Rodrigo Filipe Mendes dos Santos

DYNAMIC RESOURCE FRAMEWORK

1 2 9 0 UNIVERSIDADE D COIMBRA

UNIVERSIDADE D
COIMBRA

Rodrigo Filipe Mendes dos Santos

DYNAMIC RESOURCE FRAMEWORK

Internship Report in the context of the Master in Informatics Engineering, Specialization in Communications, Services and Infra-structures, advised by Professor Filipe Araujo from the Department of Informatics Engineering and Pedro Verruma and Gonçalo Pereira from Talkdesk and presented to
Faculty of Sciences and Technology / Department of Informatics Engineering.

January, 2021

Faculty of Sciences and Technology

Department of Informatics Engineering

Dynamic Resources Framework

Rodrigo Filipe Mendes dos Santos

Dissertation in the context of the Master in Informatics Engineering, Specialisation in Communications, Services and Infra-structures advised by Prof. Filipe Araujo from the Department of Informatics Engineering and Pedro Verruma and Gonçalo Pereira from Talkdesk and presented to the
Faculty of Sciences and Technology / Department of Informatics Engineering..

January, 2021

1 2 9 0

UNIVERSIDADE D
COIMBRA

This page is intentionally left blank.

Acknowledgements

As this is a more personal part of the document, I will write the remainder of it in Portuguese.

Em primeiro lugar quero agradecer aos meus pais, António Pereira e Margarida Cardoso, não só por todo o apoio que me deram durante este estágio, mas também por toda a ajuda que me deram durante toda a vida, sem eles não teria conseguido alcançar tudo o que consegui.

O meu obrigado aos meus orientadores, Prof. Filipe Araújo e Pedro Verruma pela ajuda, apoio, dedicação e conhecimentos que me transmitiram ao longo deste estágio.

Por todo o apoio técnico, tempo gasto, paciência e acompanhamento extraordinário durante o desenvolvimento do projeto, um especial agradecimento ao Gonçalo Pereira.

E por fim, quero agradecer a todos colegas da Talkdesk, por todo o apoio que me deram durante o estágio.

This page is intentionally left blank.

Abstract

Based on the micro-services architectural model, Talkdesk's infrastructure is now a complex system, ever-growing in the number of services and clients.

Most of the implemented clients responsible for manipulating resources data have a common codebase, differing only in hard-coded operations: how they show the resource forms and how to execute the actions (endpoint URL and HTTP method). This kind of implementation is hard to maintain because it implies a refactor on the client codebase when a resource structure is updated.

This work and document address the issue, proposing a framework architecture capable of handling resources based on high-level representations and improving service responses with meta-information regarding subsequent possible actions (endpoint URL and HTTP method).

We can then create a generic client capable of discovering available service APIs, how to invoke them and render the resources and associated forms based on its high-level representation.

In the end, this internship was completed successfully. We were able to fulfil the two main goals: have the first implemented version of the framework; decouple the client from the service implementation.

The current version of the framework already abstracts multiple operations, that otherwise would require extra development time.

The client implements generic UI elements that are only displayed/called when the server's response contains the necessary information.

Right now the framework is usable, but it is still in his early development state. We can solve some of the configuration-related flows to improve usability.

Keywords

Resource, REST, API, Micro-Service, Data Representation, Schema.

This page is intentionally left blank.

Resumo

Na Talkdesk, a infraestrutura segue um modelo arquitetural baseado em micro-serviços. Atualmente esta infraestrutura está cada vez mais complexa, existindo um número crescente de serviços e clientes.

A maioria dos clientes implementados são utilizados para manipular recursos contidos nos serviços. Estes clientes partilham a maior parte do seu código, exceto algumas partes que estão *"hard-coded"*, por exemplo: os formulários que os utilizadores podem preencher e como executar as operações sobre os recursos (URL e informação sobre o método HTTP a invocar). Este tipo de implementação tornou-se difícil de manter porque sempre que existe uma alteração no sistema isso implica alterações nos clientes.

Este trabalho serve para apresentar a arquitetura para uma *"framework"* capaz de lidar com os recursos utilizando uma representação de alto nível. A *"framework"* também será capaz de adicionar meta-informação às respostas enviadas pelo serviço.

Com esta informação extra, podemos criar clientes genéricos capazes de descobrir as APIs de um serviço, como as invocar, gerar os formulários baseados na representação de alto nível de um recurso.

Podemos afirmar que o estágio foi concluido com sucesso. Conseguimos concluir com sucesso os dois principais obectivos: ter a primeira implementação da *"framework"*; desacoplar os clients da implementação do serviço.

A versão atual da *"framework"* já permite uma abstração de algumas operações que necessitariam de uma alocação de tempo de desenvolvimento.

O cliente implementa um *"UI"* genérico, os elemento apenas vão aparecer quando o serviço envia a informação necessária na resposta.

Apesar de utilizável, a *"framework"*, ainda está num estado muito inicial do seu desenvolvimento. Ainda é possível alterar alguma da lógica relativa a configurações para que a sua utilização seja mais fácil.

Palavras-Chave

Recurso, REST, API, Micro-Serviço, Representação de dados, *"Schema"*.

This page is intentionally left blank.

Contents

Acronyms

API Application Programming Interface. 17–21, 33, 35, 37, 38, 47, 57, 61, 62

DTD Document Type Definition. 11, 13

HAL Hypertext Application Language. ix, xiii, 17, 18, 23, 33–35, 38–41, 45, 61, 62, 69, 70

HATEOAS Hypermedia as the engine of application state. 15

HTTP Hypertext Transfer Protocol. 14, 16, 20, 33

HYDRA Hypermedia-Driven Web APIs. ix, 17, 20

IANA Internet Assigned Numbers Authority. 34

IRI Internationalised Resource Identifiers. 19, 20

JSON JavaScript Object Notation. ix, 5, 7–12, 17–20, 23, 33, 36, 37, 39, 41, 42, 44, 47, 49, 61, 67, 69

JSON-LD JavaScript Object Notation for Linked Data. ix, 17–20, 23

OS Operating System. 5

REST Representational State Transfer. ix, 3, 5, 13–17, 20, 25, 33, 61

ROA Resource Oriented Architecture. 17

RPC Remote Procedure Call. ix, 13, 14, 16

SGML Standard Generalised Markup Language. 5

SMTP Simple Mail Transfer Protocol. 14

SOAP Simple Object Access Protocol. ix, 13, 14, 16

SOX Simple Object XML. 11

URI Uniform Resource Identifier. 15, 18, 21, 22

URL Uniform Resource Locator. 19, 20, 34, 37, 38, 51, 61

W3C World Wide Web Consortium. 10, 13, 18

XML Extensible Markup Language. ix, 5–7, 10–14, 16–18, 70

XSD XML Schema Definition. ix, 10–12, 66

YAML YAML Ain't Markup Language. ix, 5, 7–9

This page is intentionally left blank.

List of Figures

This page is intentionally left blank.

List of Tables

This page is intentionally left blank.

Chapter 1

Introduction

This chapter gives the reader a brief introduction to this project. The section 1.1 contains a simple description of the project where this internship is integrated. The section 1.2 explains the current problem of the system, while the internship motivation is explained in section 1.3. Finally, this internship's objectives are explained in section 1.4, and section 1.5 has a brief description of the document structure.

1.1 Project overview

The Agent Assist project is the first attempt of Talkdesk in developing an AI-powered virtual agent with the objective of decreasing the time that a call-center agent loses while searching their knowledge base during a call. Agent Assist listens to the agent call, processes it and returns a set of recommendations. This process is handled in real-time.

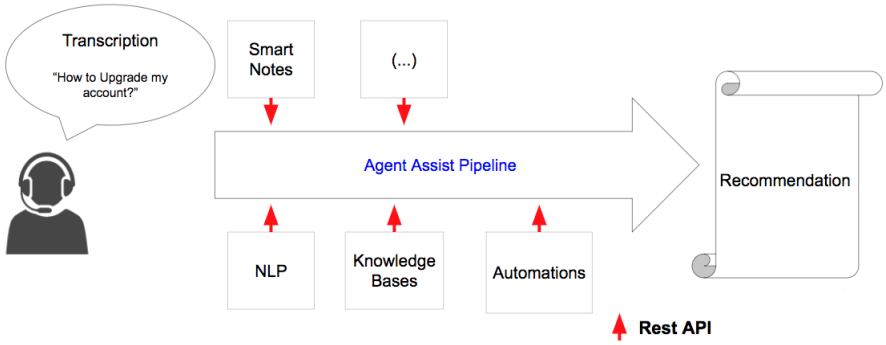


Figure 1.1: Agent Assist Pipeline

Agent Assist is a processing pipeline made up of a set of modules, figure 1.1. During a call the pipeline receives pieces of the conversation between an agent and a client (transcriptions). Then the transcriptions follow the pipeline path, and each module is responsible for performing one of the basic operations of the service. As a growing project, Agent Assist's complexity (the number of modules) is increasing.

1.2 Problem

The clients can show and manipulate resource data. How the client renders and handles resource operations is hard-coded. This kind of implementation is hard to maintain because implies a refactor on the client codebase when a resource structure is updated.

Most of the clients we have are somewhat similar; they share most of the structural code. The differences are mostly related to rendering the resources and invoking the endpoints that will perform the action.

For example, let us say that we have two different services, one that handles user information and the other handles fruits. The user is defined by a name, a surname and age and the fruit resource have a name, an origin code, and a date when the fruit was picked.

The two resources are entirely different. Suppose we wanted to create a frontend page with a form to insert a new element in the service. We would need to make the page, map each resource field to a form field, and create a button to call the endpoint to persist the resource.

After that, we would effectively have two clients that do the same thing but that are different because they have the resource form and the request endpoint hard-coded.

Another problem that we have is related to the services resource representation. In Agent Assist's current implementation, data objects are defined using polymorphic techniques. This approach allows us to have fewer objects to maintain but also increases the data validation and processing complexity, leading to more complex code that needs to be developed to perform simple system modifications and implement new features.

1.3 Motivation

As Agent Assist is growing, we are developing more modules to be plugged into the pipeline. The developed modules will have one or more resources (with different data structures) associated with them.

We want to decrease the time that we take in the development of the new modules. One place where we think it will be possible to reduce this time is in developing frontend clients.

One possible approach to that is creating a generic client capable of handling multiple resource types. This generic client might be possible if we define the forms based in a generic resource definition. The client must also know what operations are available and how to execute them because the interactions with the service that previously would be hard-coded must be removed.

1.4 Objectives

This internship's primary goal is to propose an architectural design and implement the first version of a framework to develop our services.

We want to find a way to describe the resources. It must contain the resource structure like variable names, and also the resource constraints, like variable "A" is an integer between 5 and 10, and variable "B" is a string.

As we want to deal with multiple types of resources, the framework must handle any resource. The clients will work with a high-level resource representation, and because of that, the framework must provide a way to store and deliver those representations.

Clients do not know which operations they can invoke and how to call them. The framework must handle the service response and expand it with the information necessary for the client to know how to perform operations on the resources.

Lastly, the framework will also be able to validate the resource data using the generic representation.

1.5 Document structure

This document will cover the internship's work, such framework requirements analysis, specification, project planning, development and conclusions, and is organised as follows. The current chapter describes the context, the problem and the main goals of this project.

The next chapter will cover some of the research about, data representation, data structure and validation (schemas), architectural communication patterns for web services, and discuss and compare various standard implementations of Representational State Transfer (REST)ful web services. And the third chapter will cover the system requirements (both functional and non-functional) and constraints (both business and technical). After that, the system architecture is presented, described and, in the end, we have the risk analysis and mitigation plan.

The fourth and fifth chapters will cover the implementation done during the internship. First, we will cover the framework specific implementation, and in the other, we will describe the case study and how we used the framework to implement it.

Lastly, we have the planning and conclusions chapters. In the planning chapter, we describe the tasks and time they took to be developed. The last chapter contains a summary of the work done and the project conclusions.

This page is intentionally left blank.

Chapter 2

Background and State of the Art

In this chapter, we set the stage on the various standards and frameworks used in developing services and web applications. We cover various topics, like data representation, data structure and validation, architectural communication patterns for web services (such as Representational State Transfer (REST)) and lastly we discuss and compare various standard implementations of RESTful web services.

2.1 Data representation standards

Data representation standard is the definition of a common language used to describe, store and transport complex data structures.

These standards address the issue of representing heterogeneous data in a format that everyone can equally understand and possibly represent the same information in different environments (programming language, Operating System (OS)...).

We will cover some of the most used standards, like Extensible Markup Language (XML), JavaScript Object Notation (JSON) and YAML Ain't Markup Language (YAML). We choose those standards because of popularity and for defining languages that are human-readable.

2.1.1 Extensible Markup Language (XML)

According to [6], XML is a derivation of the Standard Generalised Markup Language (SGML) and thus XML documents maintain the same overall structure as SGML documents.

The XML is a meta-language (describes other languages), allowing the user to specify the markup language for each document class. In contrast, the SGML describes a document structure and content generically, having an endless variety of document structures, this makes the SGML a very complex standard for data representation [22].

In XML, we represent the information as a set of entities and values. Entities are enclosed by an opening tag `<entity>` and a close tag `</entity>` . The entity value must be present between the tags.

Entities can also have attributes, that are key-value pairs. For a given entity, the defined attributes must have unique keys.

All XML documents follow the same structure:

Listing 2.1: XML document base structure

```
<root>
  <child1>
    <subChild1>...</subChild>
    (...)
    <subChildn>...</subChild>
  </child1>
  (...)
  <child_n>
    value
  </child_n>
</root>
```

The elements used to describe a XML can represent the document like a tree graph and the notation used is the parent and child notation, as seen in 2.1. Each parent can have one or more child nodes, and a child can only have one parent, nodes that belong at the same level are siblings.

For example, if we wanted to represent a simple Java model with user information, first and last name and age:

Listing 2.2: Simple Java Class

```
public class Users {
    private List <User> users;
}

public class User {
    private UUID id;
    private String firstName;
    private String lastName;
    private int age;
}
```

The corresponding XML representation can be something like this:

Listing 2.3: XML Representation

```
<?xml version="1.0" encoding="UTF-8"?>
<Users>
    <user id="974e7b5c-fd64-4bfe-92a5-ae60ad16d4bd">
        <firstName>Rodrigo</firstName>
        <lastName>Santos</lastName>
        <age>30</age>
    </user>
    <user id="94135e0a-0778-4cb2-8982-80e3e7bdea21">
        <firstName>John</firstName>
        <lastName>Doe</lastName>
        <age>100</age>
    </user>
</Users>
```

We have the `<Users>` root node containing multiple `<user>` (list elements). Each `<user id="...">` has an `id` attribute and 3 children, `<firstName>` , `<lastName>` , `<age>` .

2.1.2 JavaScript Object Notation (JSON)

JSON is a lightweight and straightforward data representation format. Like XML, a JSON document is human-readable, has a structure that is easy to understand, uses an extensible language and is system independent.

The JSON language assumes the mapping paradigm, where each element is a key, and each key has a value pair. Each key must always be a string without spaces. The values, on the other hand, can assume one of the following types:

- Object - a non-ordered set of key-value pairs bounded between two curly brackets `{...}` ;
- String - a character sequence that appears between double-quotation marks, `"..."` ;
- Number - a numeric value. It can be an integer, or it could also include a floating part;
- Boolean - either one of the values `true` or `false` ;
- null - empty value, using the word `null` ;
- Array - a set of ordered elements that can be of any of the supported types. The array elements appear between two square brackets `[...]` .

Using the 2.2 as a reference, the resulting JSON document is something like this:

Listing 2.4: JSON representation

```
{
    "users" : [
        {
            "id" : "974e7b5c-fd64-4bfe-92a5-ae60ad16d4bd",
            "first_name" : "Rodrigo",
            "last_name" : "Santos",
            "age" : 30
        },
        {
            "id" :"94135e0a-0778-4cb2-8982-80e3e7bdea21",
            "first_name" : "John",
            "last_name" : "Doe",
            "age" : 100
        }
    ]
}
```

Comparing the XML 2.3 and the 2.4 representation, JSON needs less information to describe the data structure of the serialized information, this means that a JSON representation is less verbose and that it will have less impact on message size [21].

2.1.3 YAML Ain't Markup Language (YAML)

According to [2], YAML is a human-friendly data serialization standard for all programming languages.

The YAML language was created to be easily readable by humans, allow for data portability between programming languages, have a structure that can be parsed using a single processing passage, expressive and extensible and also to be easy to implement and use. The design of this language had in mind the following use cases:

- Representation of configuration files;
- Log files;
- Messages between processes;
- Data share in multi-language systems
- Object persistence;
- Represent complex data structures.

The YAML syntax is similar to the python syntax, and it can use indentation to define nested information or use a language more similar to JSON.

Part of the YAML syntax is as follows:

- Whitespace - the indentation in YAML is done using `whitespaces` before the elements;
- List - a list of elements is a set of elements that begin with the minus character (`-`) each in a single line. Alternatively, we can also represent a list with single line elements, placed between square brackets and separated by commas (like `[value1, value2, ...]`);
- Associative Array - is used to define key-value pairs, the key is followed by `:<whitespace>` then the value, one per line. Like the lists there is another possible representation, the key-value can have a one-line representation like `{key1: value1, key2: value2}` .
- Strings - are unquoted sequences of characters, but the string values can also have double quotes (`" "`) or single quotes (`' '`). When using double quotes, the string can have special characters escaped using a backslash (\);
- Numbers - same representation has the Strings.

A YAML document can optional begin with three hyphens (`---`) and also optional end with three dots (`...`)

Using the example 2.2, the YAML representation can be something like this:

Listing 2.5: YAML Representation

```
users:
- id: 974e7b5c-fd64-4bfe-92a5-ae60ad16d4bd
  first_name: Rodrigo
  last_name: Santos
  age: 30
- id: 94135e0a-0778-4cb2-8982-80e3e7bdea21
  first_name: John
  last_name: Doe
  age: 100
```

Comparing with the same JSON representation 2.4 this is far more simple. But in the other and the YAML representation can have some problems, the use of whitespaces as a mean of indentation can lead to unexpected results.

2.1.4 Conclusions

Being that in this section we studied different data representations. Each language was created and can be better suited for some specific cases other than others, this does not mean that one is better than other.

	Data Representation	**Objects Data type**	**Parse**	**Verbose**
XML	Human and Machine Readable (text)	All the values are Strings	Difficult	High
JSON	Human and Machine Readable (text)	The values can be: Strings; Numbers; Booleans; Null; Arrays; Objects	Easy	Moderate
YAML	Human and Machine Readable (text)	Mix between json and xml	Easy	Moderate/Low

Table 2.1: Data Representation Comparison

2.2 Schemas for data representation

Generally speaking, a schema is a meta-representation of an object. It defines the characteristics related to each object property (name, type and other data constraints).

When developing a service, it is essential to get some security level about the input data quality, and thus the service should validate the input data before being accepted. We can use Schemas to validate documents.

Alternatively, the code could perform input validation programmatically, increasing code complexity and making the code more difficult to maintain/read. Using schemas, we can take advantage of external libraries that accept the input and validate if it follows all the imposed structural rules and data constraints. It is also easier to update a schema than refactor the code to change some validation parameters.

We will cover some of the most used schemas, like XML Schema Definition (XSD) and JavaScript Object Notation (JSON) Schema. We choose those schemas because of popularity and compatibility with the languages in section 2.1.

2.2.1 XML Schema Definition (XSD)

The XSD is the World Wide Web Consortium (W3C) recommended schema to describe the elements that define a Extensible Markup Language (XML) document, it can be used to define a set of constraints that a XML document must follow. These constraints can be related to the document structure, entities definition, data types and default values for entities and attributes.

Since the XSD defines constraints for expected XML documents, we can use it to validate the input document. This validation compares the document structure to the expected structure defined in the schema, and it also verifies if all data presented in the entities follows the correct constraints.

As the XML is one of the most preferred data representation standards, many tools that can process those documents already exist. They can transform its contents into human-readable documentation, allowing for an easier understanding of complex XML documents [18].

The XML specification defines the concept of `Namespaces` [7]. In XSD the `Namespaces` is a mechanism for defining a schema across multiple files (child schemas). This can be seen as a way to simplify the schema and avoid searching the whole schema in case of changes (only need to change the child schema). When using namespaces the value `xs` must always define the schema `xmlns:xs="http://www.w3.org/2001/XMLSchema"` [3].

These documents follow the same structure and syntax as a XML document, having the same properties and constraints.

In the example 1 (Appendix A), is shown one possible XSD representation of the XML document defined in the example 2.3.

All XSD documents start with the `<schema>` tag. This tag may contain some attributes that help defining the schema:

- `xmlns:xs` - defines that the elements and data types used in the schema come from "http://www.w3.org/2001/XMLSchema" namespace;

- `targetNamespace` - defines that the elements present in this schema (Users, user, firstName, lastName, age) come from the "https://api.example.com/api/contact-center/Users.xsd" namespace;
- `elementFormDefault` - defines if the elements declared in the document must or not be qualified (if it belongs to the namespace);
- `attributeFormDefault` - same as `elementFormDefault` but for the attributes.

The `<element>` tag defines each of the elements. We can use the attributes to determine some object properties, like name, type, occurrence constraints, default values and fixed value.

Nested values can be defined using the `<complexType>` . An element can also have some content restrictions, using the `<restriction>` tag. Restrictions can be an interval of values, enumerations of valid values and patterns and size for strings.

Using a XSD schema to represent a XML document has multiple advantages over Document Type Definition (DTD) or Simple Object XML (SOX) - two cases of early adopted schema languages for XML. A XSD schema follows the same format of a XML document, which means that the same tools can be used to parse both documents, the language is self-describing and also self-documenting.

2.2.2 JavaScript Object Notation (JSON) Schema

JSON schema [8] is a language that can be used to define the structure, the contents and data properties of JSON and yaml documents [1]. Using JSON schema we can evaluate if a JSON object contains all the required values, if the values have the correct data type and also verify if they respect the specified constraints, allowed values, size, patterns, etc.

Like XSD schemas are XML documents, a JSON schema is also a JSON document, and as such follow the same language and set of rules.

A JSON schema document can is a set of hierarchical schemas. Many of the values associated with specific keys, make use of schemas to define its constraints.

The example 2 (Appendix B), is the schema representation of the JSON document defined in 2.4, in this example, as in XSD, the defined schema can be used to verify all the input information when it arrives to the server.

The JSON schema has the following properties:

- '$schema' - defines that the document is a schema and it also points to the meta-schema that contains the valid vocabulary used to define this schema — in this case, it is the draft seven specifications;
- '$id' - unique identifier for each schema in the document (mandatory only for the root node);
- title - allows for the definition of a title for the schema (optional);
- description - a small description for the schema;

[1] https://json-schema-everywhere.github.io/yaml

- type - the data type that the property must follow, this data types must be a valid type according to the JSON specification, and this defines the first constraint;
- properties - defines an object that contains keys representing the property name and a JSON schema for each value. This schema will then validate the contents of that property;
- additionalProperties - is used to handle if properties that are not defined are allowed or not, by default all extra properties are allowed;
- items - similar to the properties, the items define the set of possible schemas for the elements that are present in a list. This property is only valid when defining constraints to Arrays/List elements;
- required - defines which of the possible properties must be present in the JSON document;
- dependencies - creates a directional dependency between one property and a set of other properties, meaning if the first property appear all of the properties that it depends must also appear, but the contrary is not true.

Having the objective of being human-readable, the JSON schema language has a lot more element complexity to describe the documents. This can lead to larger documents compared to the ones produced in the XSD language 1 [20]. This problem occurs because of how each language is structured. The XSD being a XML document, has access to a lot more elements that can it can use to describe certain aspects of an object. On the other hand the JSON language is simpler and needs to use nested properties to define some complex objects [20] [1].

2.2.3 Conclusions

Being that XSD follow the same structure of the XML documents, it can define and constrain all the elements of the document. With XSD, it is possible to add more meaning to each structural and logical part of the document [23].

The JSON Schema can generate clear, human- and machine-readable documentation. It also allows for an accurate description of the data structure. On the other side, the JSON Schema complexity when defining the property constraints can lead to a steep learning curve[1], there is also the document size problem because every nested level of JSON adds two levels of JSON Schema[1].

2.3 Architectural communication patterns for Web Services

Architectural Communication Patterns are implementations, standards and guidelines used to develop web services.

In this section, we focused the research on patterns that work with the data representations researched in subsection 2.1.

We will cover some of the most used patterns, like Remote Procedure Call (RPC), Simple Object Access Protocol (SOAP) and Representational State Transfer (REST). We will also talk about some advantages of using hypermedia.

2.3.1 Remote Procedure Call (RPC)

In Distributed systems, a RPC system is when a program can invoke resources from a remote machine. An RPC is like a program invoking functions outside of the machine. In RPC there are two distinct actors, the client and the server. The client sends a request to a remote machine, in some place over the network. This request contains all the necessary parameters to allow for the execution. The server will receive the request information. It will process the input data and then return the response (result) to the client. During this process, the client will typically be waiting for the result, making this a form of synchronous communication.

Most RPC calls are synchronous. However, using some programming techniques like using processes and/or threads to parallelize remote calls, leaving the threads waiting for the response, can mitigate this problem. This solution can lead to some performance problems as machine resources are limited, and the number of available threads is also limited. There are some RPC implementations, like the XMLHttpRequest (later called Simple Object Access Protocol (SOAP)), that allow for asynchronous requests. This kind of modifications leads to multiple protocol implementations that are not compatible with each other.

One problem that might occur during a RPC call is that the communication might not be entirely reliable, leading to some remote calls failing because of network problems. When this problem occurs, it is tough for clients to know if the server could perform the remote actions were, and in most cases, there are no mechanisms to retry the operation.

2.3.2 Simple Object Access Protocol (SOAP)

In distributed Systems SOAP is a communication protocol used to transfer structured messages. In SOAP the messages are structures using the Extensible Markup Language (XML) information set, this is a World Wide Web Consortium (W3C) specification[2] that can be used to describe an abstract data model of an XML document.

The SOAP messages have some syntax constraints:

- must be encoded using XML;
- must use the SOAP Envelope namespace
- must not contain a Document Type Definition (DTD) reference

[2] `https://www.w3.org/TR/2004/REC-xml-infoset-20040204/`

- must not contain XML processing instructions

The SOAP specification, does not define how the client and server must exchange the messages, the SOAP Bindings solves this issue [11]. The bindings are a set of mechanisms that ensure the correct exchange of SOAP messages using a transport protocol. Most of the SOAP implementations have bindings that ensure compatibility with most of the more common transport protocols, like Hypertext Transfer Protocol (HTTP) or Simple Mail Transfer Protocol (SMTP). Most of the SOAP implementations rely on the use of HTTP because it is widely used on web service development [17].

Like Remote Procedure Call (RPC), SOAP allows for a program to invoke processes that can be running on different machines. The use of HTTP for communication and XML for data representation allows for service abstraction, meaning that the invoked services language is agnostic and can also be running on multiple platforms.

The use of XML as a message format can lead to a lot of message complexity, increasing the transmission size of each message has seen in the section 2.1.

2.3.3 Representational State Transfer (REST)

The REST is a software architectural style that defines a set of rules and constraints. When a service is compliant with the REST standard, it is called a RESTful Web Service.

According to [10], the constraints of the REST architectural style affect the following architectural properties:

- **performance** in component interactions, which can be the dominant factor in user-perceived performance and network efficiency;
- **scalability** allowing the support of large numbers of components and interactions among them.
- **simplicity** of a uniform interface;
- **modifiability** of components to meet changing needs (even while the application is running);
- **visibility** of communication between components by service agents;
- **portability** of components by moving program code with the data;
- **reliability** in the resistance to failure at the system level in the presence of failures within components, connectors, or data.

According to [10], there are six guidelines or constraints that can define a system as a RESTful system. This set of guidelines define how service must process information and respond to client requests. The main goal of using these guidelines is to give the system a set of non-functional properties, such as the ones defined in the list defined above.

The formal REST constraints are as follows:

Client-Server architecture

Using a client-server architecture allows for a great separation of contexts. For example, suppose the interface layer and data layer are separated and independent (via communication interface). In that case, they can quickly be developed each layer in multiple platforms/systems without interfering with each other. This context separation also allows for a more scalable system [14], because we have less complicated modules, and it is easy to create and deploy new modules. The most crucial gain of a Client-Server architecture is the ability to iterate different components separately, meaning that we can evolve a piece without changing others.

Stateless Servers

In a stateless system, the server must not save any context about client requests, and it only contains the logic necessary to understand and reply to each request. Because of the stateless server constraint, all the clients must save all the context information, and all the requests must contain all the context information. According to to[14] this constraint also allows for simple servers, because there is no need to keep information about resources between requests. It also improves the system scalability, because the resource usage is minimum, and the server can free the used resources when it finishes serving the request. In [10] is said that using the stateless approach can generate some communication overhead. For each request, the client must send the required information necessary to perform that request and all the prior information that might be relevant.

Cache

Any response must either cacheable or non-cacheable. Cached responses can be saved (for some time, by intermediary parties) and later served to equivalent requests. By using cache mechanisms, we can reduce the number of interactions between clients and server. Serving cached responses allow for a boost in performance.

Uniform interface

Using uniform interfaces, we can decouple the interfaces from the implementations. This constraint is composed of the following sub constraints, resource identification in requests, resource manipulation through representations, self-descriptive messages and Hypermedia as the engine of application state (HATEOAS).

Resource identification in requests

In a RESTful service, each resource is identifiable using an immutable Uniform Resource Identifier (URI) if the resource is updated the URI must be the same.

Resource manipulation through representations

The client does not access the resources directly; instead, the service exposes an URI and a resource representation that contains the resource data.

Self-descriptive messages

Each message must include enough information to describe how to be processed. For example the message can define which parser to use by defining a MediaType (`application/json` or `application/xml`).

HATEOAS

The system must work in self-discovery mode, having access to an initial URI (entrypoint). The client should then be able to use provided links in the responses dynamically discover all the available actions and available resources. The server responds with text that includes

hyperlinks to other actions that are currently available. The client can be almost entirely generic. It only needs to know how to call the entrypoint.

Layered system

A client must only know the service layer that he is calling. All the other service layers must remain hidden. The client must be entirely agnostic for whom he is communicating. It must communicate with the server or with a proxy/load balancer, without the need for code changes.

By enabling load balancing and providing shared caches, we can improve system scalability. It is also possible to add security as a layer on top of the services and separate the business logic from security logic.

Finally, it also means that a server can call other servers/services to perform actions needed to satisfy the client request, without the client knowing.

Code on demand

This constraint is optional. It states that a service must allow a server to manipulate the client in runtime. The server can send code to the client to be executed, extending or customising the client functionalities.

2.3.4 Conclusions

The RPC allows for simple web services. The client can invoke the server and use the available resources to perform complex tasks; it is easy to deploy and can work over multiple transport protocols. But it also lacks support, out of the box, for some features commonly seen in web services implementations, like asynchronous calls and error handling.

The SOAP is like a successor of the RPC, supporting some of the missing features. The SOAP can also use multiple transport protocols, but it is commonly used in conjunction with HTTP. The messages use the XML structure, leading to some communication overhead.

Finally, the REST is a set of properties and constraints (guidelines) that a service must follow to achieve the following set of non-functional properties: performance, scalability, simplicity, modifiability, visibility, portability, reliability. Being only an Architectural style, this means that the REST implementations can be language and technology agnostic.

2.4 RESTful Implementations

In this section, we will cover some message implementations that we might use to create Resource Oriented Architecture (ROA) that are compliant with the Representational State Transfer (REST) proposed constraints.

From the multiple defined REST constraints, this message formats are used to make the REST Application Programming Interface (API)'s follow the Uniform interface constraint.

We will cover some of the most used implementations, like Hypertext Application Language (HAL), JavaScript Object Notation for Linked Data (JSON-LD), Hypermedia-Driven Web APIs (HYDRA), Collection+JSON and Siren.

2.4.1 Hypertext Application Language (HAL)

The HAL representation [12] gives a generic media type to specify an Application Programming Interface (API) as a series of links. The application uses a set of relations to define possible actions and associated links. Clients can then used those links to get the information necessary to flow through the application.

In the first draft [13], the HAL representation only followed the JavaScript Object Notation (JSON) language, but on a later one [19] it was expanded to also be compatible with the Extensible Markup Language (XML).

Using the object model 2.2 as an example, we can define the general user resource as such:

Listing 2.6: JSON HAL Representation

```
{
  "_links":{
    "self":{
      "href":"api/contact-center/users"
    }
  },
  "id":"users",
  "name":"Contact Center Users"
}
```

Listing 2.7: XML HAL Representation

```
 <resource rel="self" href="/api/contact-center/users">
  <id>users</id>
  <name>Contact Center Users</title>
</resource>
```

The listings 2.6 and 2.7 represent the same resource, using the two possible representations, JSON and XML.

An API response can have child resources, this resources can have a direct relation to the resources served in the current service state, meaning that expanding the response and serving the related resources can prevent future API calls. There is also the possibility that the response can have a collection of resources.

In the JSON Specification of HAL [13], the keyword `_embedded` is reserved. This keyword is optional and can be used to represent the related child resources. This resources can

be a single object or a list of objects, each one following the HAL structure, with `_links` property, the object definition and optionally an `_embedded` property for nested resources.

In the listing 3 (Appendix C), the `_embedded` property has a single object of the type user with a `_links` property that is addressed for its own Uniform Resource Identifier (URI). In the listing 4 (Appendix C), the base `_links` property has multiple values, each one of them represent a search option that the system allows for page navigation. In this example, the `_embedded` property now has a list of objects of the type "user", and each one of them has its `_links` property.

In the case of the XML representation, the specification [19] defines that the tag <resource> is used to define the embedded resources. In the case of a response that can have multiple types of resource, the attribute `rel` can be used to specify the resource type.

The example 5 like the 3 (both available in Appendix C), the response has a single embedded resource. XML representation, the self link can be represented directly in the resource attribute.

In the example 6 (Appendix C), the response has a collection of users, each one of them has a corresponding `<resource>` tag with the corresponding "self" link attribute. The root resource has a set of child `<link>` tags that represent the operations available in the page navigation. Using both JSON and XML representations, allows for representation of the same information in the responses.

The XML representation is more straightforward when dealing with self links and embedded results. Because the self links appear in the resource attributes and the child embedded resources appear all at the same level, using the `rel` attribute to make the differentiation can lead to some problems with resource organisation.

In the case of the JSON representation, there is the need to use more attributes to represent the same information. Because the data is grouped to prioritise the document's readability, this is more evident in the `\_embedded` resources, where all the elements appear inside a tag that defines the relationship.

2.4.2 JavaScript Object Notation for Linked Data (JSON-LD)

The JSON-LD is an World Wide Web Consortium (W3C) endorsed media type, this format allows for data encoding using the notation of linked data. It is a set of structured data related to other data. This link is useful when retrieving some information because there is a direct link to query the system to retrieve some possible useful extra information [16].

The specification of the JSON-LD allows for an existing API to be upgraded to use the new notation without significant modifications to the payloads [24]. This syntax was designed not to introduce breaking changes to already deployed systems, the responses will maintain the same structure (and fields) but will include some extra semantics.

Listing 2.8: Basic User JSON representation

```
{
    "id" : "974e7b5c-fd64-4bfe-92a5-ae60ad16d4bd",
    "first_name" : "Rodrigo",
    "last_name" : "Santos",
    "age" : 30
}
```

The example 2.8 can be seen as an simple JSON document that can be used to represent the information of a system user. This kind of representation can lead to some ambiguities when using simple tokens (JSON keys) to identify the represented data. In the JSON-LD, to avoid ambiguities, the JSON keys can be replaced by Internationalised Resource Identifiers (IRI)s [9].

Listing 2.9: JSON-LD document JSON IRI representation

```
{
    "http://schema.org/uuid" : {
        "@id" : "974e7b5c-fd64-4bfe-92a5-ae60ad16d4bd"
    },
    "http://schema.org/first_name" : "Rodrigo",
    "http://schema.org/last_name" : "Santos",
    "http://schema.org/age" : 30
}
```

In the example 2.9, the IRIs define a simple and unambiguous identifier to each one of the document properties. The specification stated that the keys must be the Uniform Resource Locator (URL) for the schema that defines each property. The use of IRIs can expand the property context when parsing the JSON document, using the IRIs it is possible to retrieve the object definition (schema).

In the JSON-LD documents when there is a value that define an IRI, the keyword `@id` is used for its definition [24], in this example this keyword is used when defining the user id.

This representation can be overly verbose when presenting the information, making the readability of the documents more difficult and changing all the keys to the IRIs values can lead to API incompatibilities. To avoid those problems, the JSON-LD specification defines the use of the `@context` keyword [24]. This keyword defines a map between the object keys and the corresponding IRI.

Listing 2.10: JSON-LD document with Inline context representation

```
GET: https://api.example.com/api/contact-center/users/974e7b5c-fd64-4bfe-92a5-ae6
    ↪ 0ad16d4bd
{
    "@context" : {
        "user_id" : {
            "@id" : "http://schema.org/uuid",
            "@type" : "@id"
        },
        "first_name" : "http://schema.org/first_name",
        "last_name" : "http://schema.org/last_name",
        "age" : "http://schema.org/age"
    },
    "@id": "https://api.example.com/api/contact-center/users/974e7b5c-fd64-4bfe
        ↪ -92a5-ae60ad16d4bd",
    "user_id" : "974e7b5c-fd64-4bfe-92a5-ae60ad16d4bd",
    "first_name" : "Rodrigo",
    "last_name" : "Santos",
    "age" : 30
}
```

In the example 2.10, the `@context` object is used to alias the JSON keys to the specific IRI identifier, in this example the context appears inline with the rest of the data

representation, but the JSON-LD specification allows for multiple context definition:

- Inline - the context appears inline with the properties that it defines;
- Reference - The context is a external resource that is referenced using an URL `("@context": "https://json-ld.org/contexts/person.jsonld")` ;
- Relative context - Defines the relative path to the context definition, this path is relative to the document location `("@context": "context.jsonld")` .

The use of the context object to map the IRIs to the corresponding JSON keys, allows for the compatibility of APIs. If an API is not prepared to understand the context property, the serialiser can ignore this property and parse the rest of the object as he usually would.

The JSON-LD does not support specifying the actions that can be done against a resource. To address this problem, Hypermedia-Driven Web APIs (HYDRA) provides a vocabulary expansion that allows for communications using the JSON-LD syntax.

2.4.3 Hypermedia-Driven Web APIs (HYDRA)

The HYDRA specification [15] appears as a way to simplify the creation of hypermedia-driven Web APIs. This specification is built over two core components, the JSON-LD format [24] and the HYDRA core vocabulary [15]. The JSON-LD defines the communication contract between the server and client and the HYDRA vocabulary is used to define the common elements between JSON-LD and HYDRA.

Using HYDRA it is possible to implement web services that follow the Representational State Transfer (REST) guidelines, it also allows for the creation of generic API clients instead of tailored clients for each API.

Using the HYDRA vocabulary [15], like the example 7 (Appendix D), the keyword `operations` can be used to specify the set of available steps that the client can perform over the resource identified with the `@id` property. Inside of the `operations` property it is possible to define:

- `@type` - The action type;
- `method` - The Hypertext Transfer Protocol (HTTP) method that is supported by the endpoint;
- `expects` - This property an expected template that defines the properties that are expected to appear in the request and data constraints for each one of them.

The HYDRA vocabulary [15] also defines a `member` keyword, that can be used to define embedded resources. The response also has the `nextPage` property, used to define the page used when searching for collections.

2.4.4 Collection+JSON

The Collection+JSON is a hypermedia format with the principal focus on operations over collections, like create, read, update and delete (CRUD). It is also possible to perform

search queries using query templates and write operations using specific server-side templates [5]. As the name implies, using Collection+JSON is centred in collections of multiple objects, but the specification also has the notion of single element collection.

Listing 2.11: Basic Collection+JSON document

```
GET : https://api.example.com/api/contact-center/users/88dfde3b-865c-4613-8c1d-
    ↪ df5d891055ec
{
    "collection": {
        "version": "1.0",
        "href": "https://api.example.com/api/contact-center/users/88dfde3b-865c
            ↪ -4613-8c1d-df5d891055ec"
    }
}
```

The example 2.11 represents the basic Collection+JSON response [4], it has a `collection` object that contains the `version` and the `href` properties, the version points to the specification version, in this case it is `1.0` , and the href is the URI used to identify the returned resource.

The example 9 (Appendix E), represents the expected response in case of a search request, like the basic example, the response has the `collection` object with `version` and `href` , in this case it also includes the `links` and `items` properties, the links define the page navigation options available and the items define the found objects that match the search request. For each element found there is an `href` that contains the object URI and a data property, the data contains an array of name/value pairs that define each property of the returned object.

This response also has a `template` property, this template can define the elements inside of a collection, the information present in this template gives the necessary fields to add (POST) a new resource or to update (PUT) the fields of a previously created one. To create a new element, the client must send a POST request to the `href` defined in the `collection` property ("https://api.example.com/api/contact-center/users"), in the case of a PUT request, the URI appears inside of the correct element of the `items` property.

Lastly, there is the `queries` property. This property defines the search queries supported by the collection, in the `data` object it is possible to describe all the supported query parameters.

The use of templates and defining the supported queries allow for a greater understanding of an API by new users because most of the API documentation and possible operations appear in the responses.

2.4.5 SIREN: a hypermedia specification for representing entities

Siren is a specification [25] built for entity representation, this specification also allows for the definition of available actions that a resource supports. The action definition is used for client navigation, making it possible to create decoupled clients who can use the actions information to navigate the service.

In Siren there is the concept of:

1. `Entities` - This property is used to represent the entity (Class) information:

- `class` (optional) - Array of String that can be used to identify the object type and describe the object structure and content;
- `properties` (optional) - A set of key-value pairs that are used to convey the data for the entity;
- `entities` (optional) - An array of sub-entities that are related with the parent entity, the sub-entities must always have a 'rel' property to show the relation between them and the parent entity;
- `links` (optional) - a set of items that define the URIs for each possible navigation link, in the entities the links property must always have one entry, a link to the entity self location;
- `actions` (optional) - A collection of available operations that can be performed on an entity;
- `title` (optional) - this field can be used to provide some description about the entity.

2. `Sub-Entities` - An array of sub-entities that are related with the parent entity, this array can contain an embedded entity representation or a embedded link to the entity, in this case it can contain:

   - `class` (optional) - Array of String that can be used to identify the object type and describe the object structure and content;
   - `rel` (required) - Defines the relation of the sub-enteity and its parent;
   - `href` (required) - This property defines the URI of the sub-entity;
   - `type` (optional) - The media-type of the sub-entity;
   - `title` (optional) - this field can be used to provide some description about the entity.

3. `Actions` - A collection of available operations that can be performed on an entity, they can have this properties:

   - `name` (required) - Identifier of the action, must be unique for the set of actions that belong to an entity;
   - `class` (optional) - Array of String that can be used to identify the action representation
   - `method` (optional) - defines the protocol method to be invoked, if omitted it defaults to 'GET';
   - `href` (required) - This property defines the action URI;
   - `title` (optional) - Description of the action
   - `type` (optional) - The request encoding type, by default it uses 'application/x-www-form-urlencoded'
   - `fields` (optional) -

4. `Links` - are used to define the steps available during page navigation, they can have this properties:

   - `rel` (required) - Array of String that defines the relations of a link to a entity;
   - `class` (optional) - Array of String that can be used to identify the link representation
   - `href` (required) - This property defines the URI of the linked resource;

- `title` (optional) - Text that describes the link;
- `type` (optional) - The media-type of the linked resource.

5. `Fields` : Are a set of fill rules, that exist inside of an action, they can have this properties:
   - `name` (required) - Identifier of the field, must be unique for the set of fields that belong to an action;
   - `type` (optional) - The data-type of the defined field;
   - `value` (optional) - The value that is assigned to the field;
   - `title` (optional) - Description of the field.

The example 10 (Appendix F) represents an SIREN JSON document that describes the request to search for all users of the contact-center.

With Siren, it is possible to design resources that do not have to be primarily CRUD-based. In response to the available actions that each resource supports, it is easy to provide a task-based interface through the Web API [3].

2.4.6 Conclusions

JSON-LD can improve existing APIs without introducing breaking changes. Most of the improvement gains are ways to self-document the API. HYDRA adds the necessary vocabulary to extend the JSON-LD specification to allow the communication.

HAL has a minimal document representation that has most of the benefits of using hypermedia without too much complexity to the implementation. One area where HAL struggles is, like JSON-LD, the lack of support for specifying actions.

The collection+JSON, despite the name, can be used to represent collections of items or a single item. This implementation can list specific queries that the collection supports and allow for templates that clients can use to alter (add, update, delete) the collection elements.

The SIREN implementation can represent generic classes of items and overcome the main problem of HAL – lack of support for actions. It also introduces the concept of classes to the data representations allowing for the API response to represent the object type.

We decided to use the JSON as the resource representation because it already represents the developers' representation while developing the services. As a consequence, the resources are described using JSON schemas.

The service API will follow the REST design pattern. HAL was chosen as the RESTful pattern for the responses because it is defined in Talkdesk's guidelines for external API.

[3] https://github.com/kevinswiber/siren

This page is intentionally left blank.

Chapter 3

System requirements and Architecture

As previously stated, this internship aims to develop a framework architecture and communication contracts to create a Representational State Transfer (REST) compliant services.

This chapter contains three sections: requirement analysis; description of the system architecture; initial risk analysis.

3.1 Requirements

The requirements specification is a crucial phase in the development of software. They provide goals to be met and also serve as a metric to evaluate the project progress. In this section, we specify the requirements for the project.

3.1.1 Functional requirements

In this section, we present a list of functional requirements. The requirements are composed of ID, name, description, priorities, and dependencies (requirements that it depends on). The priorities are defined in: Must Have, Should Have, and Nice to Have.

ID: FR-01 **Name**: Data Representation **Description**: The system must use rich enough language to allow the definition of each object representation; namely properties, data-types and value constraints – defined as the object schema **Priority**: Must Have **Dependencies**: None

ID: FR-02 **Name**: Schema **Description**: The system must be able to serve the object schemas for any object supported by the system **Priority**: Must Have **Dependencies**: None

ID: FR-03
Name: Resource Creation
Description: The system must be allow for the creation of new meta-resources (new schemas), without changes to the system code
Priority: Must Have
Dependencies: None

ID: FR-04
Name: Object persistence
Description: The system must be able to persist objects regardless of the object schema
Priority: Must Have
Dependencies: None

ID: FR-05
Name: Object Operations
Description: The system must allow creating, searching, updating and deleting specific objects regardless of its object schema
Priority: Must Have
Dependencies: FR-04

ID: FR-06
Name: Object Validation
Description: The system must be able to validate all requests to create and update objects using the appropriate object schema
Priority: Must Have
Dependencies: FR-05

ID: FR-07
Name: Object Rejection
Description: The system should not allow for the creation of objects, that do not have a object schema associated
Priority: Should Have
Dependencies: FR-05

ID: FR-08
Name: Response Operation Definition
Description: The system should use hyperlinks in each response to define the next possible operations
Priority: Should Have
Dependencies: None

ID: FR-09
Name: Response Schema Definition
Description: The system should use hyperlinks in each response to define the associated object schemas
Priority: Should Have
Dependencies: None

ID: FR-10
Name: Service Entrypoint
Description: The system should provide a default operation which can be used to start the chain of service discovery
Priority: Should Have
Dependencies: None

3.1.2 Non-Functional requirements

Non-functional requirements are requirements that define the behaviour of the system, as opposed to functional requirements that define the components that the system must have. The existing non-functional requirements are has follows:

ID: NFR-01
Name: Modifiability
Description: It must be possible to modify or introduce new functionalities in the service modules without affecting the existing features.

ID: NFR-02
Name: Reusability
Description: The implemented modules should work together, but if necessary, they must also be deployed independently in other systems.

3.1.3 Business constraints

Business constraints depend on business decisions and cannot be altered. The following business constraints exist:

ID: BC-01
Name: Licenses (Commercial Purposes)
Description: The licenses of the tools and libraries used must allow for commercial use.

ID: BC-02
Name: Licenses (Free)
Description: The tools and libraries used must have free licenses.

ID: BC-03
Name: Development Time
Description: The first version of the implementation has to be preformed during the internship second semester.

3.1.4 Technical constraints

Like the business constraints, the technical constraints must not be changed. This constraints are usually set by the stakeholders and have impact on a technical level. The following technical constraints exist:

ID: TC-01
Name: On-premises
Description: The software and all associated tools must be able to be deployed on-premises.

ID: TC-02
Name: Programming Languages
Description: The software and associated tools must be developed in the same language the rest of the development team is using. Java was chosen as the development language.

ID: TC-03
Name: Modular System
Description: The system must be modular allowing to a seamless change of components. Required that all components use the same implementation.

3.1.5 Use cases

A use case is a methodology used in system analysis to identify and organise system requirements. Allowing for the specification of a set of high-level requirements helps in the more detailed identification of the various functional requirements.

In this section, we will show the five use cases defined for this project. The following tables describe them.

ID	**UC01**
Name	List resources
Description	User interacts with the system, in order to list available resources
Scope	System
Actor(s)	USER
Preconditions	The user selects the option to list resources
Basic Flow	The user selects the option to list resources; The system searches for available resources; The system displays the result to the user
Exception Flow	
Post Conditions	The system displays the result to user

Table 3.1: Use Case 1

ID	**UC02**
Name	Show the details of a resource
Description	User interacts with the system, in order to get all the details about an available resource.
Scope	System
Actor(s)	USER
Preconditions	The user selects the option to get a resource; The chosen resource must exist
Basic Flow	The user selects the option to list resources; The system searches for available resources; The system displays the result to the user; The user Selects one of the resources; The system searches for details about the resource; The system displays the result to the user.
Exception Flow	If the resource is not available the system displays an error message;
Post Conditions	The system displays the result to user

Table 3.2: Use Case 2

ID	**UC03**
Name	Resource Creation
Description	User interacts with the system, in order to create a new resource.
Scope	System
Actor(s)	USER
Preconditions	The user selects the option to create a new resource; The system must have at least one resource template available
Basic Flow	The user selects the option to create a new resource; The system searches for available resource types; The user chooses the type; The user fills the proper information; The system displays the result to the user.
Exception Flow	If there is no templates available the system displays an error message;
Post Conditions	The system displays the result to user

Table 3.3: Use Case 3

ID	**UC04**
Name	Resource Update
Description	User interacts with the system, in order to update a resource.
Scope	System
Actor(s)	USER
Preconditions	The user selects the option to update an old resource; The chosen resource must exist; The chosen resource template must exist;
Basic Flow	The user selects the option to create update a resource; The system searches for available resources; The user chooses the resource; The user fills the proper information; The system displays the result to the user.
Exception Flow	If the resource does not exist the system displays an error message;
Post Conditions	The system displays the result to user

Table 3.4: Use Case 4

ID	**UC05**
Name	Delete a Resource
Description	User interacts with the system, in order to delete an available resource.
Scope	System
Actor(s)	USER
Preconditions	The user selects the option to get a resource; The system must have at least one resource available
Basic Flow	The user selects the option to list Resources; The system searches for available resources; The system displays the result to the user; The user selects one of the resources; The system deletes the resource; The system displays the result to the user.
Exception Flow	If the resource does not exist the system displays an error message;
Post Conditions	The system displays the result to user

Table 3.5: Use Case 5

3.2 Architecture

In this section, we present a high-level service architecture. The principal constraint that the service must be modular.

Figure 3.1 represents the generic framework architecture we want to achieve. The image is also present in the Appendix G, in larger size for ease of read.

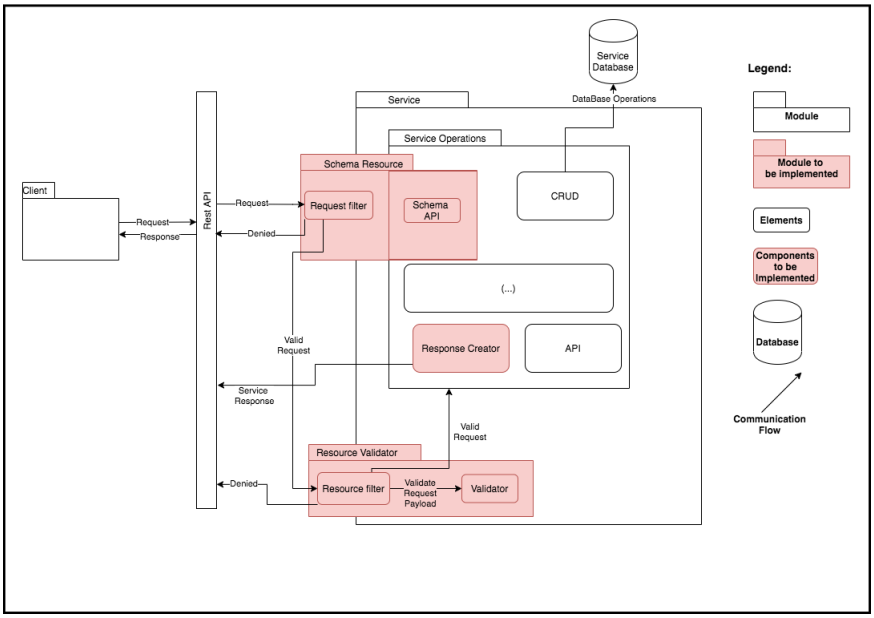


Figure 3.1: Framework Architecture

The red components/modules are the ones that will be implemented, all the others are outside of the scope of this internship.

First, we have the "Schema Resource" module, composed of two components: "Request filter" and "Schema API". The first component will act as an API filter; it will intercept all the request done to the service API and will ensure that the clients and the service speak the same language. The second component is a pluggable API and service logic to serve schemas to the client and internal modules/components.

The framework will also need to validate the input information (against the defined schema). To do that, we will have "Resource Validator". This module also has API filter, this filter ("Resource filter") takes the payload and schema identifier from the request. The validation part is done by the second component, the "validator", it will accept a document (input data) and a schema. The input will then be challenged against the schema, and it will produce a result saying if the request is valid (passes all the checks) or invalid (failed at least one check).

In the "Service Operations" module are all the supported service operations. These operations can differ from service to service. Here we have just some of the possible functions.

Lastly, we have the "Response Creator" component. This component belongs to the service operations and will receive a response and translate them to a response compliant with the RESTfull implementation.

3.3 Risk Analysis

Here we present an initial risk analysis with potential problems for the project. This section explains how risks are classified and prioritised. In the final part of the section a list of risks identified is presented. All the risks identified are classified based on their impact and their probability in order to better determine which risks' management should be prioritised.

The risk's impact can assume the following values: **Catastrophic**, **Critical**, **Marginal** and **Negligible**.

Probability can be classified as:

- Very Likely: Probability of event happening is estimated as being above 75%;
- Likely: Probability of event happening is estimated as being between 50% and 75%;
- Unlikely: Probability of event happening is estimated as being between 25% and 50%;
- Rare: Probability of event happening is estimated as being below 25%.

Impact / Probability	Negligible	Marginal	Critical	Catastrophic
Very Likely				
Likely				
Unlikely				
Rare				

Table 3.6: Risk Exposure Matrix

In the table 3.6 we have the Risk Exposure Matrix, this matrix can be used to map risks to specific zones. There are four possible zones that define the impact of a risk in the development of the project. Each zone have an specific impact:

- Green Zone: Risks with low priority, they will only addressed if the mitigation plan will demand low effort. Risk that appear in this zone have a combination of low impact and low probability;
- Yellow Zone: Risks with moderate priority, a plan to mitigate this risks might exist, they will only be executed if there are spare time.
- Orange Zone: Risks with high priority, a plan to mitigate this risks must exist, they will only be executed if they do not clash with mitigation plans of higher priority risks.
- Red Zone: Risks highest priority, there must always exist a plan and its execution to mitigate them. Risks that fall in this zone have a combination of high probability and high impact.

3.3.1 Risk List

In this section we identify some of the possible risks and have them classified according to the risk exposure matrix 3.6. The mitigation plan, if any exist, is also described.

Name: Lack of experience on Restfull implementations
Description: This project will rely on set of preexisting tools and concepts. There performance can drastically influence the system end result.
Probability: likely
Impact: Critical
Priority: Orange zone
Mitigation Plan: Study APIs that have Restfull implementations. Practice by doing some tutorials.

Name: Development tools
Description: This project will rely on set of preexisting tools and concepts. There performance can drastically influence the system end result.
Probability: likely
Impact: Catastrophic
Priority: Red zone
Mitigation Plan: It will be necessary to have a testing phase, were the tools will be tested and see their performance. Only then the we can choose the tolls that will be used.

Name: Limited Development time
Description: Due to limited development time, some requirements may not be (fully) implemented.
Probability: Unlikely
Impact: Catastrophic
Priority: Orange zone
Mitigation Plan: In order to have a working system, the development will be focused on the functional requirements with higher priority.

Chapter 4

Framework definition

The work done implementing the framework is shown in this chapter. We decided that, for this first attempt, the framework will only support the Hypertext Application Language (HAL) pattern for the responses because it is the one specified by Talkdesk's guidelines for response patterns for externally exposed Application Programming Interface (API)s.

First, we have a simple overview of the tools used in the development of the framework, and then we have a more comprehensive analysis of each component and their interactions.

4.1 Contract rules

This section defines the contract rules that service and client implementations need to respect. They are independent of the implementation but serve as guidelines to develop all framework modules.

The implemented framework can be replicated in other programming languages.

4.1.1 Communication

We use the Representational State Transfer (REST) communication pattern to enable communication between the various services and respective clients. Being the REST a protocol-gnostic implementation, we decided to use the Hypertext Transfer Protocol (HTTP) to abstract all necessary communication steps.

The will implement a custom Media-Type header to guarantee that the client and the service speak the same language. If the client doesn't send the correct header information, the service must discard the request.

4.1.2 Data representation

Our system will use the JavaScript Object Notation (JSON) format as the data representation format because it is human-readable, has low verbosity and easy to parse. Using JSON to represent the resource will give us some information about it, like the field names, and the value type.

The JSON-Schema is used as a high-level resource representation giving a way to define

complex constraints. All clients must be able to request (and be served) a resource schema.

4.1.3 Data validation

When a client sends a request, the content-type header must contain the custom Media-Type defined in the framework, and the information regarding the schema location (variable schema=<schema location url>).

The framework retrieves the schema and handles resource validation. The developer will not need to create custom code for this.

4.1.4 Response format

We decided that all responses served by services using the framework must follow the HAL pattern. It must be possible to define the "_links" and "_embedded" information.

A resource might contain "_embedded" information if the served resource contains one or more resources related to him.

The "_link" information will be used to pass the information to the clients about the operations that the client can invoke and how to call them. The links defined in the HAL pattern do not provide a mean to directly describe the action (POST, PUT, etc.) that must be performed with the Uniform Resource Locator (URL), like in other designs (like SIREN), by default, the action is a GET.

To overcome this problem, we decided to look at the Internet Assigned Numbers Authority (IANA) link [1] specification. The specification states a set of actions that are identifiable by the name associated to them. Using the same strategy, we define a set of keywords (some belong to the IANA) that the clients can map to a set of possible operations. All the returned links will need to follow the pattern defined in the table 4.1.

Keyword	HTTP Method	URL pattern	Description
self	GET	/<Resource name>/<Resource id>	link to retrieve the current resource (and embedded if they exist)
get	GET	/<Resource name>/<Resource id>	retrieve a single resource (and embedded if they exist)
edit	PUT	/<Resource name>/<Resource id>	update a resource data
delete	DELETE	/<Resource name>/<Resource id>	delete a resource (and embedded if they exist)
collection	GET	/<Resource name>	retrieve a set of resources (pagination rules apply)
create	POST	/<Resource name>	create a new resource entry
schema	GET	<Schema base URL>/<Resource class name>	retrieve a resource schema

Table 4.1: Base set of Keywords

The collection must follow pagination rules, because of that, we define, in the table 4.2, a specific set of links that those responses must contain.

Keyword	HTTP Method	URL pattern	Description
self	GET	/<Resource name>?page=<page number>	link for current page
first	GET	/<Resource name>?page=<page number>	link for first page
prev	GET	/<Resource name>?page=<page number>	link for previous page
next	GET	/<Resource name>?page=<page number>	link for next page
last	GET	/<Resource name>?page=<page number>	link for last page

Table 4.2: Pagination keywords

[1] `https://www.iana.org/assignments/link-relations/link-relations.xhtml`

In this case the response format (listing 4.1) for this request follows the format defined in Talkdesk's documentation.

Listing 4.1: HAL Fruit resource

```
{
  "count": "<number of resources found>",
  "total": "<total number of resources>",
  "_links": {
    "self": {
      "href": "http://api.com/<resource name>?page=<actual page number>"
    },
    "first": {
      "href": "http://api.com/<resource name>?page=<first page number>"
    },
    "prev": {
      "href": "http://api.com/<resource name>?page=<previous page number>"
    },
    "next": {
      "href": "http://api.com/<resource name>?page=<next page number>"
    },
    "last": {
      "href": "http://api.com/<resource name>?page=<last page number>"
    }
  },
  "_embedded": {
    (...)
  }
}
```

In the response root, we have the number of returned elements ("count"), the number of total elements ("total"), the pagination links ("_links") and finally, the result list has an embedded resource ("_embedded").

4.2 Implementation

To simplify the development process of services using a micro-service architecture, we propose a framework that handles all the logic of input validation, response generation and schema handling. The implemented modules are compliant with the set of rules defined in section 4.1.

The developed framework, figure 4.1, consists of the following components:

- Schema resource - contains the Schema API, this module allows for a quick deploy of the Schema related endpoints.
- Schema validator - this component, will handle the resource validation, it will receive a resource and the corresponding schema.
- Response creator - handles the response encoding according to HAL specification.
- Exceptions - this component is a repository for the custom exceptions thrown by the framework.

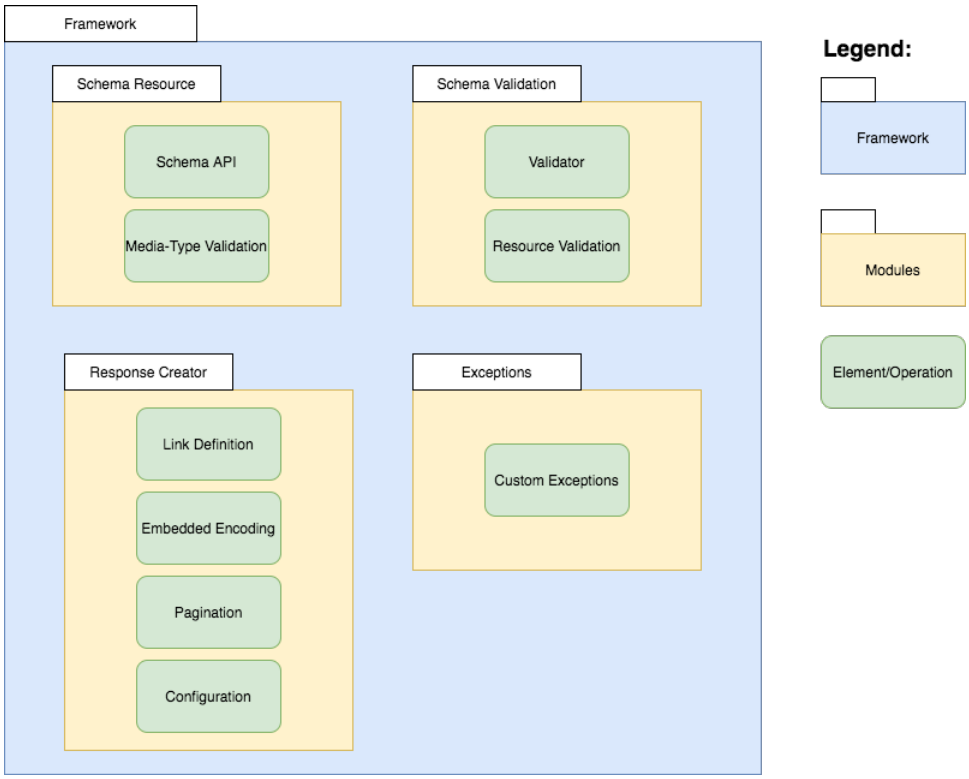


Figure 4.1: Framework overview

The framework specification took some inspiration from micro-services architecture. Each module as two key properties: they can work decoupled of the rest, and each module takes care of an essential part of framework workflow.

We developed the framework codebase in java. This programming language is the same used by the development team while developing the other services, allowing for a seamless transition of the services. Maven takes care of dependency management.

To verify the JSON input against a schema, we use the "com.networknt:json-schema-validator" [2] dependency.

We decided to use the Microprofile [3] and Jakarta EE[4] specifications to help in the development of the framework. Microprofile defines the "Jackson"[5] framework as one of the possible tools to handle JSON marshalling and unmarshalling, because of that, we decided to use it across the project to handle all the JSON related operations.

The "ContainerRequestFilter"[6] that we use in the message interceptors belong to the Jax-RS specification (part of Jakarta EE).

[2] https://github.com/networknt/json-schema-validator
[3] https://microprofile.io/
[4] https://jakarta.ee/
[5] https://github.com/FasterXML/jackson
[6] https://access.redhat.com/documentation/en-us/red_hat_fuse/7.4/html/apache_cxf_development_guide/jaxrs20filters

4.2.1 Schema resource

We started by defining the custom media type ("application/schema-instance+json") that the requests must use. This is part of the communication rules 4.1.1, set by the framework.

Before any requests enters the service, it must be validated by the "Media-Type Validation" operation. This is a JAX-RS message interceptor filter that ensures all requests follow the expected pattern, to verify that, the filter validates the content-type header. Only requests that use the media type "application/schema-instance+json" are allowed, while the others are denied with an "Unsupported Media Type" (415) error.

The resources inside the service will be represented using schemas, and clients are able to connect to the service and retrieve the correct schema for the resource they want.

When developing the service, the developer sets the schemas in the resources folder. For now, the file name must follow the following pattern: "<resource simple class name>-schema.json". We followed this pattern because the class name can be extracted and is used as the identifier for the schema, and the "-schema.json" is used to filter if the file contains a schema or not.

During the service startup, this module will read all files present in the service "resources" folder, making them available to the clients or internal validation of requests.

This module also provides the schema retrieval API. When imported, the endpoint "GET: /schemas/<id>" is enabled. The <id> in the endpoint is the resource simple class name. If the resource is found, the endpoint will return an "OK" response with the schema as the body, if not, the endpoint will return a "NOT FOUND" (404) error.

4.2.2 Schema validator

The validation of the payloads was one of the defined rules 4.1.3. After validating the requests Media-Type, we must validate the request content. We created a second filter (also JAX-RS message interceptor) responsible for verifying body content. The filter will call the "Validator" component, using the payload and the corresponding schema, the schema to use in the validation is received in the content-type, using a variable ("application/schema-instance+json;schema=<schema location>"), the schema location can be the resource name or the URL. If the payload is valid, the request proceeds to the service, if not, the request is interrupted, and the client will receive a "Bad Request" (400) error.

This module serves as a wrapper around some schema validation libraries. We defined an interface with the method template that the implementations must follow.

The interface defines the "validateObject" method that receives two inputs:

- "jsonObject" - a string encoded JSON object that represents the payload to be validated;
- "schema" - a string encoded JSON object that represents the corresponding schema.

The "validateObject" implements a void method. It does not return anything in case of a valid "jsonObject", otherwise an exception must be thrown. The exception will contain a set of failed validations (problems) found, each entry in the set includes the element path and a description of the issue found.

4.2.3 Response creator

The last set of rules, we have a module that handles the response encoding, using as base the rules set in 4.1.4. In the figure 4.2, we can see a high-level representation of the resource encoding. The HAL pattern can be divided into how to represent Resources and Links.

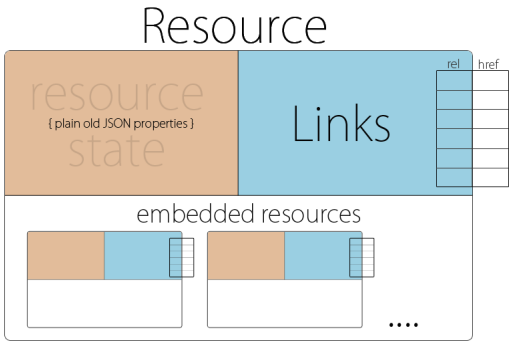


Figure 4.2: HAL model, image from [12]

The resource is composed of a set of links, embedded resources and resource state. The state is the current information that the resource contains inside the service (current state). The embedded resources are other resources inside the larger one, and to simplify, the embedded resources are the objects that are the base for a more complex object. Finally, the links represent the operations available for the resource. They are composed of a target URL and a relation (name/identifier).

In the implemented framework, we developed the tools necessary to automatically encode the API responses using the HAL pattern, without the need for extra code. The implementation process of each tool is explained in the following subsections.

Configurations

For now, the framework works using some configuration files. We defined two configuration files with information about how the service works. First, we have the "operations.json" file to define the next available operation based on the resource type, and the current invoked endpoint.

Listing 4.2: operations.json file structure

```
{
  "<resource simple class name>": {
    "<current operation 1>": [
      "<next allowed operation 1>",
      (...)
      "<next allowed operation n>"
    ],
    (...)
    "<current operation n>": [
      "<next allowed operation 1>",
```

```
      (...)
      "<next allowed operation n>"
    ]
  },
  (...)
  "<other resource simple class name>": {
    (...)
  },
}
```

The listing 4.2 is a simple representation of how to configure the service discovery. In the JSON root level, we have the identifier for the resources, and we use the simple class name for that. Inside of each element in the JSON, we have another level used to identify the available operations. Finally, for each operation, we have a list of strings.

The list elements are used to define the next possible operation and create the links used to invoke the required endpoint to perform that operation. The "current operation" and "next allowed operation" identifiers belong to the keyword list defined in the table 4.1.

The other configuration is taken care of by "definition.json" file. This file contains the resource structure, giving the framework the necessary information to create the embedded representation if needed.

Listing 4.3: definition.json file structure

```
{
  "<resource simple class name>": [
    {
      "class": "<resource 2 simple class name>",
      "name": "<resource 2 JSON key>",
      "url": "<resource 2 URL path>"
    },
    (...)
  ],
  "<resource 2 simple class name>": [
    (...)
  ],
  "<other resource simple class name>": [
    (...)
  ],
  (...)
}
```

The listing 4.3 is a simple representation of how to configure the service for encoding the response following the HAL pattern. Like the operations file, in the JSON root level, we have the identifier for the resources. Inside each element in the JSON, we have another level to identify the embedded resources (more detail in subsection 4.2.3).

The "class" key, is the simple class name of the embedded resource, the "name", is the original key in the JSON encoding of the original resource and finally, the "url", is the resource path for later creating the link information (more detail in subsection 4.2.3).

Response manipulation

We wanted the response creation to be handled automatically by the framework, making it possible to convert all the responses according to the HAL pattern. This process is done programmatically, and takes place by formating and manipulating the response contents, adding all the relevant information, like links and embedded references.

Listing 4.4: Fruit resource example

```
{
  "name": "banana",
  "colour": "yellow",
  "vendors": [
      {
        "country": "Portugal",
        "name": "Continente"
      },
      {
        "country": "Portugal",
        "name": "Pingo Doce"
      }
  ]
}
```

The listing 4.4 is an example of a "Fruit" resource that is used by a service. The "Fruit" resource contains information about the fruit name and colour, but it also has another resource inside of it, the fruit vendor ("Vendor" resource). In this case, the fruit vendor is an embedded resource, and following the HAL pattern, we must format the response accordingly.

Listing 4.5: definition.json for fruit resource example

```
{
  "fruit": [
    {
      "class": "vendor",
      "name": "vendors",
      "url": "vendors"
    }
  ]
}
```

Using the configuration 4.5, we create the proper definition.json file that describes the "Fruit" resource structure. After that, the framework has a series of algorithms (1 and 2) to parse the response and handle the embedded reference's creation.

Algorithm 1 halResponseEncoder(result, operation, simpleClassName)

1: Convert the result to <json> representation
2: **if** json contains "id" value **then**
3: Call method linkCreation(simpleClassName, operation, id) to create the resource links based on the current operation
4: Append the link information to the original json using the key "_links"
5: **else**
6: Throw an error

```
 7: end if
 8: Get resource <definition> using the "simpleClassName"
 9: if definition exists then
10:     if json has embedded elements then
11:         for Each embedded do
12:             Retrieve embedded information (element, element name, element clas) from
    json and added it to a <embeddedList>
13:             Remove that element from json
14:         end for
15:         Append the result of the method embeddedBuilder(embeddedList, operation)
    to json using the key "_embedded"
16:     end if
17: end if
18: Return Updated json
```

The algorithm 1, receives a resource, the current operation and the resource class name. First, the algorithm creates the "_links" information (more details in subsection 4.2.3), we then append them to the resource.

After creating the links, the next step is to access the existence of embedded resources. First, we get the resource definition configured in "definition.json" file, using the resource class name.

The next step is to find the embedded resources, using the "name" value from the resource configuration and place them inside the "_embedded" element. We retrieve the embedded JSON nodes and put them in a list, and send the list to the method defined in the algorithm 2, the result is then put in the "_embedded" element.

Algorithm 2 embeddedBuilder(embeddedList, operation)

```
 1: Create <baseJson> with an empty json object
 2: for Each embeddedList element <e1> do
 3:     if e1 is an array then
 4:         for Each element <e2> in the array do
 5:             Recursively call the method halResponseEncoder(e2, operation, e1 class)
 6:             Add result to an auxiliary <aux> list
 7:         end for
 8:         Update baseJson with aux using e1 name as the key
 9:     else
10:         Recursively call the method halResponseEncoder(e1, operation, e1 class)
11:         Update baseJson with the call result using e1 name as the key
12:     end if
13: end for
14: Return Updated baseJson
```

The second part of the algorithm (algorithm 2), accepts a resource list and the current operation.

First, we create an empty JSON to save the embedded resources after processing. The embedded resources must follow the same HAL pattern, with links and embedded resources; this can be done by recursively calling the algorithm 1.

For each element in the list, we have to check two possible options. It can be a JSON-ARRAY or not. In the first case, we recursively call the algorithm 1 for each element of the array and place them in a list of processed elements; In the second case, we call the algorithm 1 directly. We put the results into the JSON created at the beginning of the method, using the original names.

In the end, the "Fruit" resource will look like example 11 (Appendix H).

Link creation

The HAL pattern dictates that the responses had link information inside the resources, to aid the navigation. The framework creates the link information based on the configurations defined in the "operations.json" file. Using the example of the "Fruit" class 11, if we define the following operations:

Listing 4.6: HAL Fruit resource

```
{
  "fruit": {
    "get": [
      "schema",
      "edit",
      "delete",
      "collection"
    ],
    (...)
  },
  "vendor": {
    "get": [
      "schema",
      "edit",
      "delete"
    ],
    (...)
  }
}
```

For this example, we will see how the framework creates link information when the client wants to retrieve a single fruit resource from the service.

Algorithm 3 linkCreation(base url, resource class name, operation, resource id)

1: Get list of next available requests <requests> using the resource class name and operation
2: Create a list of links <links> for all the possible operations, "self", "schema", "delete", "create", "get", "edit", "collection"
3: **for** each link in links **do**
4: **if** link not in requests **then**
5: Remove the link from the list
6: **end if**
7: **end for**
8: Return links

We use the algorithm 3 to generate the resources links to aid the navigation by the clients.

First, we need to know all the operations that the current resource allows. That information is in the "operations.json" configuration file.

The next step is to create a list with all possible links, using the URL pattern present in table 4.1. Then, using the allowed operations, we remove the links that are not necessary.

For the "get" operation, the "Fruit" resource will look like example 12 (Appendix H).

If the client performs a collection request he must receive a pagination encoded response (defined in the response format rules), we developed the method "createPagination" (algorithm 5) into the framework to handle the creation of this response automatically.

Algorithm 4 createPagination(page, totalPageAvailable, totalResourceAvailable, simpleClassName, resourceList)

1: Create a base json <base json> with {"count": <resourceList size>, "total": <totalResourceAvailable>}
2: **for** each element <element> in resourceList **do**
3: **if** element contains "id" value **then**
4: Call method linkCreation(simpleClassName, "get", id) to create the resource links based on the current operation
5: Append the link information to the original element using the key "_links"
6: **else**
7: Throw an error
8: **end if**
9: Add the element to a list <auxList> of processed resources
10: **end for**
11: Append the auxList to the base json using the "_embedded"
12: Call the method createPageLinks(page, totalPageAvailable, simpleClassName)
13: Append the result to the base json using the "_links"
14: Return json

The "createPagination" algorithm starts by creating the base response ("count" and "total") data, after that, it takes each element of the resource list and created the links (algorithm 3) for them. After the list is updated, it puts the elements in the base response as an embedded resource.

Algorithm 5 createPageLinks(page, totalPageAvailable, simpleClassName)

1: Create a list of links <links> for all the possible operations, "self", "first", "prev", "next", "last", "schema", "create"
2: **if** page equals 1 **then**
3: Remove "prev" link
4: **end if**
5: **if** page + 1 greater than totalPageAvailable **then**
6: Remove "next" link
7: **end if**
8: Return links

The final step is to create the pagination links, the method "createPageLinks" (algorithm 5) handles this operation.

Like in algorithm 3, to create the pagination links, we build all the possible links. After that, we must check if the "prev" and "next" links can exist; if not, we remove those links.

We then return the links.

For the "collection" operation, the response will look like example 13 (Appendix H).

4.2.4 Exception module

The framework defines some custom exceptions, based on the internal operations and interactions. This module serves as a repository for those exceptions, to get them all in one place for better organization.

This module also contains exception mappers, to avoid clients receiving some untreated exceptions. These mappers will capture custom exceptions, thrown by the service, convert them to a custom error payload and send them to clients.

The error messages will follow the next examples:

Listing 4.7: General JSON error payload

```
{
    "code": "<internal error code>",
    "message": "<error message>",
    "description": "<error description>"
}
```

The listing 4.7 represents the payload returned to clients in case the service throws an exception. The "error" is the corresponding internal error code; the "message" is the error message thrown by the service; finally, the "description" is a brief description of the error that occurred in the service.

Listing 4.8: Resource validation JSON error payload

```
{
    "code": "<internal error code>",
    "message": "<error message>",
    "description": "<error description>",
    "fields": [{
            "name": "<path to json element>",
            "description": "<error description>"
        }
    ]
}
```

The listing 4.8, represents the payload returned to clients if the service throws an exception while validating resources against schemas. It is a simple expansion of the previous payload (listing 4.7). It contains an extra element, the "fields" list. This element will have a list of validation errors. The "name" is the path to the element that failed the validation; the "description", is the issue found during validation.

4.3 Framework limitations

During the development of the framework, we made some concessions. The developer must generate all schemas; He needs to ensure that all resources have a schema; if the resources change the developer must update the schemas.

The framework sets the schema location inside the resources folder, meaning that all schema files must be inside that folder. The search routine can look for schemas inside sub-folders, making the folder organization easier. However, as the search routine grabs all files inside the resources folder, we created a specific pattern (<simple class name>-schema.json) for the name of the schema files so that the framework can filter files.

For some complex resources, the configuration file that defines its structure can also become large and complex. If the developer is not careful enough while writing the definition file, it can lead to unexpected results.

We added some extra steps to define correct links in responses due to the library that we are using to write the link information. The library does not allow for new links to be added the list, nor to dynamically create them. Thus, we always create all possible links and then remove the ones that are not available.

The creation of the link keyword names was necessary because of the HAL pattern's lack of action type definitions. Clients must be able to map keywords to specific actions.

This page is intentionally left blank.

Chapter 5

Case Study

In this section we explore a simple use case devised to test the framework's micro-service implementation. This is a simple client and service that handles some "User" information.

First, we have a simple service overview. Next, we show the implementation steps that we took and finally, we quickly resume the challenges and gains that we got while using the framework.

5.1 Overview

To test the framework implementation, we decided to create a simple micro-service. This service handles the "User" related operations.

The service will contain all the CRUD (create, read, update and delete) and also list (pagination) operations.

We also developed a simple web page (client) to interact with the service. The client will read the metadata in the responses ("_links") and display visual elements accordingly.

All the forms present in the client pages are rendered based in the corresponding schema resource, making it possible to have a generic Frontend to some extent.

5.2 Tools

Like the framework, we developed the "User" service in java and maven for dependency management.

To create the service Application Programming Interface (API) and data management (access databases), we decided to use the "Quarkus"[1] framework. This framework contains, out of the box, the mechanism necessary to perform dependency injection in classes, read and apply configurations, JavaScript Object Notation (JSON) handling, and help define and deploy APIs. "Quarkus" is also the framework that the development team uses in the development of the services.

We used "React"[2] for the frontend implementation because of two factors: it was the only

[1] `https://quarkus.io/`
[2] `https://reactjs.org/`

frontend related framework that I used before, and it already had components developed to render forms based on schemas.

5.3 Service architecture

In the example, we implemented a simple service to manage user information. Two different parts compose the service: the Frontend client, and the Backend service.

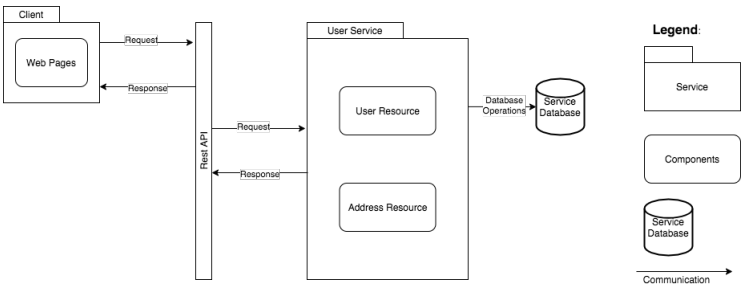


Figure 5.1: User service architecture

Figure 5.1, is a representation of the service architecture. The client contains the web pages used to interact with the resource data and the backend (User Service).

The backend handles two different resources: the User and Address, with different endpoints and actions. The table 5.1 represents all the available actions available for each resource and how to invoke them.

Resource	Operation	HTTP Method	Endpoint pattern	Description
User	create	POST	/users/	Create a new user
	read	GET	/users/<user id>	Retrieve a specific user
	update	PUT	/users/<user id>	Update a specific user
	delete	DELETE	/users/<user id>	Delete a specific user
	list	GET	/users?page=<page number>	Retrieve a set of user
Address	create	POST	/users/<user id>/addresses	Create a new address for a user
	read	GET	/users/<user id>/addresses/<address id>	Retrieve a address from a user
	update	PUT	/users/<user id>/addresses/<address id>	Update a address from a user
	delete	DELETE	/users/<user id>/addresses/<address id>	Delete a address from a user

Table 5.1: Available operations

5.4 Implementation

We started the implementation by defining the service data model for user and address resources, figure 5.2.

The User class is composed of the basic user data, “first name”, “last name” and “age”. It also contains a list of addresses, represented by the “Address” class. Because the User class includes a relation with the Address class, we defined the addresses as embedded resources.

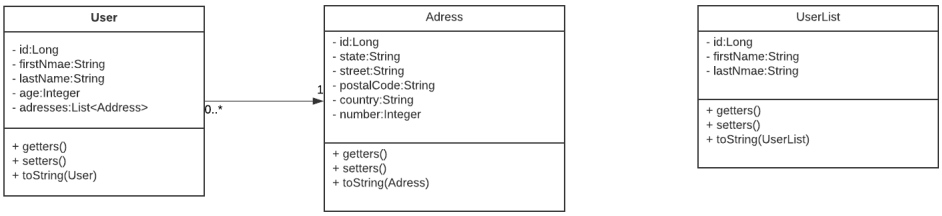


Figure 5.2: Service class Diagram

We also define a simple representation for the users, the UserList class. We use this class to encode the result of "list user" requests. This request produces a collection result with pagination data. We decided to use a second representation for this specific case to simplify the response and also because we did not want to show all the user data in this case.

The next step was to create the flow of operations that the client can perform to a resource and next available actions. This step is essential because it is in here that we can define how to interact with service, allowing for the client to have all the information necessary to perform the service discovery.

5.4.1 Service configuration

After defining the page's behaviour, we need to configure the backend to correctly create the necessary links for each performed operation and correctly encode the response embedded resources.

First, we start by configuring the operations. We prepare this configuration by creating the "operations.json" file, listing 14 (Appendix I), which contains the operations for the two resource types: "user" and "address". The framework will use this file to know which links should it write when a client performs an operation on a resource. Note that, the "self" reference does not appear in this configuration because it is an obligatory link, so we decided to exclude that reference from the configurations.

Listing 5.1: user definition.json file

```
{
  "user": [
    {
      "class": "address",
      "name": "addresses",
      "url": "addresses"
    }
  ]
}
```

Because we have embedded resources (address), the next step was to configure the backend to know how to encode the response into the correct format. To do this, we created the "definition.json" file, listing 5.1. With this configuration, we say to the framework to look for a JSON element named "addresses" and place it as a new element (with the same name) inside the user embedded resource.

Lastly, we are only missing the resources schemas. This configuration is also fundamental

because it will impact the frontend pages but also the resource validation. In the frontend, we use the schemas to render the correct fields in the forms. The framework also uses the schemas to validate the request payload, verify if it contains all the required fields, and the input data respects the constraints. The used schemas can be seen in Appendix J, the listing 15 is the schema referent to the "user" resource and it was placed in a file called "user-schema.json", and the listing 16 is the schema referent to the "address" resource and it was placed in a file called "address-schema.json".

5.4.2 Frontend implementation

We will use this section to show in more detail the frontend implementation. It will discuss the choices in web page implementation.

The client contains three different pages, each one handling a separate operation.

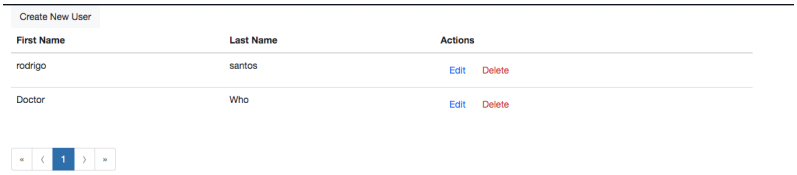


Figure 5.3: User index page

First, we have the index (entrypoint) page, figure 5.3. This page is open by default and displays the list of users in the system, each element in the list must have two buttons, one to edit it and one to delete it. This page must also contain a button to create a new user.

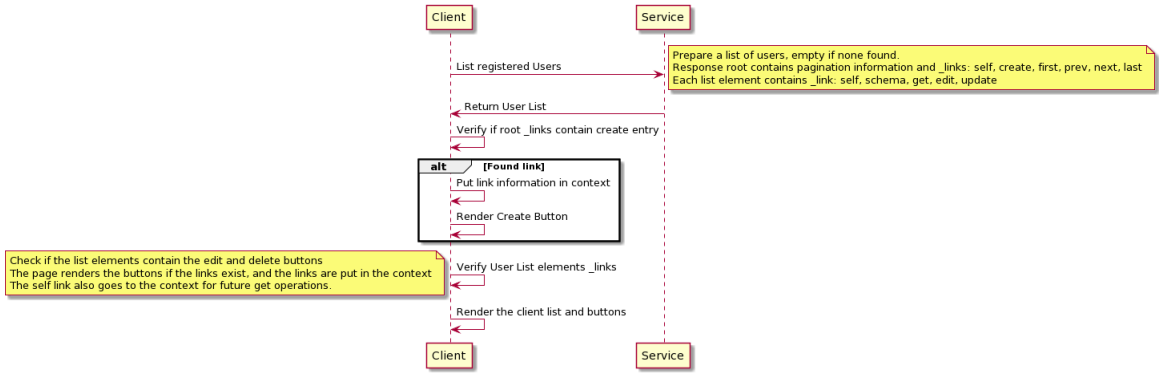


Figure 5.4: Index Interactions

The figure 5.4, is the interactions that exist in the index page. When we are at the index page, the client calls the list endpoint in the service. The service responds with a list of found users. Before sending the client's response, the user list passes in the framework to update it with the pagination fields and links, and the "create" operation link. If there are elements in the list, they appear has embedded resources. Each one will also contain the available operations links; for this example, the delete and update are the ones that matter. When the frontend receives the response, it will verify the root links and see if the create link exists to know if it needs to render the button or not. The same also happens when parsing the "users" list, element links are analysed to verify if we need to render the buttons for the delete and edit operations.

Next, we have the "create page", this page is used to create a new "User". When accessed, the page displays a fillable form, a "Submit" button, to perform the request to persist the data in the backend and finally a "Back" button to return to the previous page.

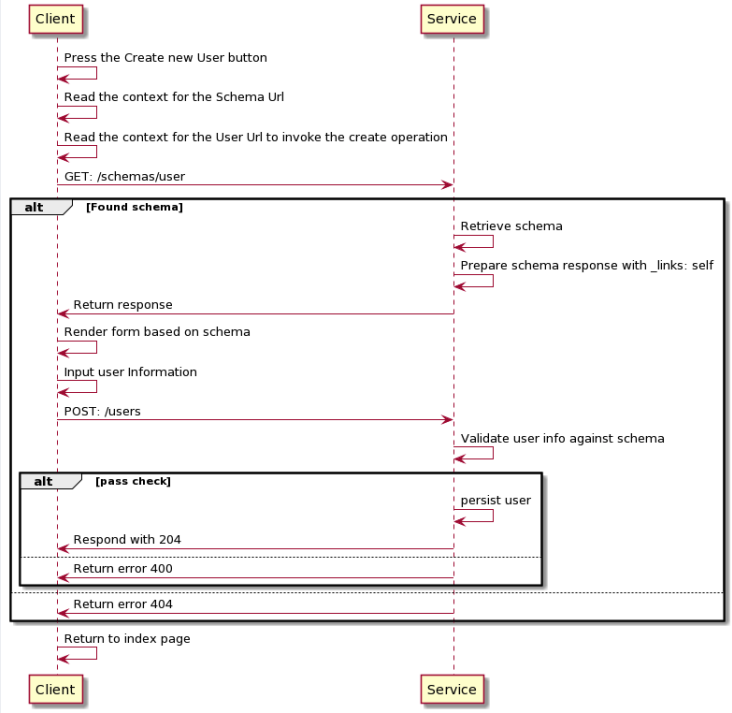


Figure 5.5: Create User Interactions

Figure 5.5, represents the interactions that exist in the create page. When we open the create page the client already knows which endpoints to call and how to call them (the information is already in the context because the index page interactions put it there). The first action is to retrieve the schema necessary to render the form.

Having the schema set, the client can render the page allowing the user to fill the form. The schema has some constraints defined for the inputs, meaning that the page already performs some validations, if the form has wrongly filled fields the submit operation can not occur.

Finally, we can submit the new user data to the backend. We prepare the (POST) request with the correct Uniform Resource Locator (URL); we set the media type to "application/schema-instance+json;schema =<schema URL>"; and the body to contain the input data.

The backend also needs to validate the request content. The framework uses the variable set in the media type to get the schema and validate the user information against it. If all the checks pass, then the data is persisted in the service.

Finally, we have the "edit page", this page is used to check and change "User" data. When accessed, the page displays a filled form with the user data, a "Submit" button, to perform the request to persist the updated user data in the backend and finally a "Back" button to return to the previous page.

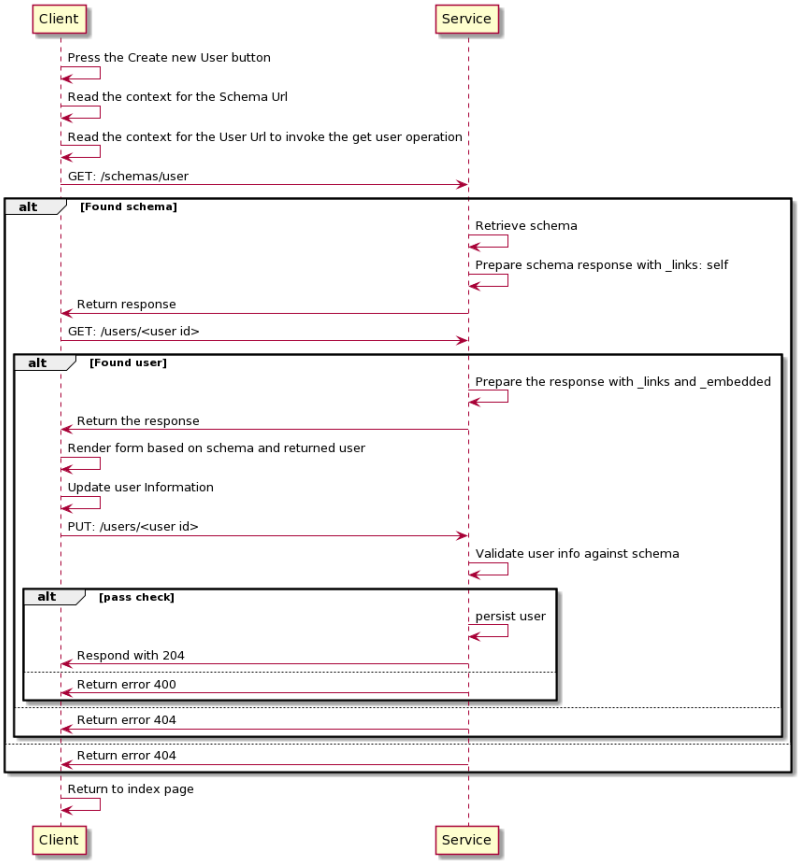


Figure 5.6: Update User Interactions

Figure 5.6, represents the interactions that exist in the update page. The flow is similar to the one in the create page, but with some extra steps. To be able to present the filled form, the page must request user information to the backend. With the schema and the user information, the page can then render the form filled and allow the user to submit the changes. The update request (PUT) has the same logic from the create request the schema for validation identified in the content-type header.

Figure 5.7: User index page without edit link

In Figure 5.7 we have the index page but, in the backend configuration, the edit link was not defined. As the frontend did not received that link it did not render the "edit" button for the list elements.

All the requests performed by the frontend, use the custom media type ("application/schema-instance+json") that was created in the framework. Otherwise the requests would be rejected by the framework filters.

5.5 Conclusions

The main challenges in the development of the case study was related to the frontend implementation. Some pages were problematic to develop because of the data manipulation we have to do to obtain all the necessary information. Due to some lack of knowledge on how to interact with the React components and javascript specific programming details, the frontend implementation took some extra effort.

The schema creation also posed some problems; some of the schema's properties had some unexpected effects in the forms.

On the backend side, the use case's implementation was more direct because of previous experience dealing with this kind of service implementation, and most of the time consuming operations were abstracted by the framework.

This page is intentionally left blank.

Chapter 6

Planning

In this section the work plan is presented. There is the task definition for each semester and the expected time for each task.

6.1 First Semester

The work done in the first semester is shown in the Gantt chart that appears in the following image 6.1 [1].

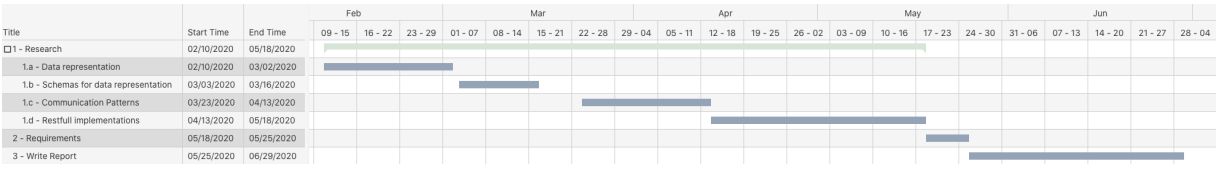


Figure 6.1: Work plan for the first Semester

This image 6.1 is displayed on Appendix L in larger size.

The tasks defined for the first semester where as follows:

1. Research:
   (a) Data representation;
   (b) Schemas for data representation;
   (c) Communication patterns;
   (d) RESTfull implementations.
2. Requirements;
3. Write Report.

During this semester, most of the time (10/02/2020 until 18/05/2020) was spent in research. This phase was divided in research about formats for data representation, schemas for representing the data formats, communication patterns used in web services and finally response models used in RESTfull implementations.

[1] Generated using https://online.visual-paradigm.com/pt/diagrams/features/gantt-chart-tool/

Later, a week (17/05/2020 until 25/05/2020) was used to define project requirements, functional and non-functional requirements, architecture, risks and project constraints.

The last weeks (25/05/2020 until 29/06/2020) were used to write the intermediate report.

6.2 Second Semester - Expected work

The expected work for the second semester is shown in the Gantt chart that appears in the following image 6.2 [2].

The expected work for the second semester was divided in the following two groups:

1. Framework development to assist in building API's compatible with the proposed Architecture:
    - Documentation:
        - Definition of API contracts;
        - Definition of tools (libraries ...);
    - Implementation, development of components to automate:
        - Resource validation;
        - Building http responses;
        - Allow definition of dependencies between resources;
        - Allow definition of allowed operations on each resource;
        - Persistence of models in database.
    - Verification:
        - Unit tests;
        - Component tests.
2. Example of API implemented using the framework guidelines (POC):

6.2.1 Task definition

In this section we defined the tasks needed to be implement in the second semester.

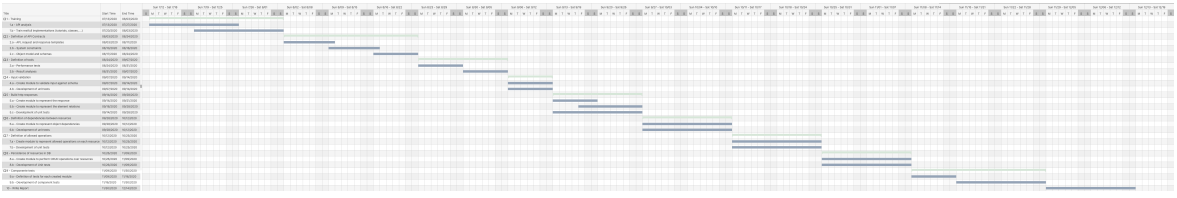


Figure 6.2: Work plan for the Second Semester

This image 6.2 is displayed on Appendix M in larger size.

[2]Generated using https://online.visual-paradigm.com/pt/diagrams/features/gantt-chart-tool/

The tasks devised for this semester are as follows:

1. Training:
   (a) API analysis;
   (b) Train RESTfull implementations (tutorial, classes,....)
2. Definition of API contracts:
   (a) Define Application Programming Interface (API) templates, requests and responses;
   (b) Define system constraints;
   (c) Define object models and schemas.
3. Definition of tools;
4. Resource validation:
   (a) Create module to validate input against schema;
   (b) Development of unit tests.
5. Building http responses:
   (a) Create module to represent the response;
   (b) Create module to represent the element relations;
   (c) Development of unit tests.
6. Allow definition of dependencies between resources;
   (a) Create module to represent the object dependencies;
   (b) Development of unit tests.
7. Allow definition of allowed operations on each resource;
   (a) Create module to represent the allowed operations for each resource;
   (b) Development of unit tests.
8. Persistence of resources in database.
   (a) Create module to perform CRUD operations over resources;
   (b) Development of unit tests.
9. Component tests:
   (a) Definition of tests for each created module;
   (b) Development of component tests;
10. Write report.

It is expected that the development follows a sequential manner, tackling each component in order.

First it is needed to allocate some time to training, this time will be used to learn about existing API's and train the development of RESTfull services.

Then there will be the documentation step, the API contracts that the services must follow will be defined as well as the tools that will be used.

The development steps come in third. In those steps the modules defined in the architecture will be implemented. At the same time all the unit tests will also be developed to ensure that the modules have the correct implementation.

After the development of the modules, there will be the component testing, this tests will test the interactions between the modules to ensure that the service is working as intended.

Lastly is the writing of the final report.

6.3 Second Semester - Actual work

The work done in the first semester is shown in the Gantt chart that appears in the following image 6.3 [3].

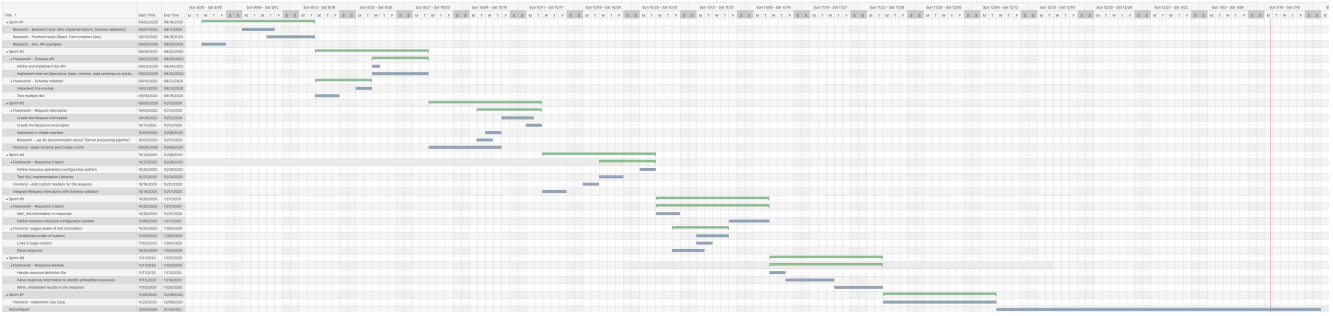


Figure 6.3: Actual Work plan for the Second Semester

This image 6.3 is displayed on Appendix N in larger size.

We divided the work done this semester in two-week sprints. Next we have a list of tasks that were performed during the sprints. The tasks are divided into “Framework” and “Frontend” implementations, and “Research” work.

- Sprint #1 - Start: 02/09/2020 End: 16/09/2020
    1. Research - backend tools (HAL implementations, dchema validation)
    2. Research - frontend tools (React, Form creation libraries)
    3. Research - HAL API examples
- Sprint #2 - Start: 16/09/2020 End: 30/09/2020
    1. Framework - schema API
        - Define and implement the API
        - Implement internal operations (save, retrieve, read schemas on startup)
    2. Framework - Schema Validator
        - Test multiple validation libraries
        - Implement the module

[3]Generated using https://online.visual-paradigm.com/pt/diagrams/features/gantt-chart-tool/

- Sprint #3 - Start: 30/09/2020 End: 13/10/2020
    1. Framework - Request interceptor
        - Research - Jax-Rs documentation about "Server processing pipeline"
        - Implement a simple example
        - Create the request interceptor
        - Create the resource interceptor
    2. Frontend - Parse schema and create a form
- Sprint #4 - Start: 02/13/2020 End: 28/10/2020
    1. Framework - Response Creator
        - Define resource operations configuration pattern
        - Test HAL implementation Libraries
        - Integrate request interceptor with schema validator
    2. Frontend - Add custom headers for the requests
- Sprint #5 - Start: 28/10/2020 End: 11/11/2020
    1. Framework - Response creator
        - Define resource structure configuration pattern
        - Add _link information to response
    2. Frontend - pages aware of link information
        - Parse response
        - Conditional render of buttons
        - Links in page context
- Sprint #6 - Start: 11/11/2020 End: 25/11/2020
    1. Framework - Response creator
        - Handle resource definition file
        - Parse response information to identify embedded resources
        - Write _embedded results in the response
- Sprint #7 - Start: 25/11/2020 End: 09/12/2020
    1. Frontend - Implement case study
- Report writing - Start: 09/12/2020 End: 18/01/2021

The first sprint was dedicated into researching and evaluating the tools we could use to ease the development. We started developing the framework in the next sprint by creating the modules that handle the schemas, the API and internal operations, and the module that can use the schema to validate a resource.

In the third sprint, we started by calling the validation module from inside the service, but it was suggested that instead, the framework should use the Jax-RS request filters, to capture the requests before they hit the service API, and perform the validations there. In this sprint, we started to try to integrate the new features with the frontend. We started by requesting a schema and render it in a form.

The next three sprints were focused on developing the "Response creator" module. From all the development this module was the one that took more time because it was the more complex. During these sprints, we were also testing if we could use the new responses in the frontend.

Lastly, we have the final (development) sprint dedicated to creating the case study that we showed in section 5.

The remainder of the time was for writing the final document.

Chapter 7

Conclusion

After analysing the different modules that constitute the Agent Assist service, we concluded that most of the clients had similar code, only changing the forms used to represent the resources and the endpoint Uniform Resource Locator (URL)s called to perform the actions.

To ease the development time and code maintainability, we proposed creating a framework that could handle the encoding of more information in the service responses so that the client could read that information and know how the service works. The clients should also be able to render the forms using a high-level resource representation, in the end we should have a "generic client".

This internship's primary goal was to propose an architectural design and implement the first version of a framework to ease the service development. The framework must be able to:

- Handle any resource;
- Store and provide high-level resource representations;
- Annotate the responses with information necessary to perform operations on the resources;
- Validate the resource data;

To achieve the proposed goals, we decided to first, study various topics, like possible data representation patterns, architectural communication patterns for Web Services (such as Representational State Transfer (REST)) and lastly standard implementations of RESTful web services.

We decided to use the JavaScript Object Notation (JSON) as the resource representation, and as a consequence, the resources are described using JSON Schemas. The service Application Programming Interface (API) will follow the REST design pattern, and Hypertext Application Language (HAL) was chosen as the RESTful pattern for the responses.

We also devised the architecture for the framework and the contract rules that the service and clients must follow. We took the modularity as the most important requirement while designing the framework. In the "contract rules", we defined the communication aspects that the implementation must follow.

To validate the implemented framework, we devised and implemented a simple case study.

7.1 Main accomplishments

In the end, this internship was completed successfully. We were able to fulfil the two main goals: have the first implemented version of the framework; decouple the client from the service implementation.

The current version of the framework is ready for developers to start developing services that respect the proposed contract rules. Right now the framework already abstracts multiple operations that otherwise would require extra development time: input data validation using resource schemas; an API to handle the schema requests; response conversion to the HAL pattern.

Because of the service discovery constraint, the response conversion in the response is configurable by resource. The developer can define the "_links" information to determine the next available operation and create a generic embedded resource definition.

We created a simple service that handles "user" information and a client that could render the appropriated forms based on the schema that it receives. Changing the schema structure also changed the visual design of the form.

The client buttons and internal operations are only displayed/called if the specific link is present in the response, the URLs and method to call are also inferred from that element.

Finally, we effectively created a generic frontend client that we can now use across multiple services. This client codebase is simple and easy to maintain because we changed the client structure and removed all the hard-coded information. It can now use the meta-information added to the responses and adapt the visual design and internal calls based on the meta-information he received from the service.

7.2 Future work

As we discussed earlier in the document, the framework is usable, but it is still in his early development state. We can solve some of the configuration-related flows to improve usability.

Right now, the configuration process is a little bit manual. The developer needs to create both configurations files and put all the necessary information inside. We can further improve this step by looking to the Java annotations, like the ones used by "hibernate"[1].

We could create an annotation for the API definition where we said if a client calls an endpoint, then the response should have the following list of links. Something similar can be done for the resource definition, we can create an annotation for a specific field and say that in the parent resource context, that field must be treated as an embedded resource.

The framework has a modular architecture, and because of this, we can still add new features to improve it further. Right now, our response module only supports the HAL pattern. One possible improvement is to expand the response creation module to allow the developer to say which pattern they want for encoding responses.

Lastly, we need to improve the tests done to the framework; most of the work done was manual functional tests.

[1] `https://hibernate.org/`

References

[1] Is json schema the tool of the future? https://www.conceptatech.com/blog/is-json-schema-the-tool-of-the-future.

[2] *The Official YAML Web Site*, 2020 (accessed February 13, 2020). https://yaml.org/.

[3] Xsd - the element, 2020 (accessed March 8, 2020). https://www.w3schools.com/xml/schema_schema.asp.

[4] Mike Amundsen. Collection+json - examples, May 2011. http://amundsen.com/media-types/collection/examples/.

[5] Mike Amundsen. Collection+json - document format, Feb 2013. http://amundsen.com/media-types/collection/format/.

[6] T Bray, J Paoli, CM Sperberg-McQueen, Y Mailer, and F Yergeau. Extensible markup language (xml) 1.0 5th edition, w3c recommendation, november 2008, 2008.

[7] Tim Bray, Dave Hollander, Andrew Layman, and Richard Tobin. Namespaces in xml. *World Wide Web Consortium Recommendation REC-xml-names-19990114. http://www. w3. org/TR/1999/REC-xml-names-19990114*, 1999.

[8] Michael Droettboom et al. Understanding json schema. *Available on: http://spacetelescope. github. io/understanding-jsonschema/UnderstandingJSONSchema. pdf*, 2015 (accessed on March 10 2020).

[9] Martin Dürst and Michel Suignard. Internationalized resource identifiers (iris). Technical report, RFC 3987, January, 2005.

[10] Roy T Fielding and Richard N Taylor. *Architectural styles and the design of network-based software architectures*, volume 7. University of California, Irvine Irvine, 2000.

[11] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, Henrik Frystyk Nielsen, Anish Karmarkar, and Yves Lafon. Soap version 1.2. *W3C recommendation*, 24:12, 2003.

[12] Mike Kelly. Hal - hypertext application language, 2013 (Accessed April 20, 2020). http://stateless.co/hal_specification.html.

[13] Mike Kelly. Json hypertext application language, May 2016 (Accessed April 20, 2020). https://tools.ietf.org/html/draft-kelly-json-hal-00.

[14] Kenneth Lange. The little book on rest services. https://www.kennethlange.com/books/The-Little-Book-on-REST-Services.pdf, 2016 (accessed on April 05 2020).

[15] Markus Lanthaler. Hydra core vocabulary: A vocabulary for hypermedia-driven web apis, Jun 2020. https://www.hydra-cg.com/spec/latest/core/.

[16] Markus Lanthaler and Christian Gütl. On using json-ld to create evolvable restful services. In *Proceedings of the Third International Workshop on RESTful Design*, pages 25–32, 2012.

[17] Michael Linderman. Object oriented communication among platform independent systems across a firewall over the internet using http-soap, November 14 2006. US Patent 7,136,913.

[18] Jonathan I Maletic, Michael Collard, and Huzefa Kagdi. Leveraging xml technologies in developing program analysis tools. In *Proceedings of the ICSE Workshop on Adoption-Centric Software Engineering (ACSE)*, pages 80–85, 2004.

[19] J. Michaud. Hal - hypertext application language, 2018 (Accessed April 20, 2020). `https://tools.ietf.org/pdf/draft-michaud-xml-hal-02.pdf`.

[20] Falco Nogatz and Thom Frühwirth. From xml schema to json schema-comparison and translation with constraint handling rules. *Bachelor Thesis. Ulm: University of Ulm*, 2013.

[21] Nurzhan Nurseitov, Michael Paulson, Randall Reynolds, and Clemente Izurieta. Comparison of json and xml data interchange formats: a case study. *Caine*, 9:157–162, 2009.

[22] J. Roy and A. Ramanujan. Xml: data's universal language. *IT Professional*, 2(3):32–36, 2000.

[23] Seva Safris. A deep look at json vs. xml, part 3: Xml and the future of json. `https://www.toptal.com/web/json-vs-xml-part-3`.

[24] Manu Sporny, Dave Longley, Gregg Kellogg, Markus Lanthaler, Pierre-Antoine Champin, and Niklas Lindström. *JSON-LD 1.1–a JSON-based serialization for Linked Data*. PhD thesis, W3C, 2019.

[25] Kevin Swiber. Siren: a hypermedia specification for representing entities, April 2017. `https://github.com/kevinswiber/siren`.

Appendices

Appendix A

Listing 1: XML Schema Definition (XSD) schema

```
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="https://api.example.com/api/contact-center/Users.xsd">
  <xs:element name="Users">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="user" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:sequence>
              <xs:element type="xs:string" name="firstName"/>
              <xs:element type="xs:string" name="lastName"/>
              <xs:element name="age">
                  <xs:simpleType>
                    <xs:restriction base="xs:integer">
                      <xs:minInclusive value="0"/>
                      <xs:maxInclusive value="120"/>
                    </xs:restriction>
                  </xs:simpleType>
                </xs:element>
            </xs:sequence>
            <xs:attribute type="xs:string" name="id" use="required"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Appendix B

Listing 2: JavaScript Object Notation (JSON) Schema

```
{
    "$schema": "http://json-schema.org/draft-07/schema",
    "$id": "http://example.com/example.json",
    "type": "object",
    "title": "The root schema",
    "description": "The root schema comprises the entire JSON document.",
    "default": {},
    "required": [
        "users"
    ],
    "additionalProperties": false,
    "properties": {
        "users": {
            "$id": "#/properties/users",
            "type": "array",
            "title": "The users schema",
            "description": "An explanation about the purpose of this instance.",
            "default": [],
            "items": {
                "anyOf": [
                    {
                        "$id": "#/properties/users/items/anyOf/0",
                        "type": "object",
                        "title": "The first anyOf schema",
                        "description": "An explanation about the purpose of this
                            ↪ instance.",
                        "default": {},
                        "required": [
                            "id",
                            "first_name",
                            "last_name",
                            "age"
                        ],
                        "additionalProperties": false,
                        "properties": {
                            "id": {
                                "$id": "#/properties/users/items/anyOf/0/
                                properties/id",
                                "type": "string",
                                "title": "The id schema",
                                "description": "An explanation about the purpose of
                                    ↪ this instance.",
                                "default": ""
                            },
                            "first_name": {
                                "$id": "#/properties/users/items/anyOf/0/
                                properties/first_name",
                                "type": "string",
                                "title": "The first_name schema",
                                "description": "An explanation about the purpose of
                                    ↪ this instance.",
                                "default": ""
                            },
                            "last_name": {
```

```
                            "$id": "#/properties/users/items/anyOf/0/
                            properties/last_name",
                            "type": "string",
                            "title": "The last_name schema",
                            "description": "An explanation about the purpose of
                                ↪ this instance.",
                            "default": ""
                        },
                        "age": {
                            "$id": "#/properties/users/items/anyOf/0/
                            properties/age",
                            "type": "integer",
                            "minimum": 0,
                            "maximum": 100
                            "title": "The age schema",
                            "description": "An explanation about the purpose of
                                ↪ this instance."
                        }
                    }
                }
            ],
            "$id": "#/properties/users/items"
        }
    }
}
```

Appendix C

Listing 3: JSON Hypertext Application Language (HAL) Single Embedded Result Representation

```
GET: https://api.example.com/api/contact-center/users/974e7b5c-fd64-4bfe-92a5-ae
    ↪ 60ad16d4bd
{
  "_links": {
    "self": {
      "href": "api/contact-center/users"
    }
  },
  "_embedded": {
    "user": {
      "_links": {
        "self": {
          "href": "api/contact-center/users/974e7b5c-fd64-4bfe-92a5-ae60ad16d4bd"
        }
      },
      "id" : "974e7b5c-fd64-4bfe-92a5-ae60ad16d4bd",
      "first_name" : "Rodrigo",
      "last_name" : "Santos",
      "age" : 30
    }
  },
  "id":"users",
  "name":"Contact Center Users"
}
```

Listing 4: JSON HAL Collection Representation

```
GET: https://api.example.com/api/contact-center/users?page=7
{
    "_links": {
        "self": {"href": "/api/contact-center/users?page=7" },
        "first": {"href": "/api/contact-center/users?page=1" },
        "prev": {"href": "/api/contact-center/users?page=6" },
        "next": {"href": "/api/contact-center/users?page=8" },
        "last": {"href": "/api/contact-center/users?page=17" }
        "search": {
            "href": "/api/contact-center/users?query={searchTerms}",
            "templated": true
        }
    },
    "_embedded": {
        "user": [
            {
              "_links": {
                "self": {
                  "href": "api/contact-center/users/974e7b5c-fd64-4bfe-92a5-
                      ↪ ae60ad16d4bd"
                },
              },
              "id" : "974e7b5c-fd64-4bfe-92a5-ae60ad16d4bd",
              "first_name" : "Rodrigo",
              "last_name" : "Santos",
```

```
            "age" : 30
          },
          {
            "_links": {
              "self": {
                "href": "api/contact-center/users/974e7b5c-fd64-4bfe-92a5-
                    ↪ ae60ad16d4bd"
              }
            },
            "id" : "94135e0a-0778-4cb2-8982-80e3e7bdea21",
            "first_name" : "John",
            "last_name" : "Doe",
            "age" : 100
          }
      ]
  },
  "_page": 7,
  "_per_page": 2,
  "_total": 33
}
```

Listing 5: Extensible Markup Language (XML) HAL Single Embedded Result Representation

```
GET:
    https://api.example.com/api/contact-center/users/974e7b5c-fd64-4bfe-92a5-ae60ad16d4bd
<resource rel="self" href="/api/contact-center/users">
 <id>users</id>
 <name>Contact Center Users</title>
 <resource rel="user"
     href="/api/contact-center/users/974e7b5c-fd64-4bfe-92a5-ae60ad16d4bd">
   <id>974e7b5c-fd64-4bfe-92a5-ae60ad16d4bd</id>
   <first_name>Rodrigo</first_name>
   <last_name>Santos</last_name>
   <age>30</age>
 </resource>
</resource>
```

Listing 6: XML HAL Collection Representation

```
GET: https://api.example.com/api/contact-center/users?page=7
<resource rel="self" href="/api/contact-center/users?page=7">
    <link rel="first" href="/api/contact-center/users?page=1"/>
    <link rel="prev" href="/api/contact-center/users?page=6"/>
    <link rel="next" href="/api/contact-center/users?page=8"/>
    <link rel="last" href="/api/contact-center/users?page=17"/>
    <link rel="search" href="/api/contact-center/users?query={searchTerms}"
        templated="true"/>
    <resource rel="user"
        href="/api/contact-center/users/974e7b5c-fd64-4bfe-92a5-ae60ad16d4bd">
        <id>974e7b5c-fd64-4bfe-92a5-ae60ad16d4bd</id>
        <first_name>Rodrigo</first_name>
        <last_name>Santos</last_name>
        <age>30</age>
    </resource>
    <resource rel="user"
        href="/api/contact-center/users/94135e0a-0778-4cb2-8982-80e3e7bdea21">
```

```
        <id>94135e0a-0778-4cb2-8982-80e3e7bdea21</id>
        <first_name>John</first_name>
        <last_name>Doe</last_name>
        <age>100</age>
    </resource>
    <_page>7</_page>
    <_per_page>2</_per_page>
    <_total>33</_total>
</resource>
```

This page is intentionally left blank.

Appendix D

Listing 7: HYDRA document with Inline context representation

```
GET: https://api.example.com/api/contact-center/users/974e7b5c-fd64-4bfe-92a5-ae
    ↪ 60ad16d4bd
{
   "@context" : {
      "user_id" : {
         "@id" : "http://schema.org/uuid",
         "@type" : "@id"
      },
      "first_name" : "http://schema.org/first_name",
      "last_name" : "http://schema.org/last_name",
      "age" : "http://schema.org/age"
   },
   "@id" : "https://api.example.com/api/contact-center/users/974e7b5c-fd64-4bfe
      ↪ -92a5-ae60ad16d4bd",
   "user_id" : "974e7b5c-fd64-4bfe-92a5-ae60ad16d4bd",
   "first_name" : "Rodrigo",
   "last_name" : "Santos",
   "age" : 30,
   "operations" : [
      {
      "@type" : "UpdateUserAction",
      "method" : "PUT",
      "expects" : {
             "@id": "http://schema.org/User",
             "supportedProperty": [
                 { "property": "first_name", "range": "Text" },
                 { "property": "last_name", "range": "Text" },
                 { "property": "age", "range": "Integer" }
             ]
         }
      },
      {
      "@type" : "DeleteUserAction",
      "method" : "Delete"
      }
   ]
}
```

Listing 8: HYDRA document with Inline context representation

```
GET : https://api.example.com/api/contact-center/users
{
   "@context" : {
      "user_id" : {
         "@id" : "http://schema.org/uuid",
         "@type" : "@id"
      },
      "first_name" : "http://schema.org/first_name",
      "last_name" : "http://schema.org/last_name",
      "age" : "http://schema.org/age"
   },
   "@id" : "https://api.example.com/api/contact-center/users",
   "members" : [
      {
```

```
            "@id" : "https://api.example.com/api/contact-center/users/974e7b5c-
                ↪ fd64-4bfe-92a5-ae60ad16d4bd",
            "user_id" : "974e7b5c-fd64-4bfe-92a5-ae60ad16d4bd",
            "first_name" : "Rodrigo",
            "last_name" : "Santos",
            "age" : 30
        },
        {
            "@id" : "https://api.example.com/api/contact-center/users/94135e0a
                ↪ -0778-4cb2-8982-80e3e7bdea21",
            "user_id" : "94135e0a-0778-4cb2-8982-80e3e7bdea21",
            "first_name" : "John",
            "last_name" : "Doe",
            "age" : 100
        }
    ],
    "nextPage": "https://api.example.com/api/contact-center/users?page=2"
}
```

Appendix E

Listing 9: Collection+JSON document

```
GET : https://api.example.com/api/contact-center/users?page=7

{
    "collection":
    {
        "version": "1.0",
        "href": "https://api.example.com/api/contact-center/users",
        "links": [
            {"rel": "self", "href": "https://api.example.com/api/contact-center/
                ↪ users?page=7"},
            {"rel": "first", "href": "https://api.example.com/api/contact-center/
                ↪ users?page=1"},
            {"rel": "previous", "href": "https://api.example.com/api/contact-
                ↪ center/users?page=6"},
            {"rel": "next", "href": "https://api.example.com/api/contact-center/
                ↪ users?page=8"},
            {"rel": "last", "href": "https://api.example.com/api/contact-center/
                ↪ users?page=17"}
        ],
        "items": [
            {
                "href": "https://api.example.com/api/contact-center/users/974e7b5c
                    ↪ -fd64-4bfe-92a5-ae60ad16d4bd",
                "data": [
                  {"name": "user_id", "value": "974e7b5c-fd64-4bfe-92a5-
                      ↪ ae60ad16d4bd", "prompt": "Identifier"},
                  {"name": "first_name", "value": "Rodrigo", "prompt": "First Name"
                      ↪ },
                  {"name": "last_name", "value": "Santos", "prompt": "Last Name"},
                  {"name": "age", "value": "30", "prompt": "Age"}
                ],
                "links": [
                ]
            },
            {
                "href": "https://api.example.com/api/contact-center/users/94135e0a
                    ↪ -0778-4cb2-8982-80e3e7bdea21",
                "data": [
                  {"name": "user_id", "value": "94135e0a-0778-4cb2-8982-80
                      ↪ e3e7bdea21", "prompt": "Identifier"},
                  {"name": "first_name", "value": "John", "prompt": "First Name"},
                  {"name": "last_name", "value": "Doe", "prompt": "Last Name"},
                  {"name": "age", "value": "100", "prompt": "Age"}
                ],
                "links": [
                ]
            }
        ],
        "queries": [
            {
                "rel": "search", "href": "https://api.example.com/api/contact-
                    ↪ center/users/search", "prompt": "Search",
                "data": [
                    {"name": "search", "value": ""}
```

```
                ]
            }
        ],
        "template": {
            "data": [
                {"name": "first_name", "value": "", "prompt": "First Name"},
                {"name": "last_name", "value": "", "prompt": "Last Name"},
                {"name": "age", "value": "", "prompt": "Age"}
            ]
        }
    }
}
```

Appendix F

Listing 10: SIREN document with Inline context representation

```
 GET: https://api.example.com/api/contact-center/users?page=7
{
   "class": "users",
   "links": [
      {"rel": ["self" ], "href": "https://api.example.com/api/contact-center/
         ↪ users?page=7"},
      {"rel": ["first" ], "href": "https://api.example.com/api/contact-center/
         ↪ users?page=1"},
      {"rel": ["previous" ], "href": "https://api.example.com/api/contact-center
         ↪ /users?page=6"},
      {"rel": ["next" ], "href": "https://api.example.com/api/contact-center/
         ↪ users?page=8"},
      {"rel": ["last" ], "href": "https://api.example.com/api/contact-center/
         ↪ users?page=17"}
   ],
   "actions": [
      {
         "name": "add-user",
         "href": "https://api.example.com/api/contact-center/users",
         "method": "POST",
         "fields": [
            {"name": "first_name", "type": "string"},
            {"name": "last_name", "type": "string"},
            {"name": "age", "type": "integer"}
         ]
      }
   ],
   "properties": {
      "size": "2"
   },
   "entities": [
      {
         "class" : ["user"],
         "links": [
            {"rel": ["self" ], "href": "https://api.example.com/api/contact-
               ↪ center/users/974e7b5c-fd64-4bfe-92a5-ae60ad16d4bd"}
         ],
         "actions": [
            {
               "name": "update-user",
               "href": "https://api.example.com/api/contact-center/users/974
                  ↪ e7b5c-fd64-4bfe-92a5-ae60ad16d4bd",
               "method": "PUT",
               "fields": [
                  {"name": "first_name", "type": "string"},
                  {"name": "last_name", "type": "string"},
                  {"name": "age", "type": "integer"}
               ]
            },
            {
               "name": "delete-user",
               "href": "https://api.example.com/api/contact-center/users/974
                  ↪ e7b5c-fd64-4bfe-92a5-ae60ad16d4bd",
               "method": "DELETE"
```

```
            }
        ],
        "properties": {
            "user_id" : "974e7b5c-fd64-4bfe-92a5-ae60ad16d4bd",
            "first_name" : "Rodrigo",
            "last_name" : "Santos",
            "age" : 30
        }
    },
    {
        "class" : ["user"],
        "links": [
            {"rel": ["self" ], "href": "https://api.example.com/api/contact-
                ↪ center/users/94135e0a-0778-4cb2-8982-80e3e7bdea21"}
        ],
        "actions": [
            {
                "name": "update-user",
                "href": "https://api.example.com/api/contact-center/users/94135
                    ↪ e0a-0778-4cb2-8982-80e3e7bdea21",
                "method": "PUT",
                "fields": [
                    {"name": "first_name", "type": "string"},
                    {"name": "last_name", "type": "string"},
                    {"name": "age", "type": "integer"}
                ]
            },
            {
                "name": "delete-user",
                "href": "https://api.example.com/api/contact-center/users/94135
                    ↪ e0a-0778-4cb2-8982-80e3e7bdea21",
                "method": "DELETE"
            }
        ],
        "properties": {
            "user_id" : "94135e0a-0778-4cb2-8982-80e3e7bdea21",
            "first_name" : "John",
            "last_name" : "Doe",
            "age" : 100
        }
    }
]
}
```

Appendix G

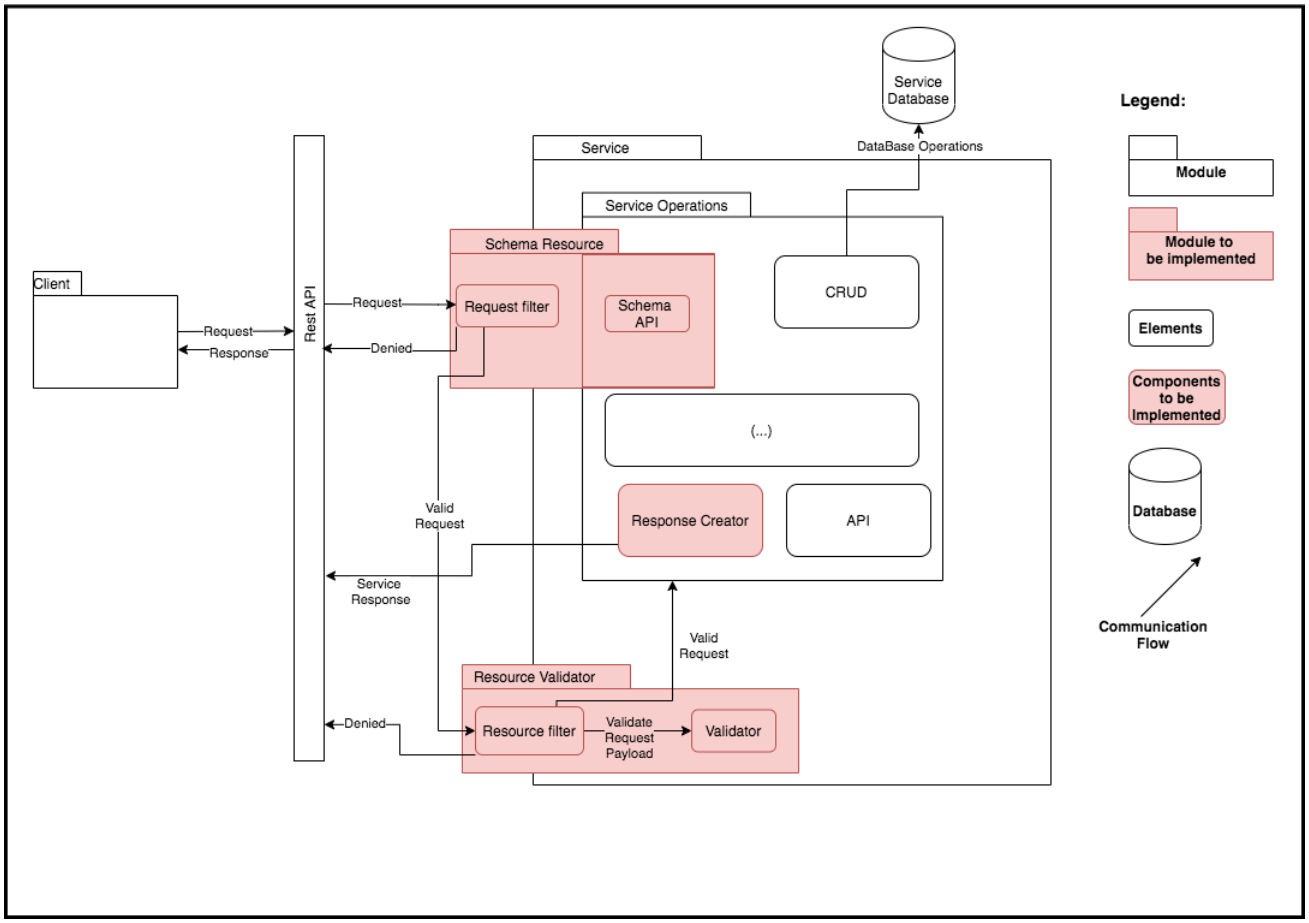


Figure 1: Service Architecture

Appendix H

Listing 11: HAL Fruit response without links

```
{
  "name": "banana",
  "colour": "yellow",
  "_embedded": {
    "vendors": [
      {
        "country": "Portugal",
        "name": "Continente",
        "_links": {
            (...)
        }
      },
      {
        "country": "Portugal",
        "name": "Pingo Doce",
        "_links": {
            (...)
        }
      }
  ]
  },
  "_links": {
    (...)
  }
}
```

Listing 12: HAL Fruit resource with links

```
{
  "name": "banana",
  "colour": "yellow",
  "_embedded": {
  "vendors": [
      {
        "country": "Portugal",
        "name": "Continente",
        "_links": {
            "self": {
              "href": "http://api.com/fruits/1/vendor/1"
            },
            "schema": {
              "href": "http://api.com/schemas/vendor"
            },
            "edit": {
              "href": "http://api.com/fruits/1/vendor/1"
            },
            "delete": {
              "href": "http://api.com/fruits/1/vendor/1"
            }
        }
      },
      {
        "country": "Portugal",
        "name": "Pingo Doce",
```

```
        "_links": {
            "self": {
              "href": "http://api.com/fruits/1/vendor/2"
            },
            "schema": {
              "href": "http://api.com/schemas/vendor"
            },
            "edit": {
              "href": "http://api.com/fruits/1/vendor/2"
            },
            "delete": {
              "href": "http://api.com/fruits/1/vendor/2"
            }
        }
      }
  ]
  },
  "_links": {
    "self": {
      "href": "http://api.com/fruits/1"
    },
    "schema": {
      "href": "http://api.com/schemas/fruit"
    },
    "edit": {
      "href": "http://api.com/fruits/1"
    },
    "delete": {
      "href": "http://api.com/fruits/1"
    }
  }
}
```

Listing 13: HAL Fruit List

```
{
  "count": 10,
  "total": 100,
  "_links": {
    "self": {
      "href": "http://api.com/fruits?page=5"
    },
    "first": {
      "href": "http://api.com/fruits?page=1"
    },
    "prev": {
      "href": "http://api.com/fruits?page=4"
    },
    "next": {
      "href": "http://api.com/fruits?page=6"
    },
    "last": {
      "href": "http://api.com/fruits?page=10"
    },
    "schema": {
      "href": "http://api.com/schemas/fruit"
    },
    "create": {
```

```
      "href": "http://api.com/fruits"
    }
  },
  "_embedded": {
    "fruits": [
      {
        "name": "banana",
        "colour": "yellow",
        "_links": {
          "self": {
            "href": "http://api.com/fruits/1"
          },
          "schema": {
            "href": "http://api.com/schemas/fruit"
          },
          "edit": {
            "href": "http://api.com/fruits/1"
          },
          "delete": {
            "href": "http://api.com/fruits/1"
          }
        }
      },
      (...)
    ]
  }
}
```

Appendix I

Listing 14: User operations.json file content

```
{
  "user": {
    "get": [
      "schema",
      "edit",
      "delete",
      "collection"
    ],
    "edit": [
      "schema",
      "edit",
      "delete",
      "collection"
    ],
    "collection": [
      "schema",
      "create",
      "prev",
      "next",
      "first",
      "last"
    ]
  },
  "address": {
    "get": [
      "schema",
      "edit",
      "delete"
    ],
    "edit": [
      "schema",
      "edit",
      "delete"
    ]
  }
}
```

Appendix J

Listing 15: User schema file

```
{
  "$schema": "http://json-schema.org/draft-07/schema",
  "$id": "http://example.com/example.json",
  "type": "object",
  "title": "The root schema",
  "description": "The root schema comprises the entire JSON document.",
  "required": [
    "firstName",
    "lastName",
    "age"
  ],
  "properties": {
    "firstName": {
      "type": "string",
      "title": "The firstName schema",
      "description": "An explanation about the purpose of this instance.",
      "default": ""
    },
    "lastName": {
      "type": "string",
      "title": "The lastName schema",
      "description": "An explanation about the purpose of this instance.",
      "default": ""
    },
    "age": {
      "type": "integer",
      "title": "The age schema",
      "description": "An explanation about the purpose of this instance.",
      "default": 0
    },
    "addresses": {
      "type": "array",
      "title": "The addresses schema",
      "description": "An explanation about the purpose of this instance.",
      "additionalItems": false,
      "items": {
        "type": "object",
          "properties": {
            "state": {
              "type": "string",
              "title": "The state schema",
              "description": "An explanation about the purpose of this instance.
                  ↪ "
            },
            "street": {
              "type": "string",
              "title": "The street schema",
              "description": "An explanation about the purpose of this instance.
                  ↪ "
            },
            "number": {
              "type": "integer",
              "title": "The number schema",
```

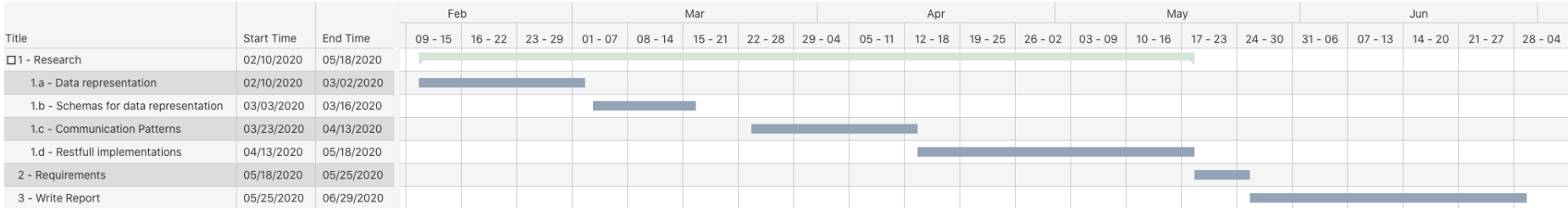
```
            "description": "An explanation about the purpose of this instance.
                ↪ ",
            "default": 0
          },
          "postalCode": {
            "type": "string",
            "title": "The postalCode schema",
            "description": "An explanation about the purpose of this instance.
                ↪ ",
            "default": ""
          },
          "country": {
            "type": "string",
            "title": "The country schema",
            "description": "An explanation about the purpose of this instance.
                ↪ ",
            "default": ""
          }
        },
        "additionalProperties": false
      }


    }
  },
  "additionalProperties": false
}
```

Listing 16: Address schema file

```
{
  {
  "$schema": "http://json-schema.org/draft-07/schema",
  "$id": "http://example.com/example.json",
  "type": "object",
  "title": "The root schema",
  "description": "The root schema comprises the entire JSON document.",
  "default": {},
  "required": [
    "state",
    "street",
    "number",
    "postalCode",
    "country"
  ],
  "properties": {
    "state": {
      "$id": "#/properties/state",
      "type": "string",
      "title": "The state schema",
      "description": "An explanation about the purpose of this instance.",
      "default": ""
    },
    "street": {
      "$id": "#/properties/street",
      "type": "string",
      "title": "The street schema",
      "description": "An explanation about the purpose of this instance.",
```

```
      "default": ""
    },
    "number": {
      "$id": "#/properties/number",
      "type": "integer",
      "title": "The number schema",
      "description": "An explanation about the purpose of this instance.",
      "default": 0
    },
    "postalCode": {
      "$id": "#/properties/postalCode",
      "type": "string",
      "title": "The postalCode schema",
      "description": "An explanation about the purpose of this instance.",
      "default": ""
    },
    "country": {
      "$id": "#/properties/country",
      "type": "string",
      "title": "The country schema",
      "description": "An explanation about the purpose of this instance.",
      "default": ""
    }
  },
  "additionalProperties": false
}
```

Appendix L



Title	Start Time	End Time
1 - Research	02/10/2020	05/18/2020
1.a - Data representation	02/10/2020	03/02/2020
1.b - Schemas for data representation	03/03/2020	03/16/2020
1.c - Communication Patterns	03/23/2020	04/13/2020
1.d - Restfull implementations	04/13/2020	05/18/2020
2 - Requirements	05/18/2020	05/25/2020
3 - Write Report	05/25/2020	06/29/2020

Figure 2: Work plan for the first Semester

Appendix M

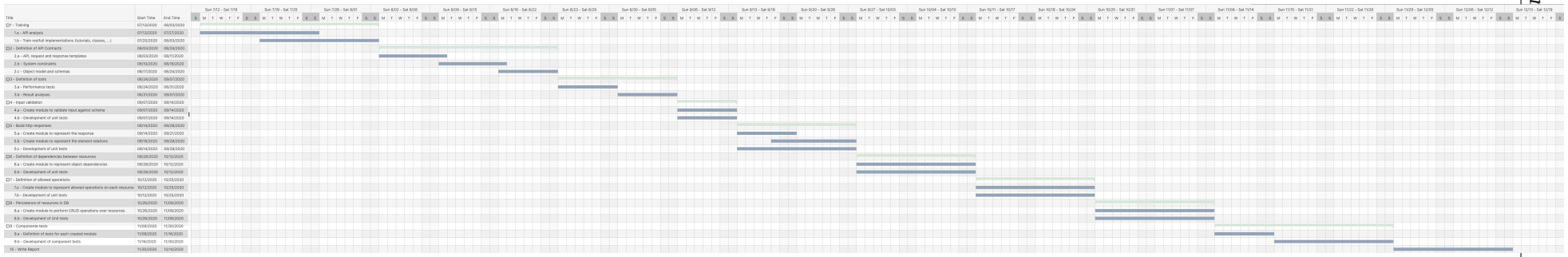


Figure 3: Work plan for the Second Semester

Appendix N

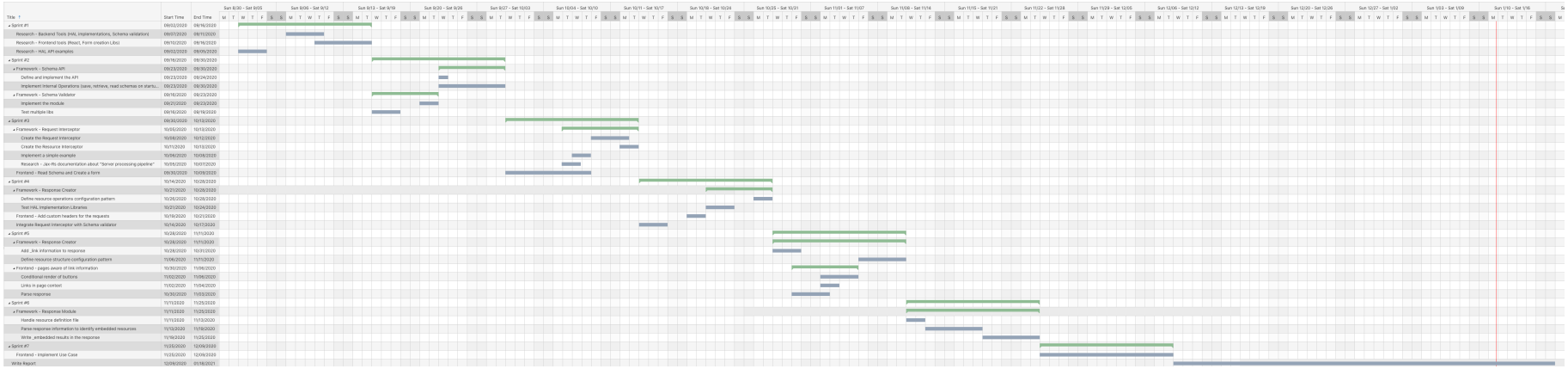


Figure 4: Actual Work plan for the Second Semester