by servers. Second, HTTP is a request-response protocol. This simply means that the only way for one computer to initiate an exchange, is
to send a request to another computer, which will send a response, completing the exchange.

Therefore, in HTTP, clients (browsers, for instance), typically send HTTP requests to servers. Servers, on the other side, do nothing but wait to receive requests from clients. When a server does get a request, it processes it (according to what the request asked for), and sends back a HTTP response to the client containing the obtained result (which could be an error). Once the client receives the HTTP response corresponding to the HTTP request it sent, the exchange is complete.

Finally, HTTP is a stateless protocol: neither the client nor the server keep information about each other in-between exchanges [6].

2.2.2 HTTP requests and responses

This section aims at describing in more details the structure of HTTP messages (requests and responses). In what follows, SP stands for a space character, and CRLF (Carriage Return Line Feed) for an end-of-line marker. Both requests and responses include a start-line, followed by zero or more header fields, an empty line to mark the end of the header fields, and an optional message-body [5].

Header fields
The HTTP header fields (or headers) are meant to contain metadata about the HTTP message. Each header field (message-header in listings 1 and 2) consists of a field-name, followed by a colon and the corresponding field-value.

Message-body
The message-body corresponds to the data embedded in the HTTP message (entity-body), if there is any.

Media types
HTTP uses Internet Media Types (formerly MIME Types) to describe the type of the data included in the message-body. Relating to this issue, two header fields should be highlighted. First, the Content-Type header field displays the media type of the entity-body included in a HTTP message. Besides, the Accept header field can be used in a HTTP request to indicate desired media types for the entity-body of the response.

For instance,

- Content-Type: application/json

is a header field which implies that the entity-body of the HTTP message contains a JSON instance,
• **Accept: application/json**

  is a header field which indicates (in a HTTP request) that the client desires to obtain the entity-body of the response as a JSON instance, if possible.

**Start-line**

Request-Line
*(message-header CRLF)*
CRLF
Optional message-body

Request-Line = Method SP Request-URI SP HTTP-Version CRLF

Listing 1: Structure of a HTTP request

Listing 1 depicts the structure of a HTTP Request. The Request-URI in the Request-Line is a Uniform Resource Identifier (URI). It is a string that identifies the resource to be accessed through the request. The Method token specifies the HTTP Method (GET, POST, PUT, DELETE...) that will be applied on the aforementioned resource. The version of HTTP being used by the client is also included.

An example Request-Line would be:

GET https://www.google.fr/ HTTP/1.1

Status-Line
*(message-header CRLF)*
CRLF
Optional message-body

Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF

Listing 2: Structure of a HTTP response

Listing 2 depicts the structure of a HTTP Response. The Status-Code (meant to be used by machines) in the Status-Line corresponds to a 3-digit integer which describes the outcome of the corresponding request. The Reason-Phrase is a short text that accompanies the Status-Code to make it more legible for humans. A limited number of status codes along with their meaning are defined by RFC 2616.

An example Status-Line would be:

HTTP/1.1 200 OK

10
2.2.3 The HTTP Methods

This section aims at understanding the semantics of the HTTP protocol more deeply, by examining its most commonly encountered methods. Indeed, HTTP methods are standardized [5]. The point of this effort being that using a method should convey the meaning which is associated to it.

Furthermore, the standards also define the status codes that should be returned in the HTTP responses corresponding to HTTP requests for the defined methods, in the different possible cases.

What is a safe method?
Safe methods can be seen as requests that do not aim at changing any state on the server. They’re safe in the way that, from the point of view of the client, performing the method zero or ten times should not make a difference [7]: it is literally ”safe” to do it.

However, these methods may usher in side effects on the server side (such as logging, for instance, which constitutes a change in server state). What is important is that these are not requested by the client, who cannot be held accountable for them (because these side effects are consequences that are not part of the user’s initial intent behind the request).

What is an idempotent method?
A method is said to be idempotent if performing it \( n \) times, where \( n > 0 \), has the same effects as performing it once [7].

The GET method
The GET method should simply retrieve the information identified by the Request-URI in the Request-Line. GET is a safe and idempotent method.

The POST method
The POST method is more intricate than the other methods presented in this section, and only part of its semantics will be covered here, for it is what is needed within the scope of this report. One of the uses of POST is to append a new resource, based on the entity-body of the request, underneath the existing resource (the parent resource [7]) identified by the Request-URI. The POST method is not safe, and not idempotent either.

The PUT method
The behaviour of the PUT method depends on whether a resource is already identified by the Request-URI in the Request-Line.

- If an existing resource is already identified by the Request-URI on the server, then the entity-body of the request should be considered as the new state
for the identified resource (it can be seen as a modified version of the whole resource),

- If not, the server may create a new resource identified by the provided URI (if possible), based on the entity-body of the request.

In all cases, it appears that the entity-body included in the HTTP request fully describes the state in which the resource should be at the provided URI after the request was performed.

Finally, PUT is an idempotent operation [5].

The DELETE method The DELETE method instructs the server to delete the resource identified by the Request-URI. It is idempotent.

The PATCH method The PATCH method did not appear in RFC 2616. Instead, it was introduced later in RFC 5789 [8] to fill a gap left by the PUT method.

The PUT method lets the client modify a resource. However, it only does so by allowing to replace it completely (the embedded entity-body is used to fully overwrite the state of the resource): there is no possibility for partial changes.

This is when PATCH comes in:

- If an existing resource is already identified by the Request-URI on the server, then the entity-body of the request should be considered as a set of modifications to be applied to the identified resource for it to reach its new state,

- If not, the server may create a new resource identified by the provided URI (if possible), based on the entity-body of the request.

The PATCH method is neither safe nor idempotent.

2.2.4 The HTTP Status Codes

This section aims at introducing a few common status codes in the context of this report. Just like HTTP methods, status codes are defined by RFC 2616 [5].

- 200 OK: the request was successful,

- 201 Created: the request was successful and ushered in the creation of a resource,

- 400 Bad Request: the server could not understand the request because there was a problem in its syntax,
• 404 Not Found: there was no resource identified by the provided Request-URI,

• 422 Unprocessable Entity: the server could understand the syntax of the request but could not process it (defined in RFC 4918 [9]),

• 500 Internal Server Error: there was a problem on the server which prevented it from bringing the request to completion.

2.3 What is an API?

This section first introduces APIs generally to then tackle APIs in the context of REST.

2.3.1 General considerations

In an API, the focus is set on the services that the developed software can provide to others rather than on the implementation, which is usually hidden. Therefore, an API usually provides a documentation meant for external users to be able to use the services easily. The documentation acts as a contract between the designers and users of the API. It describes the different services provided by the API and how to use them, so that they can be integrated to other software. Many companies, such as Uber [10], provide APIs so that developers can integrate their services in their own applications.

Let’s take an example to see why APIs are so useful. Let’s imagine an application in which, for instance, there is a need to obtain the search results from a given search engine from queries for further processing. There are two solutions to solve this problem:

• The first solution is to create software that can parse the HTML of the page containing the search results to extract the desired data from it for all queries,

• The second solution is to use the API provided by the search engine (if there is one).

The problem with the first solution is that it relies on the fact that the structure of the page that contains the results from the search engine will always remain the same. However, it could happen that the search engine changes the way it generates its pages, which would most likely break the developed parsing script.

This is a problem that is avoided by using APIs: the authors can decide to change how the service is implemented without changing the interface that allows others to use it. Furthermore, the data returned by APIs is often well structured (the structure being described in the documentation) and thus quickly usable.
2.3.2 About REST and APIs

REST (REpresentational State Transfer) is the architectural style of the World Wide Web, as defined by Roy Fielding in his doctoral dissertation [11]. REST can be seen as a series of constraints that define the Web’s architecture, and that provide the architectural properties that made the web successful. Since the latter embodies one of the most remarkable successes in the history of distributed computing, people started to think that it would be wise to use the principles behind REST to build other distributed systems, such as APIs [12]. APIs that follow all the constraints described by REST are called RESTful APIs. This section does not aim at describing in an exhaustive way the constraints that REST imposes, for it would be too long, and unnecessary. The purpose of it is more to introduce concepts of REST applied to APIs that will be referred to in the context of this project.

RESTful web services usually leverage HTTP (as described in section 2.2) as an application protocol, to the point that it is often implied. Indeed, HTTP displays properties which make it easy for the web service to comply with REST architectural constraints (being stateless, using a client-server architectural style, for example...). Therefore, even though REST and HTTP go hand in hand most of the time, it should be pointed out that REST exists without HTTP, for the former is an architectural style, and the latter an application protocol. Nevertheless, in what follows (meaning the whole report), RESTful web services (and web services in general) are considered to operate over HTTP, as it is most commonly encountered [13].

Resources
The notion of resource is at the heart of RESTful APIs and more generally, of the REST architecture style. A resource (see figure 1 for an example) refers to a concept that finds its utility in the context of the application that is being built (e.g. a user, a customer, a song...) [14]. A resource can be described through its type, the attributes (to use vocabulary from object-oriented programming) that are associated to it (e.g. a song has an artist, a genre, etc...) and its relationships to other resources. Resources can be gathered in collections when they point to the exact same concept several times (e.g. users). Collections are referred to as resources as well. Otherwise, they are just called singleton resources [15]. A RESTful API often features several resources that correspond to the concepts being used.

Identifying resources
Resources should be identified through resource identifiers (identification of resources constraint, see [11]), typically a URI in the Request-Line of a HTTP
Figure 1: Example of a resource taken from the documentation for a past project at Applidium

request (see section 2.2) [16]. This is the way clients of a RESTful API know how to access a given resource.

A design principle for RESTful APIs (though not tackled directly in REST [7]) is that they should be intuitive: users should be able to understand as much as possible without having to read any kind of documentation. The names for resources and URIs, as well as the concepts they should describe and their structure are chosen by the developers behind the API. These choices should be meaningful in the way that the API should be designed while having in mind the clients that will consume it, instead of simply defining resources and URIs so that they reflect how the data is stored in the database (which could make the API really unclear to end-users). Good practices include, for example [16]:

- Using query string parameters in HTTP requests only for filtering,
- Using pluralized words in URIs to identify collections (e.g. /users instead of /user_list),
- Using the structure of URIs to reflect relationships between resources (e.g. /users/:id/orders conveys the idea that the resource orders only makes sense within a user, i.e. orders is a nested resource within users),
- Limiting the length of URIs as much as possible for the sake of clarity.

**Resource manipulation and representations**

Resources can be manipulated by clients through the exchange of HTTP messages with the server. As suggested above, a client can act on a resource by sending HTTP requests to the URI that identifies the resource. In return, the API will send a HTTP response containing the result of the request. The identification
of resources constraint also imposes that a resource should be disconnected from its representations on a conceptual level. Indeed, information about the state of the resource (or the state the resource is wished to be in) is embedded in the message-body of HTTP requests and responses that are exchanged between the server and the client. This information does not correspond to the resource itself: it is merely a representation of the resource. For instance, a server could send a HTTP response containing a representation of a resource (say, a user) using a JSON instance (see section 2.1.1) in the message-body. However, the same information could have been represented using XML, or even HTML, for example.

Therefore, so as to manipulate resources, clients exchange representations with the API: the manipulation of resources through representations constraint states that a representation of a resource, including metadata describing the representation, constitutes enough information to ”perform actions” on that resource [11]. Furthermore, a representation does not have to describe a resource extensively: a representation could consist of any information about the resource, packaged in a machine-readable format [14].

The aforementioned metadata should not be overlooked. Indeed, a request to a RESTful API should provide all the information necessary for the server to process it, as expressed by the self-descriptive messages constraint [11]. Consequently, the client needs to indicate to the server the chosen format for the representation of a resource transmitted in a HTTP request through the Content-Type HTTP header, as introduced in section 2.2. Likewise, the client can indicate a preference for the format of the representation of a resource in the HTTP response through the Accept HTTP header in HTTP requests.

Performing actions on resources
The missing piece of the puzzle consists in understanding how clients communicate to the API the action they wish to perform on a given resource. A mistake would be to try and include this information in the resource identifier, and to create a URI such as, for instance, /get_user, which could be tempting. This would not be RESTful for it would go against the identification of resources constraint: a resource identifier (URIs in this case), as suggested by its name, is meant to identify a resource, and nothing else.

Instead, RESTful APIs (over HTTP) use HTTP Methods to solve this issue. Therefore, clients use resource identifiers to identify resources and HTTP Methods to indicate which action to perform on the resource identified by the URI. It becomes clear that the representations of resources have different meanings (as suggested above), depending on the context set by the HTTP Method.

The constraints mentioned above (identification of resources, manipulation of
resources through representations, self-descriptive messages) are part of REST’s interface constraints [11]. An interface defines how the involved components (client and server, here) interact. In REST, the exchange of information through interfaces should be done in a uniform way. This means that clients should be able to interact with all resources using the same set of well-defined operations with the same semantics: a uniform interface. This set should be expressive enough for the service to provide the required functionalities. A uniform interface is interesting because it makes clients able to interact with different resources in the same way. As will be shown below, for instance, the standardized semantics for a subset of HTTP methods can be used to define a uniform interface.

The CRUD (Create, Read, Update, Delete) pattern to manipulate resources is well known among web services, for it features a simple, small set of methods that can be easily implemented, and which is enough to satisfy the needs for a service in a lot of cases. What is even better [12] is that these methods can easily be mapped to the HTTP methods defined in [5]:

- **POST** is used to create new resources (and therefore the corresponding URIs to identify them) (Create), based on the representation included in the entity-body of the HTTP request,

- **GET** is used to retrieve resource representations (Read),

- **PUT** is used to modify the state of resources that were previously created, and therefore already identified by a URI on the server (Update),

- **DELETE** is used to request the removal of a resource (Delete).

The way these four HTTP methods are used respects the protocol semantics specified in section 2.2.3, and this specifies a very common uniform interface [12]. The thing about REST is that it does not specify which uniform interface should be used [7]. However, it should be highlighted that the uniform interface provided by HTTP detailed above is most commonly encountered in APIs than others.

Making use of another uniform interface in an API would not violate any of precepts of REST. Nevertheless, that would mean that the API’s clients, who are probably familiar with the uniform interface defined by HTTP shown above, would have to learn how to interact with the service, when it could have been avoided: most people, for instance, would naturally use GET to obtain a representation of a resource, no matter the resource [7]. This is interesting because, combined with well designed URIs, it allows clients to be able to (almost) fully understand what a request is supposed to do just by looking at it (i.e. looking at the HTTP Method and Request-URI, in this context). **What matters is the community**
gathered around the uniform interface. Instead, if API designers decide to make up their own operations, they isolate themselves in a "community of one" [14].

The very same idea can also be applied to the status codes included in HTTP responses. Status codes are part of the protocol semantics of HTTP (see section 2.2.4), and are a machine-readable way to explain to the client what happened when the server tried to process the request. They give meaning to responses in a standardized manner, just like HTTP methods give meaning to HTTP requests.

Hypermedia as the engine of application state
The previous paragraphs have made it clear that the resource state is stored on the server. As explained above in this section, REST is the architecture style of the Web. A website can be seen as a state machine. Visualizing a certain web page corresponds to a certain state of the application, and clients go from state to state by following links (or hypermedia) [14]. However, since HTTP is stateless, the application state (as opposed to the resource state) is stored on the client side: the server has no idea which state any of the clients are in. This is what REST’s fourth (and last) interface constraint, known as "hypermedia as the engine of application state", is about. A website does not need to provide any kind of documentation on how to be used. The client discovers new states and possible transitions (i.e. what it can do in a given state) as it explores the service, and none of them are known in advance. Every new state simply offers a new set of possible transitions. This makes the web convenient, and really easy to use. This made the web successful.

Many APIs do not respect the "hypermedia as the engine of application state" constraint and, consequently, are not technically RESTful. They provide a human-readable documentation (as mentioned in section 2.3.1) because they would not be usable without it, unlike a website. There is, however, a simple explanation for this: websites are used by humans, who are smart enough to know on which links to click to get things done. Things are different with APIs. Indeed, their existence is motivated by the fact that they can communicate with automated clients. Therefore, reaching a wide adoption of the "hypermedia as the engine of application state" constraint in APIs would require a great standardization effort, so that, given a state of the application, automated clients could understand the meaning (the application semantics) behind the proposed transitions and make a decision, just like humans would on the web [14].

The "hypermedia as the engine of application state" constraint makes it possible for change to happen smoothly. Users still know how to use a website after it was redesigned. They just adapt to the changes. That is not true for APIs that do not comply with the constraint, because the clients have to be redesigned as well to reflect the change that occurred in the API. This means that change is
nearly impossible for popular APIs on the scale of the Internet that do not respect the constraint, because that would require all clients to change as well to keep working. Still, it is often a requirement that updated and non-updated clients should coexist and that none of them should break. This is a huge problem, considering that since the needs of users evolve with time, APIs do need to change [14].

The ”hypermedia as the engine of application state” constraint currently constitutes the biggest challenge in API design. Therefore, in the following sections of this report, the focus will be set on APIs that do not follow the ”hypermedia as the engine of application state”, since it is what is most commonly found on the web to this date [14]. Though technically not RESTful, these APIs can still benefit from the philosophy of REST and it is increasingly the case. Furthermore, they are fairly accessible if well-documented. This is the bet that is taken at Applidium, as APIs (or back-ends) are developed following the principles described in the previous paragraphs of this section as closely as possible.

2.4 Existing work

This section aims at introducing existing solutions that tackle the issues mentioned in section 1.2.

2.4.1 API description

First, in this context, the concept of route is central. In an API, a route maps an HTTP method and an endpoint (understand URI) to an ”action” (i.e. the procedure that will be performed by the server, following the request). The endpoint corresponds to the resource on which an operation is going to be performed, and the HTTP method can be seen as the action that will be performed on the aforementioned resource. To summarize, the different services an API offers to its users are embodied by the different routes it displays.

Restlet Studio [17] is a tool that allows to describe and document APIs. It notably lets users add resources to a project and describe routes within those, through a visual web interface that makes it quick and easy to edit projects without any learning phase. HTTP requests and responses for given routes can be detailed extensively. Furthermore, Restlet Studio offers to generate documentation pages displaying the entered information in an organized and legible way. The documentation, presented in a format designed by Restlet, is available as a web page hosted online by the service, so that it can be shared simply.
Swagger [18] is similar to Restlet Studio in the way that it aims at letting users describe APIs. The major difference resides in the format in which APIs are to be described. Indeed, while Restlet Studio features a convenient user interface through which information is saved in the form of data structures, Swagger’s purpose is to make API descriptions machine-readable. To this end, it defines a specification, known as the *OpenAPI Specification* [19], which defines a standard meant to describe APIs in the form of a text file (JSON or YAML). It is possible to write Swagger specification files manually and there exists an editor dedicated to it [20]. However, the process remains undeniably intricate and cannot be done without training. Therefore, there is an incentive to generate the specification files automatically from the API source code, which, actually, constitutes Swagger’s strength and explains why the OpenAPI Specification was created. Indeed, it makes it possible for an API to describe itself. Besides, Swagger allows to specify the entity-body (whose media type is application/json, in this context) of HTTP requests and responses for routes via a subset of JSON Schema Draft 4 [21].

In addition, Swagger provides a collection of tools so as to generate exports from a specification file once it is created for an API. Swagger UI, notably, interprets the Swagger specifications so as to render documentation pages which display the API’s routes sorted by resource and let readers directly make calls to the API and observe the responses, all from the browser. Finally, Swagger specification files can also be imported to Postman in order to effortlessly create test collections for the API and test each route.

**JSON Hyper-Schema** [4], much like the OpenAPI Specification, offers a way to describe APIs in a machine-readable format. JSON Hyper-Schema is the brainchild of the people behind JSON Schema. It builds off of the JSON Schema specification and can be seen as an extension of it with the aim of describing JSON-based APIs.

One of JSON Hyper-Schema’s flaws, just like Swagger’s OpenAPI Specification, resides in one of its strengths. Deciding to describe APIs in a machine-readable format is an interesting design choice because it makes it possible for developers (not only those who authored the specification) to create tools to easily export the provided information in other formats in an automatized way. JSON Hyper-Schema and the OpenAPI Specification are very useful and powerful tools in themselves, since they allow to capture complicated concepts. Nevertheless, Swagger would probably not be so popular without the ecosystem that surrounds the OpenAPI Specification (Swagger UI, for instance).

The obvious downside is that users often have to deal with one huge file that describes a whole API, which can be hard to understand and edit, even for people who are familiar with the concepts described in sections 2.2 and 2.3.2.
Prmd [22] is a Ruby gem (i.e. a library for the Ruby language) developed by Heroku that allows to bootstrap API descriptions that use JSON Hyper-Schema, and to automatically generate documentation files in Markdown based on the description.

Each tool described in this section only constitutes a partial answer to the questions enunciated in section 1.2. The inception of the outcome of this master thesis is thus explained by a strong will to fit the needs of Applidium as much as possible.

Indeed, Restlet Studio does not quite fit the goals of section 1.4 because the documentation for APIs cannot be retrieved in the desired format (Markdown). Furthermore, this tool (as of today) does not offer to generate desired export files such as configuration files for WireMock natively, and adding custom exports is not possible as well.

Though Swagger and Prmd could fulfill certain goals detailed in this same section, the formats they depend on to describe APIs (OpenAPI Specification and JSON Hyper-Schema, respectively) are unsuitable in the context of this master thesis project, because of their lack of accessibility.

2.4.2 JSON Schema-related tools

This section aims at introducing a few existing tools related to JSON Schema that are commonly used by developers at Applidium.

JSON Schema Lint [23] is an online tool that allows to validate JSON instances against JSON Schemas through a graphical user interface.

JSON Schema Faker [24] (available both as a JavaScript library and as an online tool) allows, given a JSON Schema, to generate JSON instances that are validated by the schema and are filled with random data.

JSONSchema.net [25] is an online tool that allows, given a JSON instance, to infer a JSON Schema that validates the JSON instance.

Ruby JSON Schema Validator [26] is a Ruby gem that allows to validate JSON Schemas against the JSON Schema specification (up to JSON Schema Draft 4 [27]), and to validate JSON instances against JSON Schemas.
3 Implemented solution

This section aims at describing the implemented solution for this master thesis project and to present its features along with technical details. The goals for the system were set in section 1.4.

3.1 Overview

The implemented solution, called Pericles, is a web application (or website) available on the Internet, and implemented using Ruby on Rails and JavaScript. The tool can therefore be quickly deployed online and be easily accessible to any developer at Applidium through a browser, just like any website.

Ruby On Rails is a web application framework written in Ruby that allows users to build web applications quickly. Rails incorporates many conventions and heavily encourages users to do things in a certain way, considered as the best one by its makers. Therefore, Rails features incentives to follow these conventions, making the development really easier if so and, on the opposite, making it unpleasant not to follow the "Rails way" ("Convention over Configuration") [28].

3.2 Software architecture

This section presents the software architecture of the project.

3.2.1 Rails and the Model-View-Controller pattern

The development in Ruby On Rails is done according to the Model-View-Controller pattern which, therefore, constitutes the architecture for the project. How the pattern is incorporated in Rails is described below:

- **Controller** - In Rails, each controller receives specific requests for the application. An application usually features different controllers, and routing makes sure that requests are forwarded to the right one. Thus, HTTP requests are forwarded to a controller in the application through a given route (cf section 2.4.1), that is associated to a given action in this controller. Usually, in an application, several different routes lead to the same controller. An action is simply a (Ruby) method in the controller whose aim is to serve the received request. To do so, it typically performs operations on the database indirectly through models and then provides the information obtained from the database to a view so it can be displayed to users [28].

- **Model** - In most cases, an application needs to save data in a persistent way. For instance, a blog contains articles and comments that need to be
stored in a database so that the data can be accessed later. It is a common shortcut to use the word “model” to refer to the database, but it is not exactly what a model is in Rails, although it is related to it. A model represents a concept that is part of the business logic of the application. For instance, in the context of the blog, the Rails application would most likely feature a model for articles and another for comments; because those are two concepts that will need to be manipulated. Technically, a model is a Ruby class, as introduced in Object Oriented Programming. It notably contains information about the validations performed on objects (e.g. an article cannot be created without a title) and can also describe relationships to other models (associations) (e.g. an article has several comments).

Furthermore, in Rails, each model (through naming conventions) is automatically mapped to a table in the database, which is an important feature of Object Relational Mapping (ORM). For instance, creating an Article model in Rails usually goes along with (if conventions are followed) the creation of a table called articles in the database to which is it linked. Indeed, a row of the articles table (an article) will be represented as an instance of the Article model in Rails (an object). The columns of the table (for the given row, or article) are directly mapped to the attributes of the mentioned object, and actions on the object (in an Object Oriented fashion) can be mirrored to the database, without writing SQL statements. This is how Rails (through ORM) allows to easily create and use persistent data from the database using models [29].

- View - A view simply aims at displaying information gathered in and by controllers in a human readable format [28].

3.2.2 A few more words about models...

The concepts of validation and association mentioned above need further explaining because they are central in Rails:

Validations - In Ruby On Rails, the same information usually exists in two ways: in the form of what is called an Active Record object (that is manipulated in controllers, for instance), or as a row in a database. Whether an object is already persisted in the database does not matter: before creating a new record in the database or updating an existing one, Rails offers to run validations on the corresponding object (the validations that correspond to the object’s class, or model). Here is an example of a simple validation taken from the Project model in Pericles:

validates :title, presence: true
This validation simply means that a project cannot be saved without a title. Validations are run before SQL statements are issued to the database. Should any validation fail, SQL statements won’t be sent and the object will not be stored in the database (i.e. the creation or update will not be performed). Therefore, validations aim at ensuring that only valid objects are stored in the database.

Associations - Associations are another key concept in Rails: it allows to tell the framework about the relationships between models. It gives access to a lot of powerful features that let users manipulate these relationships more conveniently, using a syntax that is closer to what is seen in Object Oriented Programming than SQL. There is, for instance, in addition to the Project model, a Resource model in Pericles (and the corresponding tables in the database, projects and resources, respectively). The idea we would like to implement is that a project can have many resources (where "resource" refers to the concept defined in section 2.3.2).

Each resource belongs to one project, and a project can have many resources. These two connections are translated, from the point of view of the database, in the fact that the resources table has a project_id column (which stores integers). All tables created with Rails have an id attribute by default, which constitutes the table’s primary key. Each row of each table (in Pericles, at least) is therefore uniquely identified (in the context of its table) by an integer. The project_id column in the resources table is a foreign key that references the id attribute of the projects table. This is a very common pattern in Rails to implement the kind of two-way relationship illustrated here through projects and resources, in the database.

• Now that the relationship is implemented in the database, it would be possible to find all resources for a given project with id 1 by executing the following line of code:

```ruby
@resources = Resource.where(project_id: 1)
```

which is syntactically really close to the SQL statement generated by Rails to query the database in response to the execution:

```sql
SELECT "resources".* FROM "resources" WHERE "resources"."project_id" = $1 [["project_id", 1]]
```

This works, but the Rails framework is a lot more powerful than that. At this point, the framework does not know about any relationship between projects and resources.

The following line of code:
has_many :resources, dependent: :destroy

is included in the Project model. The has_many association is used to inform Rails that an instance of the Project model has zero or more instances of the Resource model [30]. The first parameter after has_many (resources, here) is just the name chosen for the association, that will be used in the code. Three other options for has_many are actually implied by default. The first one, class_name, is used to specify the name of the model on the other side of the association. If the option is not provided, Rails simply infers the model name from the name of the association (the model name is Resource, in this example). Rails can thus (also) infer the name of the table corresponding to that model (resources) through naming conventions, as explained above. The second one, foreign_key, is used to provide the name of the foreign key (in the table corresponding to the model on the other side of the association, called resources here) referencing the primary key of the projects table. Since the option does not appear, Rails assumes that the name of the foreign key is the name of the Project model (i.e. the declaring model), followed by ”_id” (project_id). Finally, the primary_key option can be used to specify the name of the declaring model’s primary key (the id attribute, by default).

Therefore, when the following line of code is executed (where @project is an instance of the Project model that is persisted in the projects table with id 1),

```ruby
@resources = @project.resources
```

Rails issues the SQL statement shown in figure 2 to the database,

```sql
SELECT "resources".* FROM "resources" WHERE "resources"."project_id" = $1 (["project_id", 1])
```

Figure 2: Correspondance between the options for the has_many association and the SQL statement generated when using the association

and finds the resources that correspond to @project, just like what was done above with the detailed version. Instances of the Resource model are created by Rails in memory based on the rows found in the resources table, as explained above, and the created resource objects are stored as an array in the @resources variable, that can be manipulated later in the code. Rails
is able to do this because every step of the reasoning behind the idea that "an instance of the Project model has zero or more instances of the Resource model" was broken down through the has_many association, as illustrated in figure 2.

Moreover, the dependent: :destroy option tells Rails that when a project record is removed from the database, all corresponding resources should be automatically removed as well.

- At this point, Rails still ignores that each resource belongs to one project. However, the following line of code is included in the Resource model:

```
belongs_to :project
```

`belongs_to` is usually seen as the companion association for has_many. It is used to inform Rails that, indeed, each instance of the Resource model belongs to an instance of the Project model. The first parameter after belongs_to (project, here) is just the name chosen for the association, to be used in the code. The (implied) options for belongs_to are the same as the ones mentioned above for has_many, with very similar semantics. class_name is still used to specify the name of the model on the other side of the association, with the same consequences if the option is not present (the inferred model name is Project, here). The foreign_key option is again used to provide the name of the foreign key for the association which is, this time, an attribute of the table corresponding to the declaring model (i.e. the resources table). The name of this foreign key is inferred by Rails by adding "_id" to the name of the association (project_id), because the option is not listed. The primary_key option would refer to the name of the primary key of the table for the model on the other side of the association (i.e. the projects table), which is assumed to be id here, by default.

Therefore, when the following line of code is executed (where @resource is an instance of the Resource model that is persisted in the resources table with project_id 1),

```
@project = @resource.project
```

Rails issues the SQL statement shown in figure 3 to the database, and finds the project that corresponds to the resource.
Both sides of the relationship between projects and resources have now been implemented in Rails. The associations can thus be used on model instances, as illustrated above, in controllers, to ease interactions with the database as well as the coding process and make the code more accessible.

Besides, in order to be sure to avoid database issues, referential integrity constraints can be added directly as database constraints (not as validations) when foreign keys are used. This is done in Pericles, as, for instance, the project_id of a row in the resources table cannot reference an id that does not exist in the projects table.

3.2.3 About routing in Ruby On Rails

Routing aims at forwarding requests made to the Rails application to the right controller so that they can be served. In the context of Rails, requests are HTTP requests, sent to a given URI (or endpoint, which identifies a certain resource) using a given HTTP method. As introduced in section 2.4.1, a route maps an HTTP method and an endpoint to a certain action in a controller.

Though this is not the only way to define routes in the framework, Rails offers the possibility to create resource (as in section 2.3.2) routes [31] (i.e. define routes used to manipulate a given resource). For instance, Pericles includes the following line of code in the file (routes.rb) used to specify routing:

```
resources :projects
```

This line of code creates eight different routes, as illustrated in figure 4, that can be used to perform actions on resources associated to the notion of project. These routes are the ones that map to actions in the projects controller, as shown in the Controller#Action column.

Following the Rails conventions, the projects controller manipulates (notably) instances of the Project model mentioned earlier, and implements the seven classic actions associated to resource routes in Rails. These actions are (explained in the context of the projects resource):

- **index** - Display a list of projects,
Figure 4: List of all routes defined in Pericles

- **show** - Display a given project,
- **new** - Display a HTML form allowing to create a new project,
- **edit** - Display a HTML form allowing to update an existing project,
- **create** - Create a new project,
- **update** - Update an existing project,
- **destroy** - Delete an existing project.

Certain routes accept parameters. An HTTP Request to the URI /projects/4 with the method GET, for instance, would trigger the show action of the projects controller to try to return a representation of the project corresponding to the :id parameter of value 4.

It now becomes clear, looking at figure 4, that resource routes in Rails aim at defining routes for each of the CRUD operations mentioned in section 2.3.2, and, especially, follow the uniform interface provided by HTTP defined in the same section (thus respecting the protocol semantics of HTTP).

On its own, figure 4 defines all possible operations and behaviours possible on Pericles. Therefore, it highlights that, though being a website, Pericles can be seen as an API over HTTP, following REST principles closely (including the "hypermedia as the engine of application state" constraint), as introduced in section 2.3.2. Indeed, figure 4 shows that, under the hood, Pericles exhibits a certain number of
resources (projects, routes ..., see the Controller#Action column) that can be manipulated using routes defined (only, except for the ones needed to handle errors) through Rails’ resource routes. The trick is that the representations of resources returned by the server in HTTP responses are HTML documents that contain transitions allowing to move from the current to the next state of the application. Each of these transitions simply corresponds to one of the routes defined in figure 4.

3.2.4 Overview

Figure 5 summarizes the many concepts discussed in this section and illustrates the software architecture behind the project.

![Software Architecture Diagram]

Figure 5: The software architecture of Pericles

3.3 Implementation

This section describes how the solution was implemented.

3.3.1 API Description

As stated in section 1.4, Pericles is, above all, a tool made for API description. It is a web application which provides a user-friendly interface that lets users describe APIs by entering data through HTML forms. This data is persisted in a database,
as introduced in section 3.2, so that it can be accessed again later. Therefore, a
great part of the work aimed at being able to model an API, as they were de-
scribed in section 2.3.2, so that they can be “stored” in database tables. It could
be added that, in the context of mobile applications at Applidium, the front-end
and the back-end (i.e. the API) exchange representations of resources over HTTP.
In almost all cases, to put it simply, the representations come in the form of JSON
text.

![Figure 6: The models in Pericles](image)

Figure 6 is an entity-relationship diagram that presents all models in Pericles.
As explained in section 3.2.1, each model is associated to a table in the database,
whose name can be derived by basically pluralizing the corresponding model’s
one (e.g. the QueryParameter model is associated to the query_parameters table,
etc...). Therefore, figure 6 describes the models in Pericles as well as the database
underneath it, and the attributes in the right column of each entity in the diagram
are the attributes that indeed appear in the database table corresponding to each
represented model.

The database is the result of the constraints that come with the concepts that
need to be represented, of course. It was designed so as to avoid the duplication
of information as much as possible, and make storage efficient. Effort was also
put into making the design clear and precise, in order to favor its extensibility for,
once again, this master thesis constitutes the first step of a bigger project: the
names for tables (as well as the idea behind these), their attributes and the types
for these attributes were chosen to be meaningful regarding to the concepts that
are manipulated in the application. Finally, Rails, as a framework, offers certain
features and thus imposes certain choices when it comes to the database.

The arrows in the diagram are used to model one-to-many relationships, which
are implemented in Rails using the has_many and belongs_to associations in models.
For instance, as explained in section 3.2.2, a project can have many resources, and
each resource belongs to one project. The rest of this section will be dedicated
to making how APIs were modeled in Pericles clearer, and will tackle certain
difficulties that were encountered during the process.

Project A project simply corresponds to an API that is described through Per-
icles.

Resource The concept of resource is meant as it was defined in section 2.3.2. A
project typically introduces several resources that can be manipulated by users.
The intention behind the Resource model in Pericles is to gather all information
related to the concept behind a given resource. For instance, a project could have
a User resource. However, routes to manipulate users could have been defined
just like in section 3.2.3, with routes corresponding to the standard index, show,
create, update and destroy action in Rails (CRUD). To be perfectly clear, section
2.3.2 explained that different resources are identified by different resource identi-
fiers. URIs such as /users and /users/1 (or more generally, /users/:id) technically
identify different resources (the first is a collection, and the other is a singleton
resource). However, routes such as POST /users (to create new users) and GET
/users/:id (to obtain a representation of a given user) are meant to be gathered
as routes (i.e. the model in Pericles) in the User resource. The User resource, in
Pericles, should thus be understood as ”everything about users”. This to explain
that a single resource in Pericles may gather several routes with different URIs
that may actually identify different resources, technically speaking.

Attribute Each resource has several attributes. In Pericles, attributes have
a name, a type, and can have a description, as well as a descriptive example
associated to them.

Integer, string, boolean, and null are available as primitive types for all at-
tributes. Besides, when a resource is created in a project, it becomes an available
type (a resource type) for attributes in other resources in the same project. This is
how relationships between resources are modeled inside projects. Primitive types
and resource types are the basic types that can be chosen for attributes. However, attributes can also be arrays of these basic types.

The \textit{is\_array} boolean in the \textit{attributes} table is used to store whether an attribute corresponds to an array or not. The primitive type of an attribute (if it has one) is stored in the database as an integer through the \textit{primitive\_type} column (each of the four primitive types is simply mapped to a different integer, this being handled transparently through what is called a Rails enumeration). Besides, \textit{resource\_id}, a foreign key, is used to store the \textit{id} of the resource that constitutes the attribute’s resource type, if it has one. As a result, the resource to which the attribute belongs to is referenced using another foreign key, called \textit{parent\_resource\_id} for the sake of clarity (but deviating from Rails conventions). Referential integrity constraints enforced in the database make sure that \textit{resource\_id} and \textit{parent\_resource\_id} indeed reference existing resources. Finally, validations (see section 3.2.2) in the \textit{Attribute} model are applied so that attributes cannot exist without a basic type, but also cannot have both a primitive type and a resource type.

This paragraph highlights that modeling something that seems as simple as the type of an attribute is actually far from straightforward. This modeling was heavily influenced by the specification of JSON values in [2] ("A JSON value MUST be an object, array, number, or string, or one of the following three literal names: false null true"), as a way to ensure that the description of types in Pericles is complete.

\textbf{Route} Routes were discussed in section 3.2.3. The concept of route is at the heart of Pericles, as it is at the heart of APIs (in the context of Pericles), because APIs provide services in the form of routes. The different routes defined for a given resource (i.e. for a given URI) can be seen as the different operations that can be performed on that resource in the API.

As explained in section 2.3.2, resources are manipulated by clients through the exchange of HTTP messages. In order to describe APIs exhaustively, Pericles makes it possible to specify HTTP requests as they are expected by the server, and the different HTTP responses that can be returned (as defined in section 2.2); in the context of a given route.

Within a route, all HTTP requests follow the same pattern (only one "type" of HTTP request can be sent). The HTTP method and Request-URI (stored using the \textit{http\_method} and \textit{url} attributes in the \textit{routes} table, respectively) will, indeed, always be the same. The available HTTP methods in Pericles (implemented using a Rails enumeration) are POST, GET, PUT, PATCH and DELETE, because they constitute the uniform interface introduced in section 2.3.2.

Besides, Pericles allows to detail which HTTP headers need to be included (through field-names), if any, as well as possible fields for query string parameters.
in requests, so that the modeling captures all possible HTTP requests for a given route. The field-name of the headers and the names for the fields of the query string parameters themselves will not change depending on the HTTP request for a given route, only their associated value may vary. Finally, the resource representations included in the message-body of HTTP requests for a given route all respect the same structure. This structure is persisted through the request_body_schema attribute of the routes table, meant to save a JSON Schema.

Therefore, Pericles does not include a Request model, because it would have ushered in additional joins in the database that do not need to be. The fact that there is, conceptually, a one-to-one relationship between routes and requests explains why all information related to HTTP requests is included in the routes table. Creating a Request model would have, perhaps, made the database design clearer, but the choice of names for attributes (and associations) in the routes table (e.g. request_body_schema) that are related to HTTP requests makes it clear enough anyways.

Response   However, several "types" of HTTP responses can be associated to a route, for a request to the API may succeed, but it may also fail. This one-to-many relationship justifies the existence of a Response model. HTTP responses, in Pericles, are specified through a status code, HTTP headers to be included by the server (if any), and a description to explain the meaning of the response. Pericles responses can also include a JSON Schema (body_schema attribute in the responses table), meant to describe the structure of all resource representations included in the message-body of HTTP responses, that are returned in the context of a given route with the same status code.

Requests (through routes) and responses, in Pericles, should not be understood as actual HTTP requests and responses, but more like patterns that characterize the HTTP messages exchanged when the API serves clients.

About JSON Schemas   The Route and Response models include custom Rails validations (as introduced in section 3.2.2) that make sure that the JSON Schemas eventually stored in the database are valid according to JSON Schema Draft 4. The Ruby JSON Schema Validator gem introduced in section 2.4 was used to this end. This gem was chosen because it was one of the most popular to offer the desired functionalities and to support recent versions of the JSON Schema specification (at the time of development), but also because the associated GitHub repository showed recent activity, which suggested that the gem was still being maintained.

Header   The headers table simply includes a name attribute to store the field-name of a HTTP header, and a description attribute to explain why the header
needs to be included in the HTTP message.

**Query Parameter** In the following URI,

http://example.com/users?last_connection=tuesday&age=25

*last_connection=tuesday&age=25* is known as a *query string*. The beginning of the query string is indicated by a question mark in the URI. The query string consists in a series of field-value pairs, or parameters. Fields are separated from values using equal signs, and the parameters are distinguished using ampersand signs. Query string parameters are often used to implement filtering functionalities in APIs.

Much like the *headers* table is basically meant to store field-names of HTTP headers, the *query_parameters* table includes a *name* attribute used to store names corresponding to fields of query string parameters. The *description* attribute allows to explain what the query string parameter should be used for, and *is_optional* if it is mandatory or not. Finally, three types are available (integer, string and boolean) for the value corresponding to the described field. This choice was, again, motivated by the specification of JSON values, adapted to query string parameters. Each of these types is mapped to a different integer (through, once again, a Rails enumeration) so that the type information can be stored as an integer in the database (*primitive_type* attribute).

### 3.3.2 The validation API

Pericles incorporates its own "API" (one route, actually) that lets users validate any JSON instance against any JSON Schema. Such a service could not be found on the Internet during the development of this project. Indeed, most websites, such as JSON Schema Lint [23], only provide the possibility to validate JSON instances against schemas through graphical user interfaces.

This section can be seen as a documentation file. It aims at presenting a bird’s-eye view of the API, which would allow developers to integrate it into their own software.

The route in Pericles through which the service is available is the following:

**POST /validations**

It could have been tempting to make the validation service available through an URI like /validate.json. The JSON schema and the JSON instance to be validated against it could have been embedded in the message-body of the HTTP request,
and the API would have simply returned the result of the validation following a certain syntax, and nothing else. It would have worked, but it would have also gone against the principles of REST described in section 2.3.2. In REST, everything is thought in terms of resources. Therefore, there is a Validations resource in Pericles. A Validation has a json_schema and a json_instance which correspond, respectively, to the JSON schema and JSON instance sent by the client. It also has a status, which describes the result of the validation. Validations are stored in the validations table (cf figure 6). Finally, it can have json_errors (stored in the json_errors table, see figure 6). These json_errors correspond to the errors that may have happened during the validation.

The POST /validations route is thus meant to create a new validation resource (identified by a new URI), which basically gathers the parameters for, and the result of the validation performed by the client. This is different from saying that it is simply meant to "perform validations". Furthermore, here, the use of the POST method is consistent with the uniform interface defined by HTTP described in section 2.3.2. The paragraphs below show that HTTP status codes are used in a consistent way as well.

Any HTTP request to this route should include the Content-Type header. In what follows, it is assumed that the associated media-type is application/json. The JSON instances included in the message-body of HTTP requests should be validated by the JSON Schema provided in appendix A.1. The json_schema and json_instance keys in the representation of a validation as a JSON instance correspond to the attributes of the same name in the Validation resource described at the beginning of this section.

Below are described possible inputs that can be sent to the API, and how they are handled by Pericles. In each case, it is assumed that the errors described in the previous cases are fixed. For instance, the first case assumes that the JSON instance in the message-body of the HTTP request is not a valid JSON text. In all cases following this first one, it is implied that the JSON instance in the message-body of the HTTP request is a valid JSON text.

- First, if the JSON instance in the message-body of the HTTP request is not a valid JSON text according to the JSON specification [2], the answer is an HTTP response with an empty message-body, whose status code is 400. Rails provides this feature by default.

The behaviours described below were implemented during the project in the create method of the Validations controller, to which the POST /validations route maps, as shown in figure 4.

- If the JSON instance in the message-body of the HTTP request is not an
object containing a validation key, the answer is an HTTP response with an empty message-body, whose status code is 400.

In the cases described above, the syntax of the request is not correct, and it thus cannot be understood by the server. Indeed, client and server should exchange representations of validations. Anything that is not a valid JSON text containing an object with a validation key is not considered as a representation of a validation by the server. When receiving something that is not the representation of a validation in the message-body of a HTTP request, the server does not create a new validation resource, and returns 400 as the status code, consistently with what was explained in section 2.2.4.

For the next cases, it is assumed that the message-body of the HTTP request contains a valid JSON text, that is an object which includes a validation key, whose corresponding value is a non-empty object. In all of these cases, the server attempts to validate the sent JSON instance against the supplied JSON Schema, in two steps. The validations are performed using a method (fully_validate) provided by the Ruby JSON Schema Validator gem [26]. The method takes two arguments, a schema and an instance. Both of these arguments need to be strings. First, this method is used to validate the JSON schema against the specification (JSON Schema Draft 4). If the provided schema is a valid JSON schema, then the server performs the validation of the given JSON instance against the validated schema (using the same method). Otherwise, that is not the case, because the schema is not valid. No matter what happened, the server then creates a new validation resource and returns 201 as the status code in the HTTP response, to indicate that a resource was indeed created.

The JSON instance included in the message-body of the returned HTTP response is validated by the JSON Schema provided in appendix A.2, where each key (json_schema, json_instance, status and json_errors) in the representation of a validation as a JSON instance corresponds, once again, to the attribute of the same name in the Validation resource described at the beginning of this section.

- If the value corresponding to the json_schema key in the message-body of the request is not a valid JSON text (according to [2]) represented as a string (which includes the case where the json_schema key is not present), the status for the validation returned in the response is schema_parse_error.

- If the value corresponding to the json_schema key in the message-body of the request is not a valid JSON Schema according to JSON Schema Draft 4, the status for the validation returned in the response is schema_validation_error.
• If the value corresponding to the `json_instance` key in the message-body of the request is not a valid JSON text (according to [2]) represented as a string (which includes the case where the `json_instance` key is not present), the status for the validation returned in the response is `instance_parse_error`.

• If the value corresponding to the `json_instance` key in the message-body of the request is not validated by the supplied JSON schema (which is valid), the status for the validation returned in the response is `instance_validation_error`.

• If the value corresponding to the `json_instance` key in the message-body of the request is validated by the supplied JSON schema (which is valid), the status for the validation returned in the response is `success`.

3.3.3 The validation API, integrated

The validation API is fully integrated in Pericles (i.e. as part of the web application to describe APIs) through graphical user interfaces. This is part of a larger goal to further incorporate JSON Schemas while working on projects at Applidium, and to gather several scattered features all in one place.

• First, users can validate JSON instances against JSON Schemas that are already persisted in the database in the `routes` and `responses` tables as part of API descriptions (as shown in figure 7). The user can simply enter the JSON instance in a HTML form, and pick (using HTML radio buttons) against which available JSON Schema the instance should be validated. This feature is included as part of the HTML view that represents `routes` (i.e. instances of the `Route` model, as introduced in section 3.3.1). It is important to highlight that, in this case, the JSON schemas are already known to be valid according to JSON Schema Draft 4.

• Pericles also lets users validate JSON instances against any JSON schema, in a dedicated view (much like what is done in JSON Schema Lint [23]). In this case, both the JSON schema and the JSON instance are simply entered through HTML forms.

In both cases, the validation is performed by sending an HTTP request to the POST `/validations` route, as explained in section 3.3.2. jQuery, a famous JavaScript library that is less verbose and easier to use than pure JavaScript, was used to this end.

• First, jQuery allows to select and manipulate HTML elements. jQuery is thus used in both cases described above to obtain the user input (which consists
Figure 7: Excerpt from the representation of a route in Pericles as a HTML document. The page includes a form that lets users validate JSON instances against the JSON Schemas persisted in the database that are related to this route in a JSON schema and a JSON instance), entered through HTML forms or radio buttons, when the user clicks on the validate button to perform a validation.

- An AJAX call is then made with jQuery to the validation API. This simply means that an HTTP request is sent asynchronously by the user’s browser to the POST /validations route. The HTTP request includes the supplied user input in its message body, as a JSON instance representing a validation resource (cf section 3.3.2).

- The HTTP request is then processed by the API, as described in section 3.3.2. In return, the browser receives an HTTP response which contains a representation of the created validation resource. By using jQuery to manip-
ulate HTML elements, the result of the validation is displayed to the user, according to, notably, the \textit{status} attribute of the obtained representation of the validation as a JSON instance.

It would have been possible to allow users to validate JSON instances against JSON Schemas without any JavaScript at all. This is, actually, what was originally done in Pericles. When the user clicks on \textit{validate}, it would directly fire an HTTP request to the POST /validations route to create a validation resource. The issue is, however, that there would be no way to show the result of the validation without displaying a whole new HTML view to the user, which would be contained in the returned HTTP response.

It would also be possible to provide this service using only JavaScript, thanks to libraries such as Ajv [32], that is to say without firing any HTTP request when the user clicks on the \textit{validate} button. In this case, the whole validation process would happen in the browser on the user side, through JavaScript. The advantage is that the validation is, in this case, fully transparent to the user. There is no need to load a new HTML page (which can be confusing to the user, or at least, unexpected) and the result of the validation is simply displayed on the same page where the user clicked on \textit{validate}. This solution provides simpler and more flexible ways to obtain a better user experience than the JavaScript-free solution above. However, the drawback, compared to this same solution, is that no history of past validations is persisted in the database.

Choosing to implement the possibility of validating a JSON instance against a JSON Schema by creating an API allows to take the best out of these two worlds. Using jQuery and AJAX calls makes it possible to perform the validation on the server side (i.e. by sending an HTTP request to the API) without having to reload the whole web page. When the user clicks on \textit{validate} on a certain HTML view, a new validation resource is created and persisted in the database on the server side, and the result of the validation appears on the very same view (by using jQuery to manipulate HTML elements). As a result, Pericles keeps a history of past validations that can be consulted by users.

Finally, the choice of creating an API provides great modularity. Not only was it easy to integrate the API in different parts of a given project (Pericles), using mostly the same (factorized) code every time, but the API can also be re-used in future projects \textit{as it is}, without further effort.

\subsection{Automated generation of documentation}

Once an API is described in Pericles, the tool allows users to \textbf{automatically generate documentation files in Markdown} for the project (i.e. for the API).
A *project*, in Pericles, is a resource (as in section 2.3.2), identified by a resource identifier (a URI). The actual information about a given project (or API) that is described in Pericles is persisted thanks to the database that can be derived from figure 6. As explained in section 3.2.2, a *project* can have many *resources*. Each *resource* (i.e. a row in the *resources* table) of a *project* is a resource (as in section 2.3.2) as well.

Users, in Pericles, can visualize a *resource* from a *project* (to see the different *attributes*, etc...). More precisely, what the users actually see in this situation (cf section 3.2.3) is a representation of that *resource* as a HTML document (a view, in the model-view-controller pattern). The process behind the generation of such a view was already explained in figure 5.

The possibility to generate documentation files in Pericles was implemented consistently with the principles of the model-view-controller pattern. The pattern enforces great *separation of concerns* which results in low coupling between the different software components (i.e. views, models and controllers). This modularity is directly translated in the fact that several different views can easily be provided for the same model.

Therefore, in Pericles, the documentation for a *project* (i.e. an API) simply consists in a collection (which can be downloaded as a .zip file) of Markdown files, one for each *resource* in the *project*. Each of these files is just another representation (a view) of a *resource*, just like the HTML documents mentioned above. Finally, the collection itself can be seen as a representation of the *project* (in Markdown) as well.

### 3.4 Timings and project organization

This section presents the actual timings for the project through a Gantt chart. **At the beginning of the project**, the work on Pericles was divided into the following features:

1. Create *projects* inside the application, and add JSON Schemas related to these projects (simply as text, through forms). Users could then access projects and consult the associated JSON Schemas.
2. Edit and delete existing projects and JSON Schemas.
3. Be able to describe *resources* (see section 2.3.2) inside projects, as well as to add *routes* to projects and make it possible to associate JSON Schemas and routes.

The functionalities described above constituted the minimum to be implemented as part of this internship at Applidium. The features below were considered as stretch goals.
1. From the information entered in the tool, automatically generate files in Markdown to be used to create documentation pages for the projects.

2. Integrate existing libraries and tools related to JSON Schema in the project (see section 2.4.2).

3. From the information entered in the tool, automatically generate configuration files for WireMock. Mocks, through WireMock [33], simply return mock answers so that, notably, front-end developers can test their side of the application while the back-end itself is being designed.

4. From the information entered in the tool, automatically generate Postman collections to test the API. Postman [34] lets users create test *collections* that can gather requests to routes. When a collection is run, all requests contained in the collection are performed automatically as a means to test routes and thus the API. It is therefore possible to see easily if an API displays an unexpected behaviour.

5. Development of a proxy between clients and external back-ends (not developed by Applidium) that would keep track of all exchanges (HTTP requests and responses) between those.

6. Development of an algorithm for the proxy which, using the ”intercepted” HTTP requests and responses, would be able to determine and update the JSON Schema corresponding to the request and responses for each route in the API. This would mean that the descriptions of the routes would be based on what is actually returned by the back-end, dynamically.

The Gantt chart below illustrates the time actually spent on each of these features. In the chart, a week corresponds to 5 working days. Besides, this chart can be compared to the one illustrating the planned timings before the project, available in appendix B.
The main difference with what was originally planned is the time spent working on the project. Indeed, the original chart made the wrong assumption that the full internship would be dedicated to this master thesis project.

The intention in the original planning was to grant more time than necessary to each task in case something went wrong. Tasks I (took 5 days when 15 days were planned) and 1 (took 4 days when 20 days were planned), for instance, were really overestimated. The complexity of task 1, notably, was misjudged.

The validation ensuring that only valid JSON schemas (cf section 3.3.1) can be persisted, as well as the validation API and its integration (sections 3.3.2 and 3.3.3, respectively) were implemented as part of task 2, which was the most underestimated of all. The plan was for tasks 2 and II to take 10 days, when task 2 alone took 15 days. There are multiple reasons for this:

- The described features are all based on the Ruby JSON Schema Validator gem. Finding a gem (or library) that fulfills a given purpose takes time. Not only must the gem provide the wanted functionalities, but it should also be preferably still maintained. Furthermore, once the gem is chosen, integrating it in a project can also be time-consuming. Here, learning how the gem works required to dive into its code, because the documentation was not always complete.

- The way the validation API was implemented was changed over time, before reaching the final result described in section 3.3.2. The modifications were
made so that the implementation was clearer, and adhered more to REST principles.

- The integration of the API also required time, because it involved to understand the basics of jQuery and how to incorporate JavaScript in a Rails project.

The main goal of the internship, which was to implement tasks I, II and III, was achieved (the milestone is represented as a black square in the Gantt chart). This allows users of Pericles to describe APIs (cf section 3.3.1).

It is interesting to consider how what was going to be implemented, and how it was going to be done, was refined as the project evolved and time passed. For instance, Pericles originally had a JsonSchema model and the corresponding json schemas table in the database so as to store JSON Schemas. At the end of Task I, Pericles offered to add json schemas directly into projects. During week 7 of development, the JsonSchema model was removed from the project, because it did not make sense anymore, since JSON Schemas would be stored as shown in figure 6.

The project was divided into features so as to add functionalities step by step, instead of trying to plan and do everything at once. The next feature would thus be added only once the current one was finished. The example of JSON Schemas above illustrates that this comes with the price of having to deconstruct things, sometimes, when a new feature needs to be added. However, it allows to always have a functioning prototype.

### 3.5 Current limits

As of now, Pericles suffers from several limits due to the time constraints that were imposed for its development. First, and most obvious, the features corresponding to tasks 3, 4, 5 and 6 (cf section 3.4) could not be implemented during this internship at Applidium.

Furthermore, creating Pericles involved both front-end and back-end development, since it is a web application. During the development of each feature, the focus was mainly set on the back-end side of things. Even though a consequent amount of time was still spent on views (using Bootstrap, which is a framework for front-end web development) to make the user experience as satisfying as possible, the user interface itself is minimal, compared to what is proposed in Restlet Studio, for instance. The reasoning that motivated this choice was that it is better to have features that are implemented properly, i.e. that are fully functional, with a modest design, than dedicating too much time on enhancing the user experience,
at the expense of the implementation of new functionalities and of how these are implemented.

Besides, as explained in section 3.3.1, Pericles includes a modeling of HTTP requests and responses, which could be made more precise. For instance, the status_code for responses could be chosen from the list of existing status codes specified by [5] and more recent RFCs, whereas currently, any integer entered by the user can be persisted as a status code. Likewise, standard field-names could be suggested for the name attribute of headers.

### 3.6 Benefits and sustainability

The Brundtland definition of sustainable development ("meet the needs of the present without compromising the ability of future generations to meet their own needs", cf [35]) introduces equity between generations as a major part of what sustainable development embodies. The authors of [35] reformulate the Brundtland definition of sustainable development as follows: "making sustainable use of our planet to maintain its function of fulfilling human needs.". Therefore, we can, according to [35], highlight the two aims of sustainable development as "making sustainable use of our planet" and "fulfilling human needs". However, if sustainable development aims at avoiding to endanger the possibility to live a decent life in the future [35], the fact that all human beings should also be able to reach that goal in the present is also an issue that it tackles. The value of equity is expanded in time and space, so as to be able to "fulfill human needs" in any case.

Therefore, sustainable development also spearheads environmental values, as a result of its wish for equity. Indeed, our unsustainable model for development and growth has had an irreversible impact on the environment. Future generations, because of global warming, for instance, may have to deal with environmental disasters on a regular basis. To ensure that these generations will be able to live decently (equity), sustainable development suggests "limitations" that need to be applied to our impact on the environment, and that should be compatible with the growth and development necessary to fulfill our "needs" [35]. This is what is meant by "making sustainable use of our planet".

Applidium is a startup with great momentum. Since its inception in 2010, it has been continuously growing in size to reach about 50 employees today. Furthermore, the company's expertise was originally focused on making applications for the iPhone with iOS. However, the activity at Applidium is, nowadays, much more diversified, since applications for Android as well as back-ends are produced. This consequent growth obviously goes together with expanding needs in order to sustain it.

So as to produce more value per time unit, more energy (in the form of electricity to power computers) is spent in the effort to "meet the needs of the present".
In this context, the outcome of this master thesis constitutes a very interesting tool. First, it allows to centralize information used by developers and project managers while working on projects, which enhances the overall organization during these. Second, it can also be used to automatize time-consuming tasks that used to be manually performed by the same people. Therefore, it becomes clear that the system allows employees at Applidium to increase their efficiency. This means that less energy (i.e. electricity) can be used to maintain same production levels. The energy spent during the development of this project can thus be seen as an investment in terms of sustainable development, and such an effort could be categorized as Green Software engineering [35].

Furthermore, the system was launched as a website. This constitutes a guarantee, in terms of obsolescence. The solution will keep evolving, as new functionalities will be added after the end of this master thesis. The principles of REST, explained in section 2.3.2, ensure that change will always be possible and will happen gracefully. Even when the development on the project eventually stops, it will still be available through any existing browser, as long as it is hosted online. Indeed, the web has been based on the same technologies (HTML, JavaScript...) for many years [14], and this is likely to continue. Thus, the resources spent on this master thesis project are not likely to be eventually "lost", in this way.

Besides, it is hosted on a platform called Heroku, on which websites go to sleep if no traffic is received during 30 minutes. This environmentally-friendly behaviour can also be seen as "making sustainable use of our planet", in order to make sure that future generations can fulfill their needs as well.
4 Evaluation

In order to evaluate the relevance of Pericles in relation with the goals listed in section 1.4 and, thus, the questions stated in section 1.2, a qualitative evaluation was performed. Applidium employees who had tested the different features implemented in Pericles, at the time of writing, were invited to fill a questionnaire. The answers were gathered anonymously, and the only question tackling the profile of the respondent targeted his or her profession at Applidium. Table 1 displays the different questions asked as part of this questionnaire, and presents the results of the survey.

Six employees took the questionnaire, among them 2 project managers, 2 front-end developers, and 2 employees that work both on front-end and back-end development. Only questions number 1, 2 and 3 were mandatory, and the last column on the right of table 1 indicates the number of answers for each question.

The different questions aimed at probing the feelings of the respondent about Pericles in relation with the goals listed in section 1.4, by quantifying them through a numbered scale from 1 to 4. For each question, 1 was interpreted as ”not at all”, and 4 meant ”absolutely”, except for question 5, where 1 signified ”the quality has decreased” and 4 ”the quality has increased”.

<table>
<thead>
<tr>
<th>Question</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Questions and results of the survey.
<table>
<thead>
<tr>
<th>Qu. #</th>
<th>Question</th>
<th>Answers</th>
<th># of re-sps.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Would you say that Pericles allows to describe APIs easily and exhaustively ?</td>
<td>0 1 3 2</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>Would you say that Pericles allows to integrate JSON Schema in projects at Applidium adequately ?</td>
<td>0 1 3 2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>Do you think that Pericles allows to enhance the centralization of information related to APIs at Applidium ?</td>
<td>0 0 2 4</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>Do you consider that Pericles, through the automatic generation of exports (e.g. documentation), notably, allows you to work on projects in a more efficient way ?</td>
<td>0 1 4 1</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>According to you, what is the influence of the automation of the generation of exports on their quality ?</td>
<td>0 0 5 1</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>Would you say that Pericles is helpful to specify and document external back-ends at Applidium, and makes these processes more efficient ?</td>
<td>0 0 1 3</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>Do you think that Pericles allows you to optimize your exchanges with external developers in the context of projects at Applidium ?</td>
<td>1 0 1 2</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 1: Results of the survey performed at Applidium
As explained above, the questions in table 1 were designed with each of the goals of section 1.4 in mind. Overall, the results depicted in the table reveal that the system developed as part of this master thesis, Pericles, fulfills these goals convincingly, according to Applidium employees who took part in the questionnaire. Indeed, for each question, at least 75% of respondents answered 3 or 4 (i.e. answers that convey positivity in relation to the aspect of Pericles being tackled).

The results for questions 1 and 3 show that a majority of people who tested Pericles believe that the system constitutes a valid option to centralize API-related information in the company. It also suggests that the way proposed by Pericles to describe APIs was deemed as user-friendly by the respondents.

Currently, the documentations for back-ends at Applidium use sample JSON instances to specify the representations of resources to be included in the message-body of HTTP requests and responses, which constitutes an issue, as evoked in section 1.2. Indeed, these JSON instances may not be fully representative of the structure of all possible representations that could be included in the HTTP messages. For instance, a certain key found in the example provided in the documentation for a HTTP response could not appear in the JSON instance actually returned by the backend, because it is optional. Section 2.1.2 explains that using JSON Schema fully solves this problem, and section 3.3.1 underlines that Pericles indeed incorporates JSON Schema to this very end. The issue could thus (rightfully) be considered as solved, but question 2 goes a bit further by investigating if users of Pericles are satisfied by the way it integrates JSON Schema (which also includes the features related to the validation API described in section 3.3.3, for instance).

Question 4 could be put in perspective since Pericles currently allows to automatically generate only documentation files from the information concerning APIs entered in the tool. It is likely that the overall appeal of the automation of file generation provided by Pericles will increase with the number of file types (Postman collections, configuration files for WireMock...) that can actually be generated automatically. Likewise, the feature should then ignite the interest of a range of people at Applidium that encompasses more diverse roles in the company. The results of question 5 interestingly point out that respondents unanimously consider that the quality of documentation files for back-ends has increased since their generation was automatized.

Respondents were only invited to answer questions 6 and 7 if they had got to try Pericles in the context of a project that involved an external back-end, which explains why less answers were obtained than in previous questions. The fact that some external APIs do not come with complete or accurate documentation, or
that certain external API providers do not implement what was planned during workshops represents a major issue for Applidium, because it ushers in a consequent loss of time. Table 1 shows that Pericles was very well received as a way to curb these effects. It indicates that the tool is valued to specify external APIs with their implementers, before the development or while it is happening, for instance.

Still, the fact that a few respondents chose to answer 1 or 2 (i.e. answers that convey negativity) for certain questions highlights that Pericles is actually a work in progress (as indicated in section 1.6) which currently suffers from limits, mentioned in section 3.5.

Besides, it is interesting to point out that the results did not convey any obvious or pronounced difference of opinion regarding any of the questions shown in table 1 between the different ”groups” of respondents (i.e. project managers, front-end developers, and front-end and back-end developers), separated according to their profession at Applidium. This suggests that Pericles finds its use uniformly among these groups. Indeed, most negative opinions came from one particular individual (a project manager), who answered 2 for questions 1 and 2, 1 for question 7 and 3 to all other questions, which could suggest that project managers deem Pericles less helpful than the other groups of respondents (especially concerning the goals related to questions 1 and 7). However, the other project manager who filled the questionnaire answered 3 to questions 1, 4 and 7, and 4 to all other questions. This last conclusion should therefore be nuanced.

Finally, if the results globally reflect that Pericles represents a valuable tool for Applidium, it should be underlined that their scope could nonetheless be limited, due to the small number of respondents.
5 Conclusion

The present master thesis constituted an attempt to ease the life of Applidium’s developers and project managers by improving aspects of the company’s production cycle. Having to deal with external back-ends that are delivered without accurate or exhaustive documentation, for instance, embodied one of the issues identified in section 1.2 as a hinderance while working on projects. As a way to solve these problems, section 1.4 shaped a tool in the form of a list of goals. The solution was imagined as a system centralizing all API-related information at Applidium, by allowing its users, notably, to describe APIs through a user-friendly interface. Pericles, as depicted in section 3, came as the embodiment of the vision detailed in section 1.4, and serves as the outcome of this master thesis. The qualitative evaluation conducted on Pericles after its inception, related in section 4, suggested that, overall, Applidium employees who got to test and work with the created tool deem that the implemented features fulfill the goals that guided their implementation. It can thus be considered that Pericles optimizes the development and validation of projects at Applidium.

However, this evaluation also highlighted that the presented version of Pericles can only be seen as the first step of a larger project that will continue after this master thesis. For instance, the prospect of an algorithm able to automatically describe APIs based on intercepted HTTP messages (see section 1.4) triggers great interest among Applidium employees. Furthermore, features that were not originally part of the scope for this master thesis, such as an authentication system, may also be added to Pericles.

On a more personal note, this project, and my internship at Applidium in general, was truly fruitful in terms of learning experiences. By working as a back-end developer on Pericles, but also on projects for Applidium’s clients, I could deepen my technical knowledge on Git, HTTP, APIs, database design, the Ruby language and the Ruby on Rails framework. My organizational and communication skills were also put to the test by having teammates relying on me while working on projects, for instance.

Furthermore, I was constantly reminded that there is no ”perfect” answer in engineering, only trade-offs. Indeed, working for clients, with deadlines, showed me that being an engineer is about solving problems with constraints. The projects I worked on were thus as many opportunities to learn to make enlightened and justified choices that best fit a given situation. Besides, I also realized, more than ever, that time is a key value, and that making sure that the task that needs to be achieved is clearly defined and understood is essential to avoid putting effort into things that were not needed in the first place, and to focus, instead, on what actually matters. Therefore, I want to thank my manager, Emilie Paillous, and
the whole team at Applidium for accompanying me during 6 months, and for their warm welcome.
References


Appendices

A JSON Schemas for the validation API

The following JSON schemas are used in Pericles to describe the structure of the representations of validations that are exchanged in HTTP messages in the context of the validation API (section 3.3.2).

A.1 In HTTP requests

```json
{
    "type": "object",
    "properties": {
        "validation": {
            "type": "object",
            "properties": {
                "json_schema": {
                    "type": "string"
                },
                "json_instance": {
                    "type": "string"
                }
            }
        },
        "required": ["json_schema", "json_instance"
        ]
    },
    "required": ["validation"
    ]
}
```
A.2 In HTTP responses

```json
{
   "type": "object",
   "properties": {
      "validation": {
         "type": "object",
         "properties": {
            "json_schema": {
               "type": ["string", "null"],
            },
            "json_instance": {
               "type": ["string", "null"],
            },
            "status": {
               "type": "string"
            },
            "json_errors": {
               "type": "array",
               "items": {
                  "type": "object",
                  "properties": {
                     "description": {
                        "type": "string"
                     },
                     "required": ["description"]
                  }
               }
            }
         }
      },
      "required": ["json_schema", "json_instance"]
   }
}
```
"status",
"json_errors"
]
}
}
"required": [
"validation"
]
B Planned timings at the beginning of the project

This section illustrates the timings that were originally planned before the beginning of project through a Gantt chart. The numbers that are used to identify the tasks refer to those used in section 3.4.

<table>
<thead>
<tr>
<th>Weeks</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tasks II and 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task III</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tasks I, II and III achieved</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
C Screenshots

This section gathers screenshots taken from the outcome of this master thesis, Pericles, which was introduced in section 3.

![Screenshot of Pericles interface](image)

Figure 8: Representation of a resource as a HTML document in Pericles
Figure 9: Representation of a route as a HTML document in Pericles
Figure 10: Example of validation of a JSON instance against a JSON Schema in Pericles
Applidium est une startup localisée à Paris, spécialisée dans le développement d’applications mobiles. Le fonctionnement de ces dernières se divise en deux parties, le front-end et le back-end, qui communiquent entre elles via HTTP. Le front-end correspond à la partie de l’application avec laquelle l’utilisateur peut directement interagir sur son téléphone. Au contraire, le back-end est incarné par tous les composants qui ne sont pas visibles pour l’utilisateur, et qui visent à servir les requêtes envoyées par le front-end. Le back-end se présente sous la forme d’une API (Application Programming Interface, ou interface de programmation applicative), c’est-à-dire sous la forme d’une interface logicielle fournissant un certain nombre de services.

Il y a en permanence de multiples projets en développement chez Applidium, de façon concurrente, et l’entreprise recherche constamment des solutions pour améliorer les différents aspects de son cycle de production. Par exemple, certains projets impliquent l’utilisation d’APIs qui ne sont pas développées par l’équipe d’Applidium dédiée au back-end. La documentation pour celles-ci n’est parfois pas exhaustive ou exacte, ce qui rend le développement laborieux.

Un autre problème concerne le format actuel utilisé pour décrire les données incluses dans les messages HTTP (dans le message-body), dans la documentation pour les back-ends créées par Applidium. Cette information est utilisée par les développeurs front-end pour savoir quoi inclure dans les requêtes, et à quoi s’attendre dans les réponses. Actuellement, la documentation fournit un certain nombre d’exemples pour décrire ces données, ce qui, possiblement, n’est pas représentatif de toutes les instances pouvant être incluses dans ces messages HTTP. Ce problème est également rencontré dans des APIs externes de façon régulière.

Enfin, les développeurs d’Applidium ont recours à un certain nombre d’outils pour tester et documenter les différentes parties d’un projet. Ceci prend actuellement beaucoup de temps et peut mener à des erreurs, puisque tous les fichiers de configuration pour les tests, ainsi que les fichiers de documentation sont écrits manuellement. Par ailleurs, le contenu de tous ces fichiers est largement basé sur la même information, à savoir la description des APIs utilisées dans le contexte du projet. Ceci signifie qu’ultimement, différentes versions de la même information, éparpillées dans différents fichiers, peuvent coexister.

Ce rapport s’intéresse ainsi aux questions suivantes: Comment le développement et la validation de projets chez Applidium peuvent-ils être optimisés à travers la création d’un outil interne ? Quelles fonctionnalités devraient-êtres incorporées à l’outil en question ?
Le délivrable attendu pour ce projet est donc un outil trouvant son utilité à la fois lorsque le back-end d’une application mobile est développé en interne par Applidium, ou par un prestataire. Un certain nombre de buts pour celui-ci sont déterminés comme solutionnant les problèmes identifiés ci-dessus:

Dans le cas de back-ends développés en interne par Applidium

- Le système permet, avant tout, de décrire des APIs et vise à rassembler toutes les informations liées aux APIs chez Applidium.
- Un aspect important est de pouvoir spécifier la structure des données incluses dans les requêtes et réponses HTTP échangées entre le front-end et le back-end, le format choisi à cet effet par Applidium étant JSON Schema.
- La solution est conçue pour permettre la génération automatique de fichiers de test et de documentation à partir des informations entrées dans l’outil par ses utilisateurs.

Dans le cas de back-ends développés par des prestataires externes

- Lorsque le back-end est déjà développé, l’intention derrière le système est d’assister les développeurs et chefs de projet d’Applidium afin de documenter l’API, lorsque la documentation fournie est incomplète, ou inexacte.
- Si le back-end est en cours de développement, l’objectif est plutôt de permettre de spécifier l’API, et de jouer le rôle de ”contrat” entre les équipes d’Applidium et le prestataire. La solution est donc envisagée comme un outil incorporant une interface utilisateur qui rend son utilisation aisé et efficace.

La vision du système détaillée ci-dessus est matérialisée par Pericles, qui constitue le délivrable de ce projet. Pericles est une application web développée en Ruby on Rails et JavaScript, ayant pour objectif de concrétiser les buts décrits précédemment.

Une évaluation qualitative est conduite pour évaluer la pertinence de Pericles par rapport aux buts qui ont guidé son implémentation. Cette évaluation suggère que les employés d’Applidium qui ont pu essayer et travailler avec l’outil proposé considèrent que les fonctionnalités présentées implémentent bien les buts suscités, et donc que Pericles permet d’optimiser la validation et le développement de projets chez Applidium. Par ailleurs, les résultats soulignent également que Pericles constitue un travail en cours, dont l’ambition, pour Applidium, dépasse le cadre de ce projet de fin d’études. En effet, du fait des contraintes de temps imposées pour le projet, certaines fonctionnalités n’ont pas pu être implémentées pour le moment, et le seront donc prochainement, puisque le développement de Pericles est destiné à continuer après la fin de ce stage.