INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Hypermedia-based Web Services as System Integrators

LUCIANO DE CASTRO OLIVEIRA BROCHADO TEIXEIRA
Outubro de 2015

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

Departamento de Engenharia Informática

Hypermedia-based Web Services as System Integrators

An experimentation of the REST approach applied in a BPM context

Luciano de Castro Oliveira Brochado Teixeira

A dissertation submitted in partial fulfillment of the requirements
for the degree of Master in Informatics Engineering
Area of expertise in Architecture, Systems and Networks

Porto, October 25, 2015

Supervisor: Professor Alexandre Bragança

Provisional Version

Instituto Superior de Engenharia do Porto

Hypermedia-based Web Services as System Integrators

An experimentation of the REST approach applied in a BPM context

Luciano de Castro Oliveira Brochado Teixeira

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática
Área de Especialização em
Arquiteturas, Sistemas e Redes

Orientador: Professor Alexandre Bragança

Júri:
Presidente:

Vogais:

Porto, Outubro 2015

Dedication

To the memory of my grandfathers
To my family
To my friends

À memória dos meus avós
À minha familia
Aos meus amigos

Abstract

As we move more closely to the practical concept of the Internet of Things and, our reliance on public and private APIs increases, web services and their related topics have become utterly crucial to the informatics community. However, the question about which style of web services would best solve a particular problem, can raise significant and multifarious debates.

There can be found two implementation styles that highlight themselves: the RPC-oriented style represented by the SOAP protocol's implementations and the hypermedia style, which is represented by the REST architectural style's implementations.

As we search examples of already established web services, we can find a handful of robust and reliable public and private SOAP APIs, nevertheless, it seems that RESTful services are gaining popularity in the enterprise community. For the current generation of developers that work on informatics solutions, REST seems to represent a fundamental and straightforward alternative and even, a more deep-rooted approach than SOAP. But are they comparable? Do both approaches have each specific best suitable scenarios? Such study is briefly carried out in the present document's chapters, starting with the respective background study, following an analysis of the hypermedia approach and an instantiation of its architecture, in a particular case study applied in a BPM context.

Keywords: Web Service, REST, SOAP, BPM

Resumo

Devido ao facto de estarmos cada vez mais próximos do conceito prático de *Internet of Things*, assim como da nossa dependência em APIs públicas e privadas estar a aumentar, o tópico de *web services* e outros tópicos relacionados tornam-se bastante cruciais para a comunidade dedicada à área informática.

Pode-se encontrar dois tipos principais de estilos de implementação que se destacam: o estilo orientado a RPC, cujo conceito é representado pelas implementações do protocolo SOAP e o estilo *hypermedia* representado pelas implementações do estilo arquitetural REST.

Ao procurarmos exemplos de *web services* estabelecidos no mercado, é possível nos depararmos com várias APIs SOAP públicas e privadas classificadas como robustas e fiáveis. No entanto, aparentemente, os serviços cujas implementações são orientadas ao estilo arquitetural REST, estão a ganhar popularidade na comunidade empresarial. Para a geração atual de *developers* que trabalham em soluções informáticas, REST aparenta ser uma alternativa mais essencial, direta e até sólida que SOAP. Mas será que são comparáveis? Será que cada abordagem tem o seu cenário de melhor enquadramento? O estudo presente neste documento tenta responder a este tipo de questões, começando com um estudo do *background* correspondente, seguido de uma analise da abordagem *hypermedia* e uma instanciação da sua arquitetura, num caso de estudo aplicado num contexto BPM.

Palavras-chave: Web Service, REST, SOAP, BPM

Acknowledgments

To my supervisor, Professor Alexandre Bragança for accepting the responsibility of overseeing the developed work, for providing me certain technical and theoretical contributions of the technological aspects and, for giving me insight of research and documentation development methodologies.

To Rui Pereira *MSc*, for all the provided support and monitoring contributions during the development of the study and respective experimentations, I express my sincere gratitude.

To Diogo Soares *MA*, João Moreira *MSc* and Maria João Borges *MSc*, for all the advises that they offered me over the realization of the present documentation and its respective graphical design.

Acronyms

API - Application Programming Interface
AJAX - Asynchronous JavaScript and XML
BPM - Business Process Management
BSD - Berkeley Software Distribution
CORBA - Common Object Request Broker Architecture
CORS - Cross Origin Resource Sharing
CPU - Central Processing Unit
CRUD - Create Read Update Delete
CSR - Certificate Signing Request
CSV - Comma Separated Values
EAI - Enterprise Application Integration
ESB - Enterprise Service Bus
HATEOAS - Hypermedia As The Engine Of Application State
HTTP - Hypertext Transfer Protocol
HTTPS - HTTP Over TLS
IDL - Interface Description Language
IANA - Internet Assigned Numbers Authority
IP - Internet Protocol
JAR - Java Archive
JMS - Java Message Service
JSON - JavaScript Object Notation
JSONP - JSON with Padding
MA - Master of Arts
MIME - Multipurpose Internet Mail Extensions
MSc - Master of Science
OMG - Object Management Group
CSRF - Cross Site Request Forgery
PC - Personal Computer

POJO - Plain Old Java Object
POX - Plain Old XML
QoS - Quality of Service
REST - Representational State Transfer
RFC - Request for Comments
RMI - Remote Method Invocations
ROA - Resource-Oriented Architecture
RPC - Remote Procedure Call
RPO - Research Primary Objective
RPQ - Primary Question
RSQ - Research Sub-Question
SB - Service Bus
SOA - Service-Oriented Architecture
SOAP - Simple Object Access Protocol
SSL - Secure Sockets Layer
TCP - Transmission Control Protocol
TLS - Transport Layer Security
UDP - User Datagram Protocol
URI - Uniform Resource Identifier
W3C - World Wide Web Consortium
WADL - Web Application Description Language
WSD - Web Services Description
WSDL - Web Services Description Language
XML - EXtensible Markup Language
XSD - XML Schema Definition

Corporate group

Salvador Caetano Group accepted and hosted the conditions for the development of some of this present document contents, specially the case study's environment and its respective business purpose. The corporate group is a multinational enterprise that has invested in new information technologies and whose businesses surround the automotive development and retail business.

Contents

List of Figures

List of Tables

Listings

Chapter

1 Introduction

This chapter describes the dissertation purpose, its technological context, the main topics that were researched and the methodologies that were followed to do so.

1.1. BPM Context

The corporate group in which the dissertation's case study was carried out, is expediting its business processes with BPM-oriented digital solutions. During the development of some process solutions, the developer's team noticed some common needs between them, such as file generation and sharing, notifications, integration with other applications and the global monitoring of the system and its processes. Further more, some business process steps (or tasks) are practically identical and, could even be re-utilized if they are correctly decoupled and prepared for parameterizable input.

The figure 1.1 illustrates parallel BPM processes in which, the α blocks represent a step of the process in which they all have a common necessity or functionality.

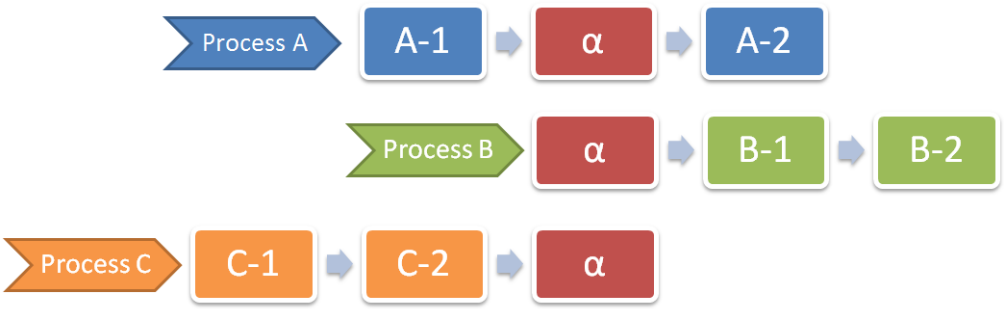

Figure 1.1.: Inter-process common needs.

One common scenario, is a request to obtain or update some entity's data and, a feature of this kind can be implemented as a BPM Suite service, however, if one would intend to reuse

the latter functionality logic, in an external application, in its own architecture, he would have to implement it there as well. Also, if there was the need to update such functionality logic, it would have to be re-implemented in every place it was (Scripts and/or Classes). Due to this previous facts, the importance of a intermediate agent that would provide such types of functionalities, was considered. Further more, if this intermediate had the capability to also interact with the other existing web applications, and possibly, with other systems such as AS/400 and SAP, this intermediate would extend its service range availability to a level beyond the inter-process paradigm. These types of issues seemed to fit well in this dissertation's context and, the respective concepts were considered to be used for the experimentation carried out in the chapter 4. In the following figure 1.2, such middleware is illustrated by the intermediate that connects the *Web Apps* element and the *BPM Engine* element.

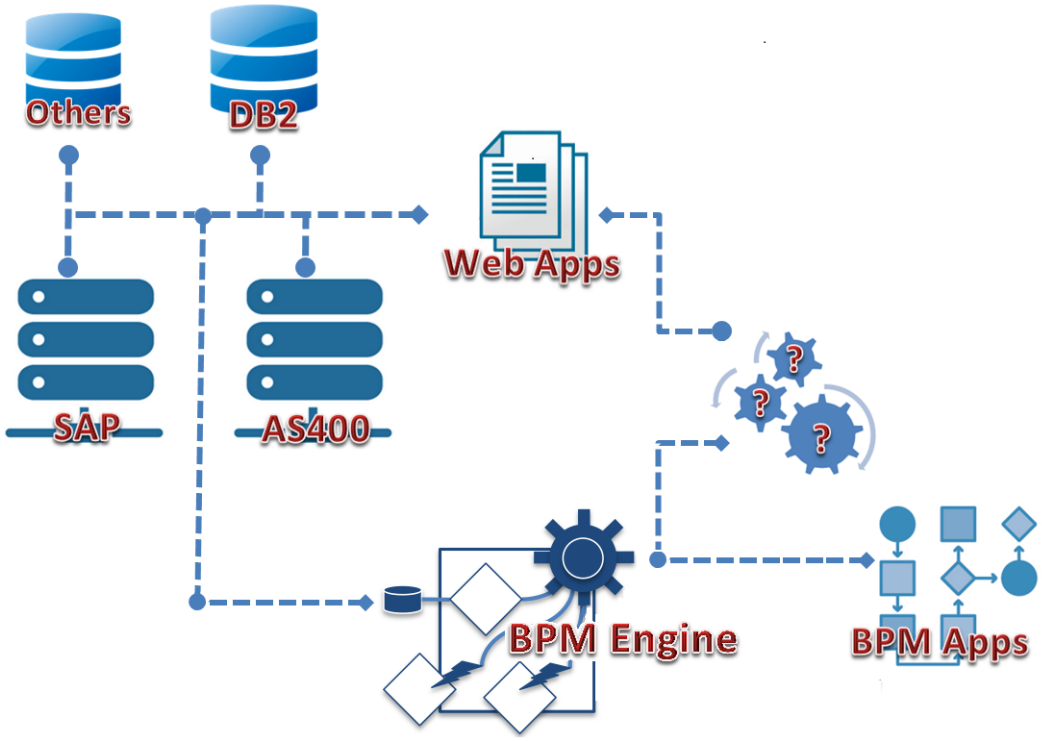

Figure 1.2.: A possible middleware.

Another possible contribution of the weighted intermediary, was the opportunity to cover some gaps of the BPM suite's engine interface and enhance its functionalities. Such advantages are further detailed in the chapter 4 of the present document and also justify why the BPM Engine interface was not chosen to be the direct data source of the elements represented by the *Web Apps* icon in the figure 1.2.

1.2. Dissertation's goals and purposes

The main purpose of this dissertation is to develop a comparative study of the available solutions to the problem presented in the previous section, a selection of a possible solution and, an analysis of its feasibility complemented by a case study consisted of the designing and development of the architecture that best solves this centralization of services and utilities. This document also aims to present the usually implemented techniques in this type of services. Bearing in mind the need of a scalable and highly interoperable solution, the hypothesis that the solution should be web services based, was weighted.

1.3. Research methodology

As one of the main goals of this dissertation was to achieve a design solution for problems similar to the one presented in this chapter, the design science research methodology seemed suitable to follow, in order to organize the necessary work.

Design science research is a paradigm that seeks to improve organization skills, products and services or, its actual performance, via an iterative development process of new and innovative artifacts (Hevner et al., 2004). The resulting knowledge should be able to be applied in design solutions for problems in similar environments. This objectives relate to the sociology and natural sciences goals as well, like theoretical physics, whose main purpose is to develop knowledge to describe, explain and predict (Jiang, 1998).

This iterative development process is assisted by analytical techniques and guidelines to execute the actual research. This analytical techniques should persistently evaluate the artifact that is being built and, it is considered to be crucial that the final artifact must effectively respond to the problem(s) at hand. An application of the design artifact should be instantiated to firmly prove its efficacy and efficiency. The mentioned iterative development process is illustrated in the figure 1.3.

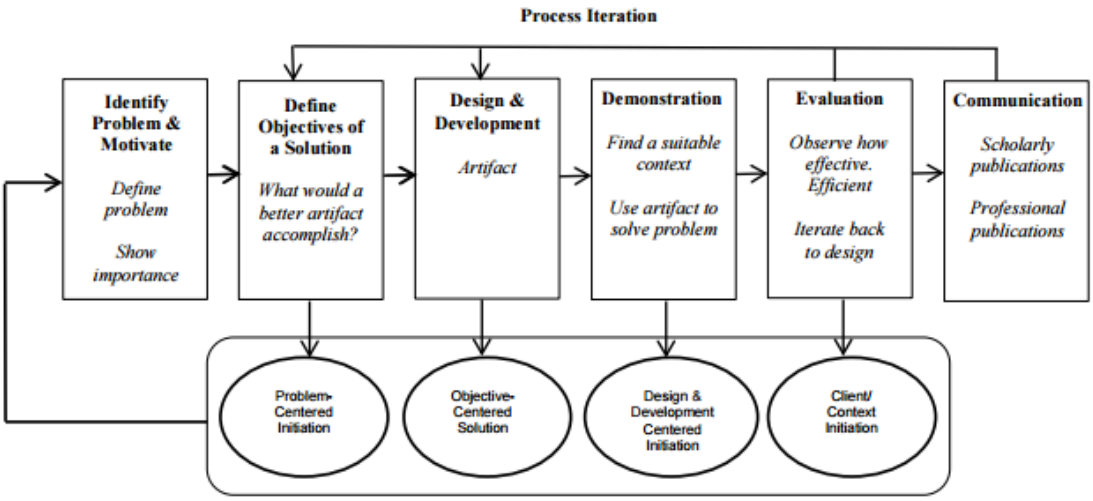

Figure 1.3.: *"Design Science Research Methodology Process Model"* Peffers et al. (2008).

As stated by Dr. Hevner *et al*, there are 7 guidelines required for a successful design science research (Hevner et al., 2004):

1st Guideline- Design as an Artifact:

"*Design-science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation*".

This guideline was planned to be achieved in the production detailed in the chapter 4 - *Case Study*.

2nd Guideline- Problem Relevance:

"*The objective of design-science research is to develop technology-based solutions to important and relevant business problems*".

As stated in the previous sections of the present chapter, the relevancy of the dissertation's context and its objectives relate to this guideline.

3rd Guideline- Design Evaluation:

" *The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods*".

Certain metrics were planned to be measured during the development of the artifact, and, its results are enumerated in the chapter 5 - *Analysis of the results Chapter*.

4th Guideline- Research contributions:

" *Effective design-science research must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies*".

The present dissertation was planned to result in design guidelines for architectures with similar problems and, to synthesize the base and required knowledge to do so.

5th Guideline- Research rigor:

" *Design-science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact*".

As the instantiation of the design artifact was planned to be prepared to be the basis of an enterprise product, such construction and evaluation methods were considered to be crucially required.

6th Guideline- Design as a Search Process:

" *The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment*".

If the actual research was not in accordance with the constraints of the problem, it would increase the chance of the resulting artifact to lack a solid basis.

7th Guideline- Communication of Research:

"*Design-science research must be presented effectively both to technology-oriented as well as management-oriented audiences*".

Such presentation qualities were considered to be important to complement and to substantiate the decisions and, to highlight the final product's perks to the process owners, the final users, the development and support teams.

1.3.1. Research questions

The research work is divided in Research Primary Questions (RPQ) and Research Sub-Questions (RSQ). The main problem stated at the section 1.2 relates to the goal highlighted in the beginning of the previous section, which can be aggregated in the following question:

RPQ → Is it viable to use an Hypermedia-based REST architecture as a solution to integrate the required information systems along with the BPM engine?

This question implies more sub-questions, such as:

RSQ1 ⟶ Why there is the need to combine information systems?
RSQ2 ⟶ Which environments usually surround integration solutions?
RSQ3 ⟶ Which models and architectures are utilized to solve system integrations?
RSQ4 ⟶ Which technologies and techniques can be used on architectures found in *RSQ3* ?
RSQ5 ⟶ Which characteristics should have the desired system?
RSQ6 ⟶ Which metrics can be utilized to classify the final solution as viable?

Besides the answers to the previously itemized questions, there are also Research Primary Objectives (PO) to accomplish, for instance:

RPO1 ⟶ Production of a viable architecture design;
RPO2 ⟶ Execution of tests on an instantiation of the developed architecture design;
RPO3 ⟶ Documentation of design remarks associated with the development of REST APIs.

The following chapters try to solve the main research question by responding partially or fully to the sub-questions. Some brief sections that summarize the answered questions are positioned at the very end of certain chapters.

1.4. Technological environment

Currently the provision of business services (between distributed systems) is mainly carried out in the form of web services. The respective literature highlights well the two existing types, one is the style represented by the SOAP protocol implementations [1] and the other one, is the REST architectural style, which some authors classify as being Hypermedia-oriented.

There are present a considerable amount of SOAP web services in the business world, due to their safety specifications, maturity of their development libraries and in some cases, due to legal issues and guidelines [2]. However, watching the big web players like Amazon and Google, it can be inferred that the current trend in the business world, for the provision of services between distributed systems, has been the partially or fully use of the REST architectural style. Possibly this previous fact is due to the quickness of the client's development and the flexibility of its services, when compared with the SOAP's. Such comparison study was highlighted as an

[1] Despite the fact that SOAP is classified as a protocol, its utilization is defined as a style by some authors. This topic is approached in the section 2.3.1

[2] In a *Diário da República* issue (accompanying this document at the appendixes), it was specified that SOAP, version 1.1, was compulsory to be utilized on the message's structure of exchanged information for the integration of two or more information systems developed for the Public Administration and State owned enterprises Presidência do Conselho de Ministros (2012) .

important goal to support the solution presented on this dissertation, as well as a networking background study, embroiling some other communication technologies.

1.5. Document structure

The following figure 1.4 illustrates the present document's defined organization and its respective structure.

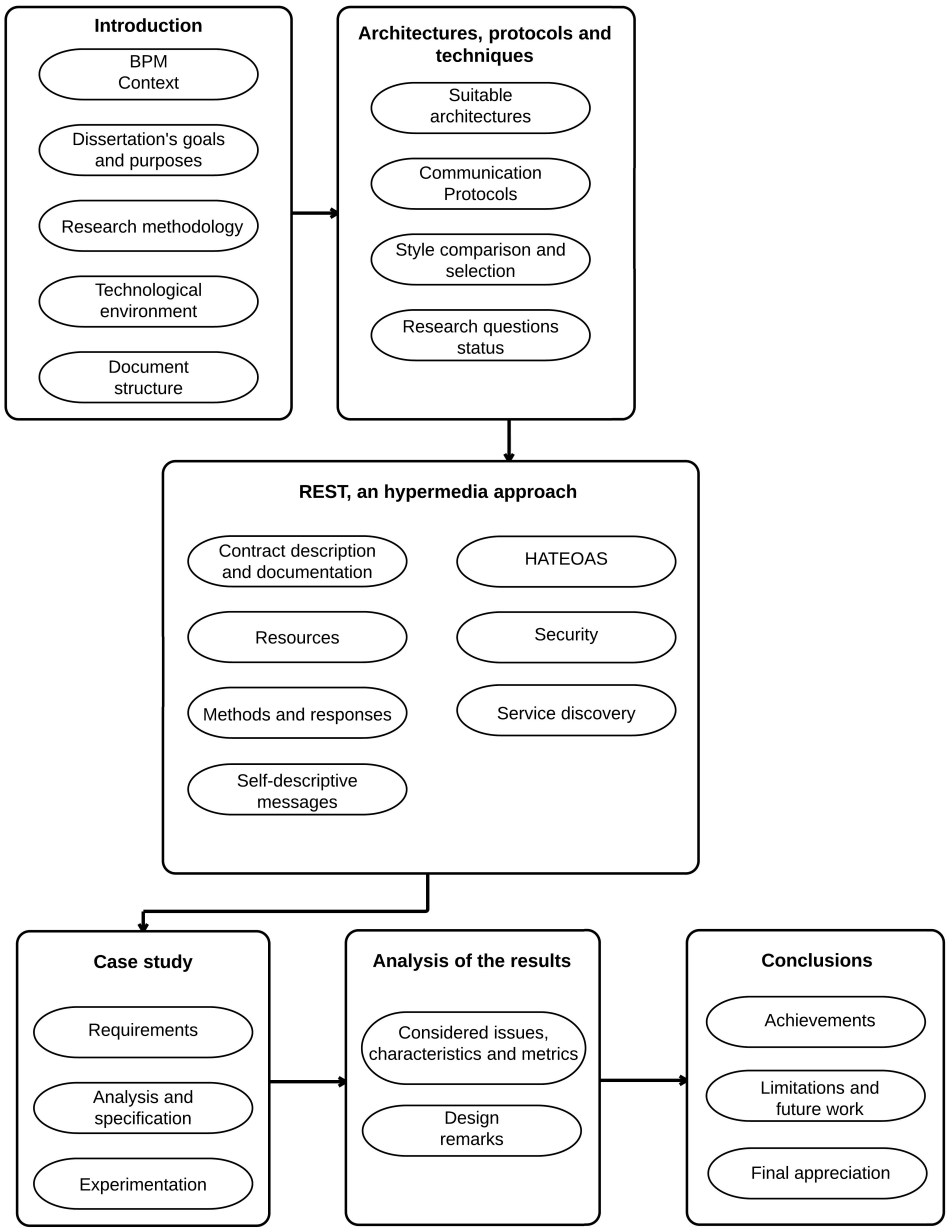

Figure 1.4.: The defined organization of the document's content and its respective structure.

Chapter 2

Architectures, protocols and techniques

Firstly, to help and substantiate the analysis of what kind of intermediate should be implemented, this chapter presents some documented architectural approaches to solve the types of problems presented in the 1.1 section, such as Enterprise Application Integration and Service-Oriented Architecture. Following them, a few communication methods that can be utilized on the previously mentioned architectures, are briefly detailed.

To finalize the present state of the art chapter, a comparison of the two styles that web APIs usually follow, is given. As mentioned in the section 1.2, the studied approaches that were considered for this comparison were the style represented by SOAP protocol implementations and the REST architectural style.

2.1. Suitable architectures

This section presents a succinct view of the main architecture models and deployment scenarios in which applications are usually built on.

2.1.1. Models of distributed systems

Distributed systems usually fit in one of two models, the client-server or the peer-to-peer. Although there are some other models that can be classified under different architectures styles or variations (*e.g.* the event-based architecture's publish-subscribe model), they can all be considered variations of the two main models (Eliassen, 2011; Eles, 2007).

The client-server model implies functional decomposition (Sousa, 2014), for instance a web browser running on a personal computer (PC) that requests content of a given web server. In the peer-to-peer model, as opposed to what occurs in the former model, the peers usually have similar functionalities in the system that they belong (*e.g.* the *Bitcoin* system). An example of

each of these two models is given in the figure 2.1 and in the figure 2.2, respectively.

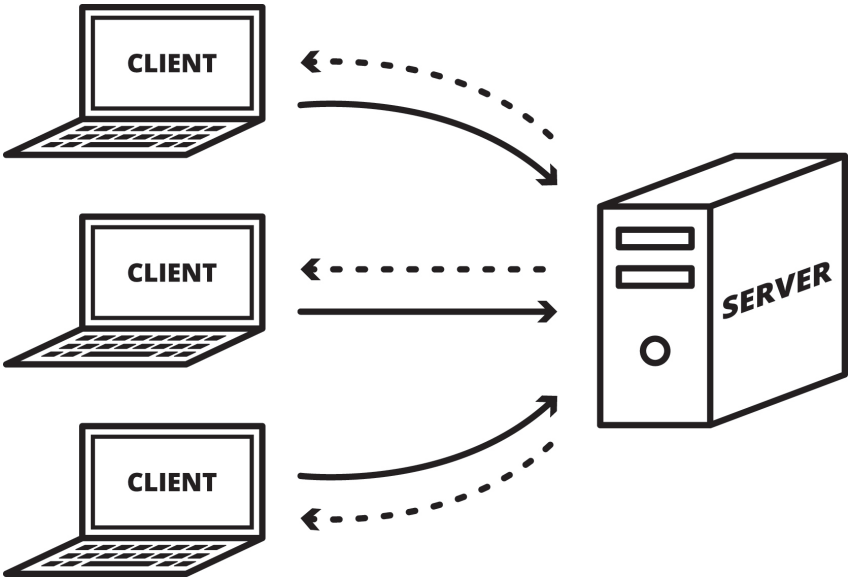

Figure 2.1.: A web server handling requests of multiple clients.

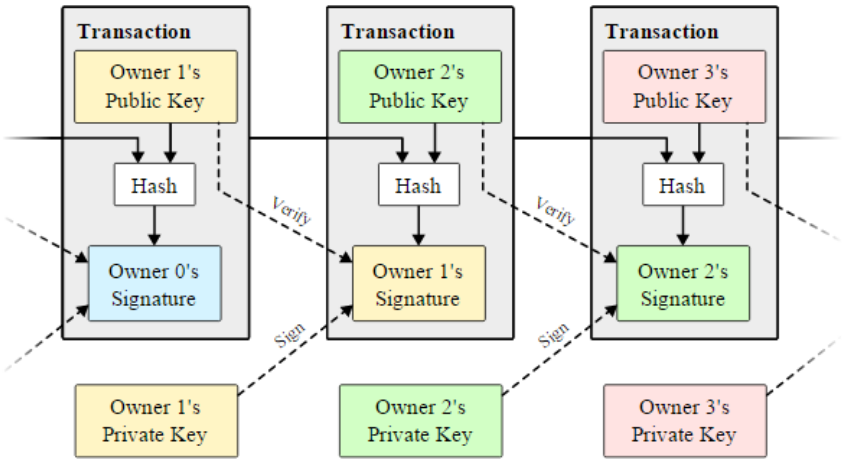

Figure 2.2.: "*Bitcoin transactions occurring in a peer-to-peer model*" Nakamoto (2008)

2.1.2. Deployment scenarios

There can be found three main deployment scenarios concerning distributed systems, which are commonly known and mentioned as the Intranet, Internet and Extranet deployments.

A service that is used internally on an organization, for instance, a solution for inter-departments use cases, is said to be established on an Intranet deployment scenario. The figure 2.3 illustrates an example of it.

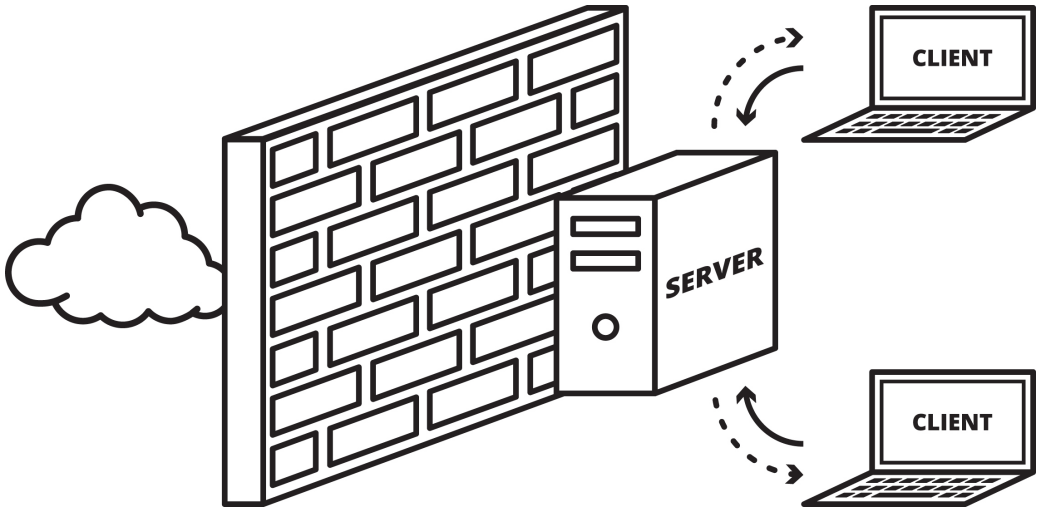

Figure 2.3.: Intranet deployment scenario.

Services that are exposed on the Internet or, to clients that can come from outside of the organization, are considered to be deployed on an Internet scenario. This scenario can be considered more sensible when compared with the previously presented one, due to the fact that the organization usually doesn't have control over the client's intents. Generally, this services usually require authentication and authorization measures. This is illustrated in the figure 2.4.

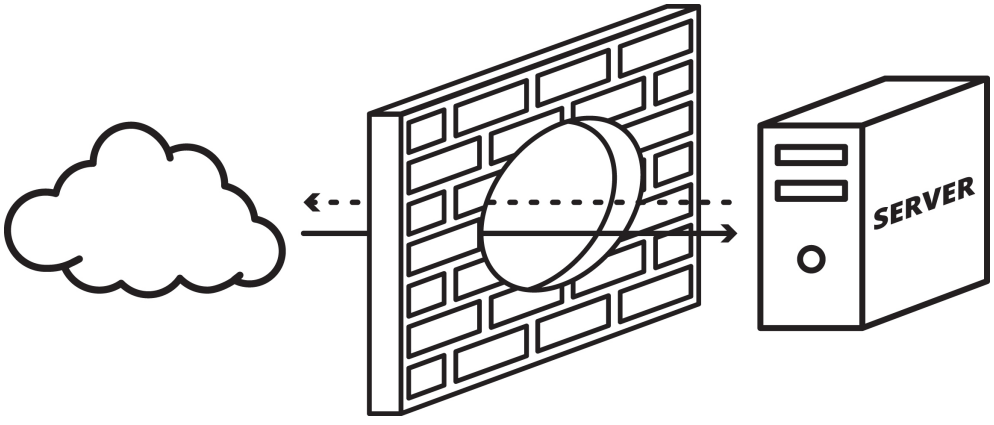

Figure 2.4.: Internet deployment scenario.

Services that are exchanged between ”controlled” endpoints over the Internet (*e.g.* between two Intranets), are considered to be deployed in Extranets scenarios. This deployment is usually applied when the communication between an organization and ”trusted” third-parties is executed (*e.g.* between offices of an organization and a business partner). This scenario is illustrated in the figure 2.5.

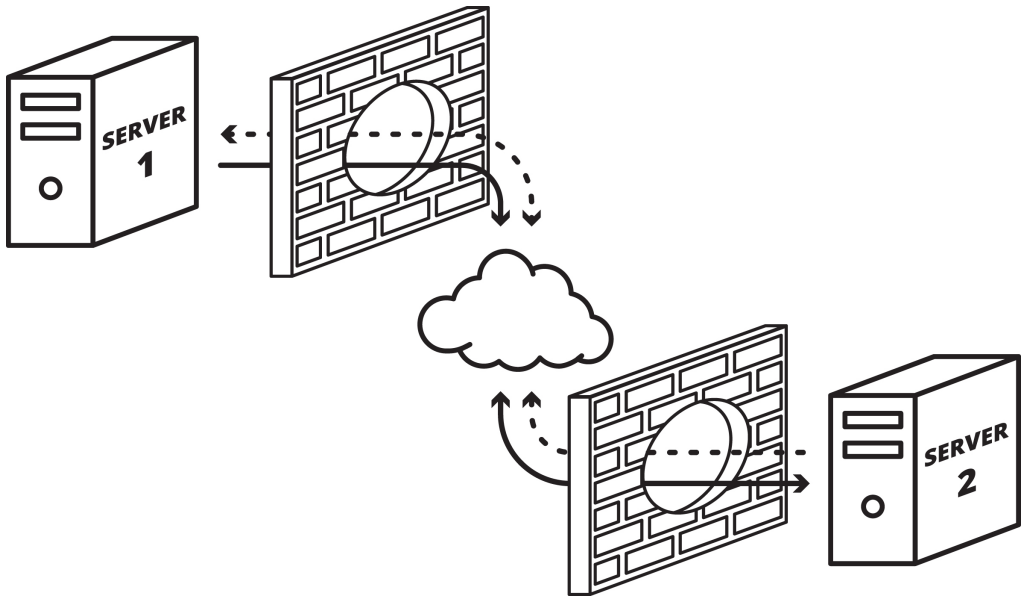

Figure 2.5.: Extranet deployment scenario.

2.1.3. Enterprise Application Integration

The Enterprise Application Integration (EAI) topic became popular around mid-1990, on account of the fact that enterprises already had multiple applications to support their businesses and, the need of the integration between some of them appeared (Clark, 2015). If those integrations were not made properly, each integration required multiple adapters to link each member application and could create a lot of entropy and development efforts. This entropy is illustrated in the figure 2.6, in which, four applications are linked together by the means of the implementation of multiple adapters (or connectors), in each of them.

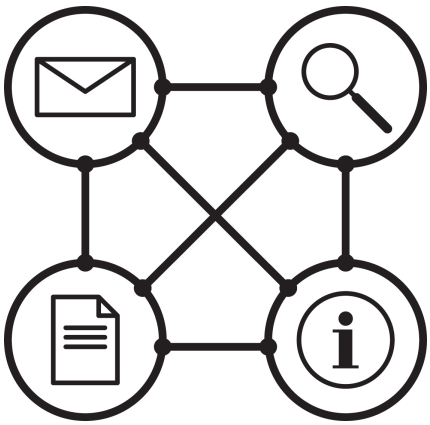

Figure 2.6.: Entropy created by multiple application adapters.

The EAI's architectural principles brought new approaches to respond to this kinds of problems. The traditional approach of this integration pattern usually occurred via the handling of the messages originated by applications that were connected to a "hub-like" system. These systems are also mentioned as Hub and Spoke systems (Hudson, 2003). Generally, this handling involved the analysis, translation, validation and distribution of the messages. Some authors consider that the actual handling of the messaging is the best approach of EAI, on account of the low evasive nature of the implementations and/or procedures that were needed to integrate them and, the loose coupled characteristic of this methodology (Hohpe/Woolf, 2004; Endrei/Keen/Sadtler, 2004). The following figure 2.7 illustrates the mentioned "hub-like" system integrating four applications.

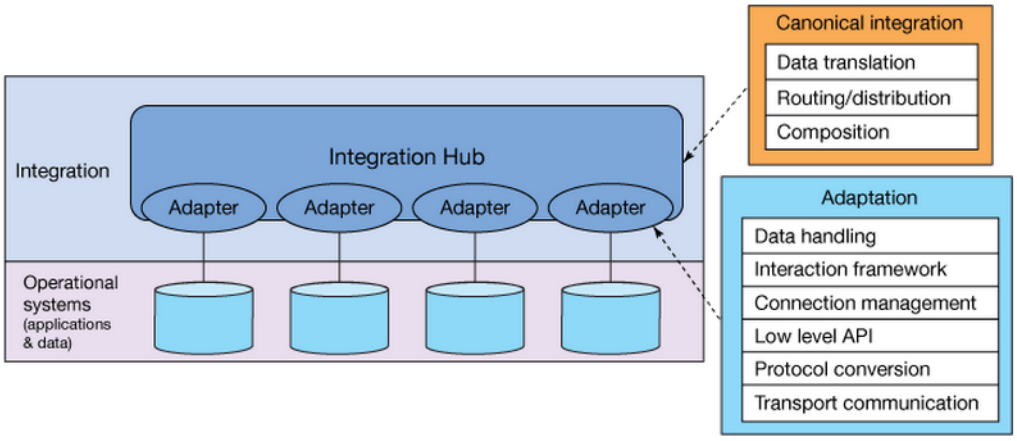

Figure 2.7.: "Hub and spoke architecure" Clark (2015)

When comparing with situations as shown in the figure 2.6, the EAI's approach style simplified a lot of work and implementation efforts, however, each new type of system that was connected to the hub required additional work to connect it, and also, by definition this archi-

tecture gives a centralized point of connection, which can result in a single point of failure to all systems. The latter facts have lead some organizations to stop using this traditional style of EAI in the solutions as popularly as they used to (Baker et al., 2005). These scenarios are depicted in the figures 2.8 and 2.9, respectively.

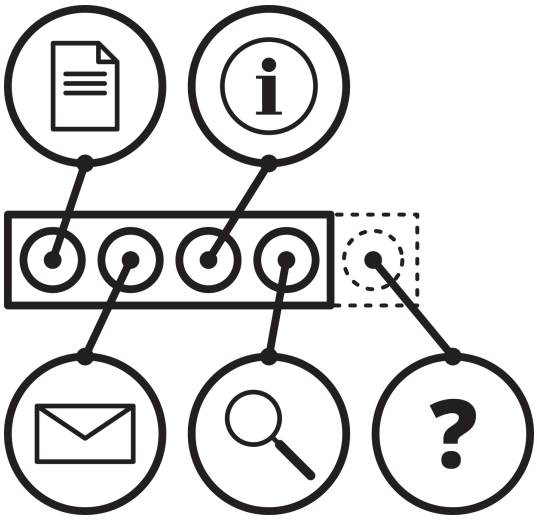

Figure 2.8.: New adapter needed in a hub-like system for application integration purposes.

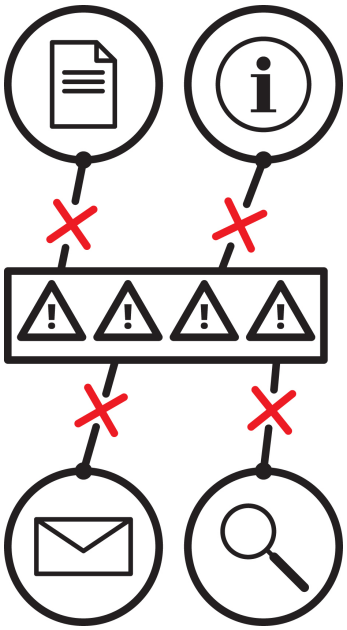

Figure 2.9.: Single point of failure in a hub-like system for application integration purposes.

2.1.4. Enterprise Service Bus

Some of the products developed to respond to the EAI concept presented in the section 2.1.3, such as the mentioned ”hub-like” system, were adapted and repositioned as Enterprise Service Bus (ESB) or Service Bus (SB) products by their developers (Kress et al., 2013). Although the ESB/SB purpose is to solution EAI problems, its technical specifications has yet to be globally agreed by all industry players. None the less, it is generally accepted that its definition is an architectural pattern of the Service-Oriented Architecture (SOA) to respond to the communication and messaging between multiple systems (Clark/Flurry, 2011; Delgado et al., 2007). This SOA concept is further detailed in the section 2.1.5. The following itemization lists key features that some authors identify as being core principles of the ESB definition (Sousa, 2015c; Kress et al., 2013; Flurry, 2007):

- Interconnectivity of systems;
- Separation of concerns by decoupling the senders and the receivers of the messages;
- Service’s orchestration;
  - Utilization of fine-grained services to accomplish richer results;
- Process flow mediation;
- Transformation of the message formats and data types;
  - Such as converting CSV to JSON;
- Negotiation and translation between different transport protocols;
  - For instance, between Java Message Service (JMS) and HTTP.

An illustrative model of an ESB architecture is given in the figure 2.10.

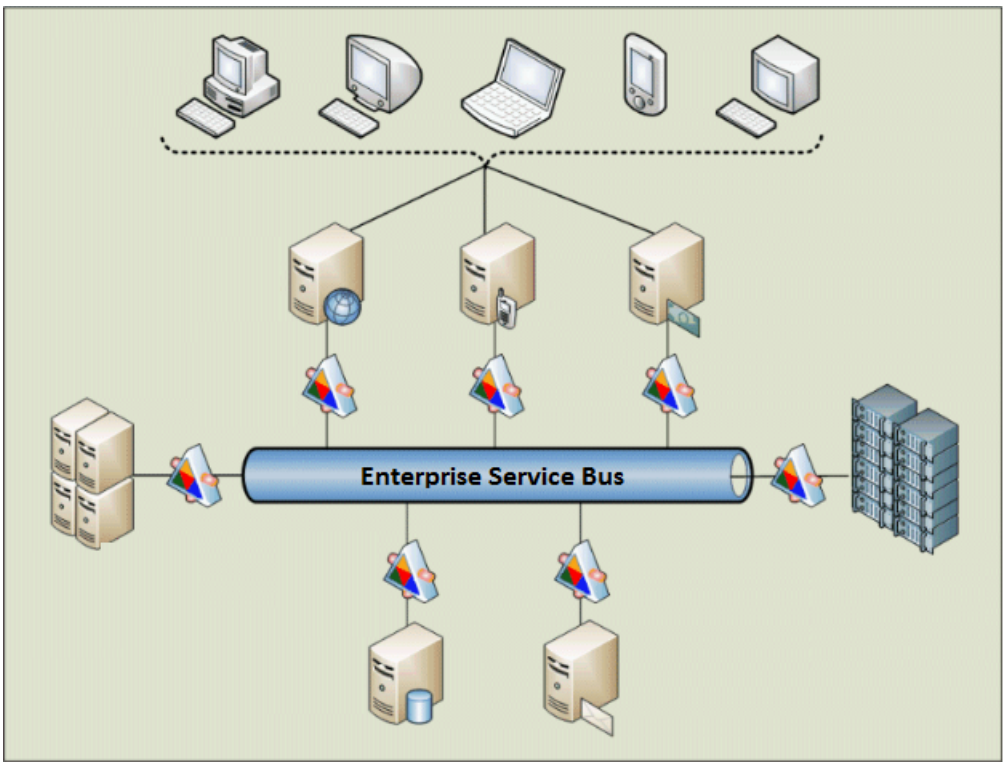

Figure 2.10.: "A high-level example of the connectivity provided the Enterprise Service Bus architecture" Delgado et al. (2007)

2.1.5. Service-Oriented architecture

A system in a Service-Oriented Architecture should aggregate interoperable services that can be utilized by multiple systems and by multiple business domains (Ghosh, 2011) .

Services in the context of distributed systems, are considered modules of business and provide functionality(ies) for applications (Sousa, 2015d) . They should also be reusable and capable of integrating other services to create compositions of services. The resulting services are commonly mentioned as composite services, relating to the composite pattern (Woolf, 2006). Also, services can be divided in four main categories:

1. Entity Services;
2. Utility Services;
3. Task Services;
4. Orchestrated Task Services.

The SOA's definition is also put in four tenets by some authors (Sousa, 2015d; Evdemon, 2015):

- Boundaries are clear and well known;
- A contract with an agnostic (or generalized) data schema should be exposed;
    - Only the schema and contract should be provided, not the programming-language or platform's specific data types, as it could break the interoperability principle;
- The service's compatibility should be defined by policies;
    - As the digital contract cannot describe all the requirements needed for interactions with the service, it should be complemented with additional policy documentation, such as WS-Policy (Vedamuthu et al., 2007)[1];
- Services should be self-contained, in order to ensure the already mentioned service compositions.

An example of a web application taking advantage over of a service-oriented architecture is given in the figure 2.11.

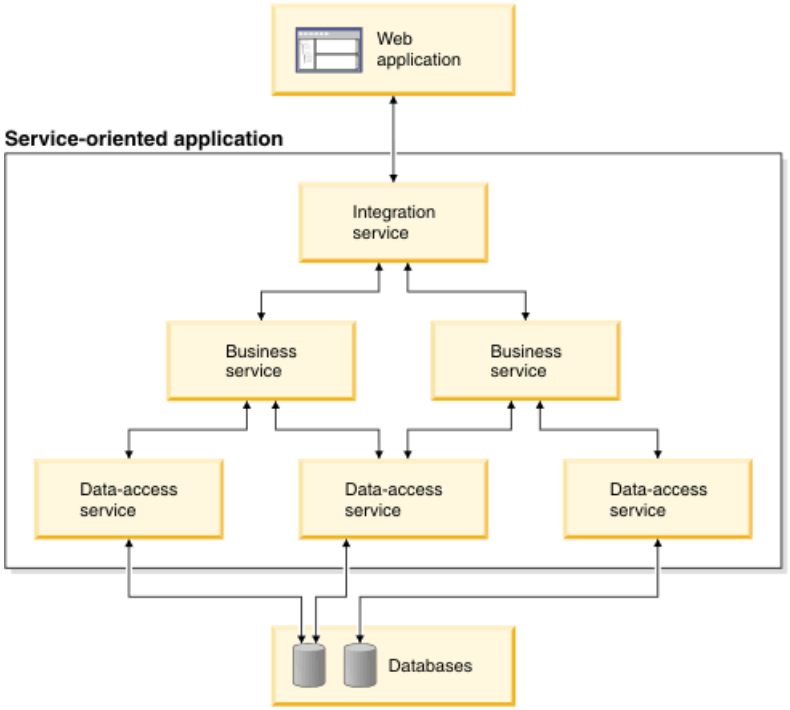

Figure 2.11.: A web application taking advantage of a service-oriented architecture The Eclipse Foundation (2009)

In the figure 2.11, three decoupled data-access services feed two independent business services that are integrated in a single service. This chain of hierarchy illustrates a well designed service-oriented architecture.

[1] An example of a standard policy sharing technique is the WS-Policy, however, there are other ways to accomplish it.

Following the analogy given in the figure 2.11, some authors indicate that the SOA consists in multiple layers. One of these interpretations is depicted in the figure 2.12.

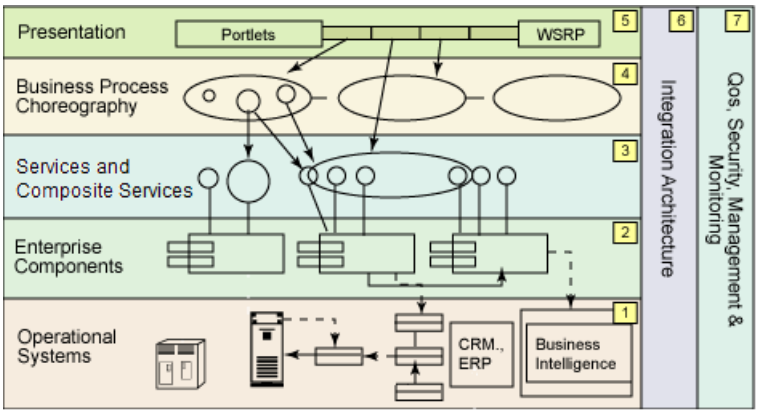

Figure 2.12.: "The layers of a SOA" Arsanjani (2004)

The elements illustrated in the figure 2.11 can be considered to be established in the third layer of the figure 2.12.

There are also some articles and community forums that mention another type of architecture, the Resource-Oriented Architecture (ROA), which is usually related with the REST architectural style. On account of the fact that in this architecture the services are implemented as resources, it's closer to the REST paradigm and by consequence, the web's mechanisms, however, it still follows some of the main principles of the SOA architecture (*e.g.* allow re-usability of services).

2.2. Communication protocols

When planning a distributed system, one of the points to consider is which kind of communication the system should use (*i.e.*, its message structure and its transmission methodology). It will directly sway the final quality metrics of the information system, such as performance and scalability. In most environments, so a communication between two components can happen, both of them must have an Application Programming Interface (API) defined. This API nomenclature basically means a way to allow one piece of software to talk with another one. An API provides a network interface to its system, which is crucial to the intermediate system that was pondered to solve the present dissertation's study problem, and, with this in mind, some APIs will be exemplified as well.

In summary, this section focus on briefly exposing the characteristics of some communication protocols and their respective most suitable scenarios, to substantiate the decision of the followed approach.

2.2.1. Sockets

Like the techniques presented in the following sections, Sockets provide an interface to the network, although they accomplish it in a lower level. This Socket API is, directly or indirectly, used in most of the web applications as it allows applications to run across diverse networks via the TCP/IP protocol.

There are two kinds of Socket APIs, the Stream Socket which reliably [2] sends a stream of bytes to another application, and the Datagram Socket which has the same purpose, but does it unreliably (Shetty, 2007) . The first kind is TCP-based, whereas the second one is UDP based. The present section is focused on the former type, the Stream Socket, and, most of the presented documentation notes was based on the Berkeley Socket's literature.

In the Open Systems Interconnection (OSI) Model, the Socket API in study is positioned between the Session and Transport layer, allowing a connection to the latter and vice-versa (Khan/Khwaja, 2003). The following figure 2.13 depicts the layers defined on the OSI and TCP/IP Models and, this Socket API location.

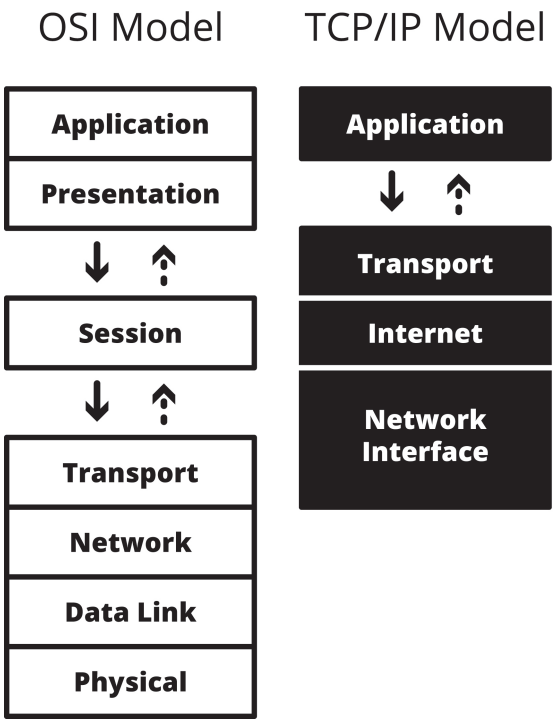

Figure 2.13.: Socket API position in the OSI and TCP/IP Models.

Sockets are endpoints of a data connection, this means, they are present on both sides of the communication channel. The connection can either be between running processes within the

[2] In the present context, reliability concerns the delivery of the data to the intended receivers.

same machine or, between two processes running on remote machines. They are identified by an Internet Protocol (IP) address and a Port number (a 16 bit identified integer). This identification can be either on IPv4 or IPv6 versions and, can be combined with DNS names which is a more commonly accepted approach rather than the exposure of a raw IP address (Perkins, 2015).

Sockets are widely used on low-level networking programming and are largely cross-platform compatible. An example of this technique, are the Berkeley Sockets developed in the C programming language. They were first introduced in the BSD Unix operating system in the year 1983, but are currently available on the majority of the operating systems such as Linux, MacOS X and Windows (FreeBSD Foundation, 2015). Its utilization is based on the following steps:

1. Definition of the socket endpoint;
2. Establishment of the port on the server;
3. Enable connection requests;
4. A connection request is sent to the server endpoint;
5. Server "accepts" the connection;
6. Initiation of read and write operations;
7. The connection is finished and the endpoints are closed.

The next picture 2.14 illustrates the socket endpoint's behavior and some primitives applied in this scenario, based on the previously stated steps.

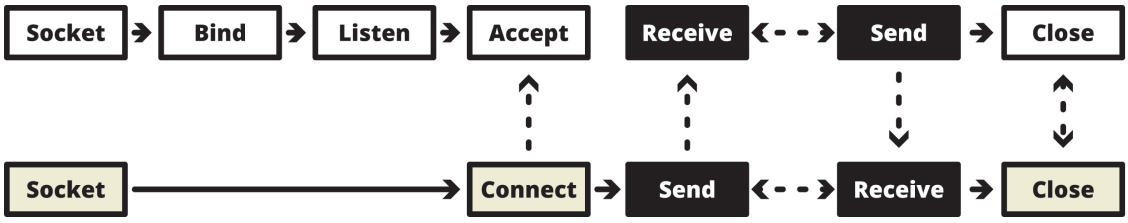

Figure 2.14.: Socket endpoints interaction

An analysis of the figure 2.14 can result in the enumeration of four different steps of the communication, the initialization (from *Socket* to *Listen*), the connection definition (from *Connect* to *Accept*), the data transfer (between *Send* and *Receive*) and the closing (via the *Close*).

The following table (2.1) summarizes the Socket API's primitives and their respective purpose.

Primitive	Purpose
Socket	Creates a communication endpoint
Bind	Links an address and reserves a Port to a Socket.
Listen	Indicates availability to allow connections.
Accept	Accepts an incoming connection
Connect	Tries to establish a connection
Send	Sends data to the other endpoint, with delivery and sorting reliability
Receive	Receives data from the other endpoint
Close	Turns off the connection

Table 2.1.: Socket API primitives.

The Socket primitive only creates the interface, it does not specify the data origin, nor where it will be sent. The Bind primitive can be used to reserve a Port [3]. A Port can only be held by one process at a time, so the Kernel [4] will register every Port that each process holds. The Listen primitive makes the socket available for other connections, by turning it into passive (by default it is active, which makes it not available to other connections). The Accept, Connect and Receive primitives are blocking calls, which means, the program is halted by the operating system until something happens.

There are also two modes, the Blocking mode which is the default upon creation, and Non-blocking. A socket running in Blocking mode, will wait until the data requested by the application is received from the network's endpoint or, until the defined timeout is reached (McGuire).

Due to fact that the functions usually assigned to the Transport Layer are very complex (*e.g.*, wireless link detection), the Socket API is very crucial because it provides simple connectivity functions to the layers above.

Socket API provides a basis to other high-level networking interfaces, such as Remote Procedure Call (RPC) which is further explained in the next section.

2.2.2. RPC

RPC (Remote Procedure Calls) is a protocol to be used in a client-server architecture. The main purpose of RPC implementations is to provide communication between two processes, generally from different machines, but this communication can also be between two processes from the same machine. Usually in this paradigm, the client synchronously requests the execution of a function from a remote server, this is, the client awaits the response from the server before proceeding any further. Although it is possible to make the implementations asynchronous

[3] By combining the Socket API's Bind primitive with some operative system's configuration variables (*e.g.* the Linux's IP *ip_local_port_range* and *ip_local_reserved_ports*), the developers can avoid other process applications to get hold of a certain Port (Linux Kernel Organization Inc, 2015). However, this exclusivity cannot be guaranteed and can result on conflicting applications (*e.g. Mozzila Thunderbird vs Outlook* SMTP port conflict).

[4] Kernel is program that handles input/output procedures of the software and converts them into instructions interpretable by the central processing unit (CPU) (Bovet/Cesati, 2006).

(Schlichter, 2002), some of the synchronization downsides can be mitigated, if the requests are made by a multithreaded program[5].

As RPC has characteristics that might relate some authors to the Application, Presentation and Session layers (Borghoff/Schlichter, 2000; George, 2009; Schmidt, 2012), its specific position in the OSI Model is not globally clear. There is also some discrepancy in this topic documentation, however, the majority of the researched authors defend it has attributes that places it between the 7th and 5th layers. Such discrepancy is probably caused by the fact that some of the RPC aspects and its general applications, cover the client-server model (Application Layer), the structured and transparently presented data handling (Presentation Layer) and the request-answer protocol (Session Layer), however, its independence of the TCP and UDP protocols (Transport Layer) appears to be unanimous.

RPC eases up the distributed application programming as it enhances the distribution transparency. This transparency is due to the hidden low-level networking functions (Rouse, 2015), like data packet acknowledgments, retransmissions and byte sorting. Despite the fact that this functions are hidden, if they are needed in a higher programming level, they can be delegated to an independent software component (*e.g.* a C program), in which they can be re-implemented according the intended purposes.

There are some similar or related implementations that are considered to be ”evolved RPC technologies”, cataloged as Remote Method Invocations (RMI) such as the Common Object Request Broker Architecture (CORBA) and Java RMI implementations (Carballeira, 2015).

2.2.3. Web Services

Web services are web application components that can be published, found and be utilized by other web applications. Their main purpose is to integrate web-based applications, capable of communicating via standard-based protocols (*e.g.* HTTP) (Ghosh, 2011). If it can be configured for that purpose and intention, the HTTP protocol can be established as the underlying communication protocol, which permits requests to pass smoothly through firewalls, as opposed to the RPC implementations.

They are primarily designed to allow businesses to communicate with each other and also with its clients via an interface held by each endpoint. Some typical implementation of web services have informative purposes, such as weather forecasting, order tracking and airfare ratings. Web services are considered to be an essential tool and technique, due to the fact that most softwares cannot be isolated in order to provide top-notch features to their users.

As briefly described on the section *Technological environment* 1.4, there are two kinds of web service's implementation styles, commonly referred as SOAP and REST, which are further detailed in the section *Style comparison and selection* 2.3.

[5] A program that can make use of CPUs which can concurrently process multiple processes or threads.

Web services became popular due to some of their characteristics (Iyengar, 2009):

- Provide the possibility to deliver information to any system, anywhere and anytime;
  - Some can include consumption costs;
  - Information flow between heterogeneous systems is a primary concern;
  - Integration of loosely coupled applications;
- Some of the major software organizations, providers and vendors have worked together to define and improve them, for instance W3C and OASIS;
- A large portion of development tools and middleware that are compatible with web service's standards has become available, such as *Axis2* (Sandakith, 2007);
- Web services provide a more efficient infrastructure for the modeling and integration of business processes and applications.

Such functionalities usually don't come alone, some challenges appear as a trade-off for the web service's mentioned perks. Some of them are itemized bellow:

- Message passing;
- Heterogeneity;
- Security;
- Efficiency;
- Scalability;
- Concurrency;
- Failure handling;
- Performance;
- Transparency;

The challenges mentioned above can also relate to the ones found in general distributed systems. The degree of the system's capability to accomplish this issues can be considered a qualitative or similar-to-qualitative metric.

Due to the fact that interoperability is the highest priority of web services (Sabbouh et al., 2011), they should be independent of programming languages, frameworks, server technologies and platforms in which they "reside". Also, as stated in the W3C web service definition, a machine-processable interface descriptor must be present (Booth et al., 2004). The following enumeration describes the two main specifications of web service descriptors that are generally found.

1. WADL (Web Application Description Language), which supports only the HTTP protocol. This descriptor was specified by the company *Sun Microsystems*;
2. WSDL (Web Services Description Language), this descriptor is a W3C recommendation. As it is independent of the message formats and network protocols of the endpoints it describes, it is compatible with protocols such as SOAP, HTTP and MIME (Christensen et al., 2001).

As both of these service descriptors can be applied in services developed in a ”RESTful” approach, they are more thoroughly detailed in the chapter 3 *REST, an Hypermedia approach.*

2.3. Style comparison and selection

This section is firstly focused on briefly detailing the two styles of web service implementations mentioned in the previous section (*i.e.*, the "RPC-oriented" style represented by SOAP protocol implementations, and, the Hypermedia style referenced as the implementations of the REST architecture). Also, and to finish the analysis of this state of the art chapter, the section 2.3.3 *Suitable scenarios* lists a classification of the environments in which, each of them, can possibly be the most appropriate solution.

2.3.1. RPC-oriented style of the SOAP Protocol

SOAP is a protocol for exchanging data between two points in a distributed system (Bragança, 2012). The nomenclature of this protocol dates back to the year 1998, originally and currently meaning Simple Object Access Protocol, however, some authors refer to it as Service Oriented Architecture Protocol (Kalin, 2009). SOAP web services expose services on a particular business model, they are considered suitable for synchronous and asynchronous processing and they also provide very mature security and integrity protocols, among other Quality of Service (QoS) features.

Although SOAP is a protocol, some authors categorize its utilization style as "Remote Procedure Call-oriented" or "Tunneling" (Jendrock et al., 2014; Oracle, 2013; Tidwell/Snell/Kulchenko, 2001). The style approached in this section is based on the following pillars:

1. There must be a RPC-like interface exposed to allow the binding of the clients;
2. The format of the messages exchanged between the client and the server are XML-centric;
3. It is transport agnostic, meaning, provided that the information is compatible and usable by the system, it does not matter how it is encoded nor transported.

Due to the fact that SOAP is considered a standard and also, to supposedly be the most renowned API protocol of the "RPC-oriented"/"Tunneling" style (API Academy, 2012), it was chosen to be the focused protocol to study this style.

The SOAP message is an XML document which essentially represents an element named *envelope*, which is itself constituted by two elements: an *header* element, which is optional, and a mandatory *body* element (Box et al., 2000). If present, the *header* element must be the first sub-element of the *envelope*. The body element is generally utilized for control purposes, such as authentication and transaction management. As for the *body* element, it is commonly used to aggregate the service's information itself or the information that is meant to be send. An example of its message structure is given in the figure 2.15.

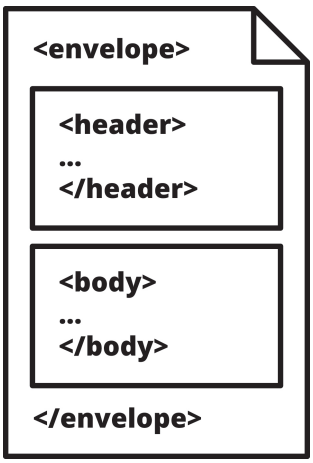

Figure 2.15.: The envelope's structure and respective sub-elements of SOAP's messages.

The SOAP protocol is utilized (often in parallel with other specifications) in fields that have strict requirements, for instance spatial information and legal areas (Villa et al., 2008; Harris et al., 2008). SOAP can also have contracts[6] with a very formal syntax and, if there is the need for a data format with very strict specifications between the consumer and service provider, the SOAP usage can be a great choice. The contract formality can be reached via the WSDL, and also, with XML Schema Definition (XSD) document imports into the WSDL itself (Ballinger et al., 2006). XSD documents aim to facilitate the structure description of a XML document and its respective constraints (Gao et al., 2012). The following listing 2.1 shows an excerpt of a sample XSD document with an "order" element, in which, the maximum number of "coffee" sub-elements is 3.

[6]Contracts in the context of web services are further detailed in the chapter 3.

Listing 2.1: Excerpt of a XSD document.

```
<xs:element name="order">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="coffee" type="xs:string"
      maxOccurs="3"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Moreover, there are three main specifications associated with SOAP, the WS-Security, WS-ReliableMessaging and WS-AtomicTransaction (among many other specifications). Although these three specifications significantly augment the level of technical complexity (Marks/ Lozano, 2000), they are still important QoS "tenets" and can lead solution architects to choose the SOAP protocol utilization instead of a REST approach. WS-Security is a specification that provides security features, integrity and confidentiality, WS-ReliableMessaging ensures that SOAP envelopes are delivered between distributed applications, even in the presence of failures in software, system or network components and, WS-AtomicTransaction establishes a condition in which, if a single atomic transaction fails, the entire transaction fails.

Regarding the SOAP protocol technical utilization, the interaction usually starts in the client, where it issues a SOAP request, in which, the client indicates the exposed WSDL operation that it intends to execute. If the request is validated, the SOAP service will process the requested operation and answer the client. Before the sending and after the receiving of requests and responses, the serialization and deserialization of data into and from XML usually occurs. The figure 2.16 briefly depicts such actions.

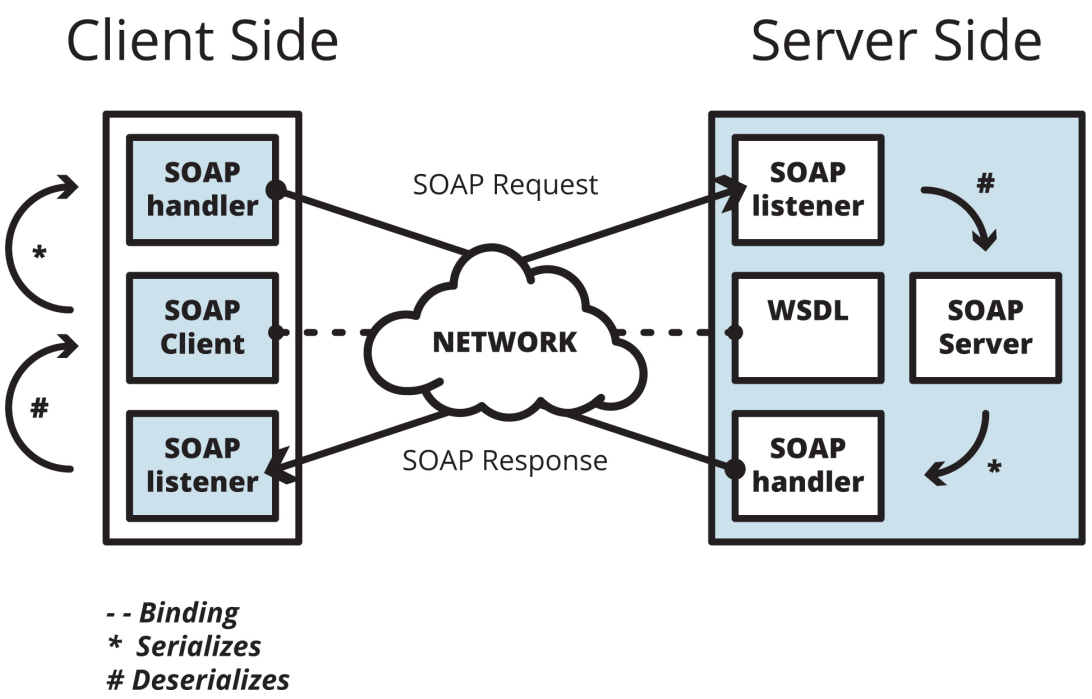

Figure 2.16.: SOAP entities interacting

2.3.2. Hypermedia REST style

This style of web services dates back to the year 2000, when it was mentioned in Dr. Roy Fielding's doctoral dissertation (Fielding, 2000a). REST is an architectural style to serve as a guide for building client-server applications and its main objective is to result in highly interoperable, resource oriented and hypermedia-driven services.

The architectural style in study at the present chapter, is not a concrete thing, is more an abstraction, which provides a set of architectural guidelines and constraints. The resources and the protocol can be considered as two of the most crucial constraints of this architectural style.

Resources

In this context, one of the constraints indicates that the application state and functionality should be abstracted in resources. In this architectural style, the clients and the servers transfer resource representations between them. These resources can be addressed as a collection of items such as "*/books*" or, as a single item of a collection, such as "*/books/the silicion web*". Some characteristics of these resources constraint are itemized bellow.

- Resources should be reached by URIs (uniquely addressable);

- Are represented by nouns, not verbs;
- Its granulation should be *coarse grained*, rather than *fine grained*;
    1. The resources should encapsulate all of its information, making them capable of reacting well to changes over time and to evolve independently;
    2. Provides use case scalability, meaning the functionality range of a particular resource can be changed over time, but its address will be the same;
    3. The client's use case itself (*e.g.* collect partial or full information from a resource), can change over time, but the target resource will still be the same.
- Uniform Interface;
    1. The usage of standard methods simplifies the system architecture, augments the visibility of interactions between components and also facilitates independent evolution of implementations and services (Fielding, 2000b);
    2. The REST specification encourages the usage of two kinds of representations, the resource's representation itself (*e.g.* an HTML), and the resource's representation metadata (*e.g.* a media type identifier).

Architecture and Protocol

Although the specification of REST does not limit the protocol to be HTTP, most of the known REST API's use HTTP as its method for transferring and exchanging data. This protocol constraint of the REST architecture is that the mentioned resources should be reachable by a protocol that allows the system to have an architecture with the following characteristics:

- Fits in the client-server model;
- Stateless;
- Cacheable;
- Layered;
- Code-On-Demand[7].

As the current Web environment is filled with larger systems and connected to more clients running on an increasingly wide variety of heterogeneous systems, the communication paradigm mentioned in the previous section 2.3.1, is not considered optimal by some authors (XyzWS, 2015; Papazoglou, 2008; Kanjilal, 2006). Some claimed facts are, for instance, the mandatory marshalling of objects for transmission, the requirement of parsers to translate the received data representations between the endpoints and, type conflicts between inter-technology data

[7] This characteristic is further detailed in the chapter 3 at the section 3.2.2 *Representation*.

primitives. As for implementations of the REST style, there are cases where they can have some of the mentioned disadvantages. However they can mitigate some of them, if each service can expose data representations to its respective clients, in a format of their choosing (*i.e.* standard format representations), and also, exchange it via communication standards, which most of the technologies support, such as HTTP, which is (and has been) considered by some authors the current *de facto* communication standard (Marshall, 2012; Banzal, 2007; Baccala, 1997). The hypermedia architectural style of REST attends to some of the stated disadvantages, and, projects the transport protocol beyond its responsibilities as it promotes the embracing of it to a level of application syntax and semantics. These can directly result in higher performance, scalability and interoperability.

REST also defines that the status of the application must be converted to a resource state or stored at the client. A server should not save the state of communication for any service client who communicates beyond a single request. When isolating the client, it will also be unaffected by changes on the server. This points out that it is suitable for cached and layered systems.

Some authors classify the emergence of the definition of this style as an important contribution, because REST resembles the architecture of the web as it works today. So if the desired system has elements such as web applications, it can seem logical for some developers that its architecture should follow its own environment's architecture (*i.e.* the web's). The architecture follows the client-server paradigm, as illustrated in the next figure 2.17.

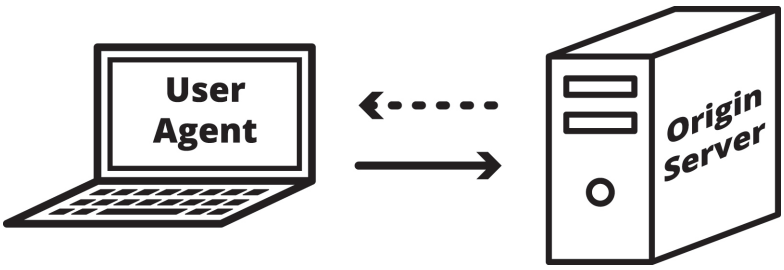

Figure 2.17.: Client-server or user agent - Origin Server architecture.

The client is usually referred as the user agent and the web server, as the origin server. There is usually a lot of intermediaries between the user agent and the origin server, such intermediaries can be considered the layered part of this architectural style. They can be placed in various points between the client and the server, without changing the interface of the service. They can provide useful support functionality such as translation or improve performance (*e.g.* caching). Each request is stateless because each request is supposed to be independent. There can be proxies and gateways between the client and server. Proxies are chosen by the client, whereas the gateways are chosen by the server (Nottingham, 2013). These intermediaries

are illustrated in the figure 2.18.

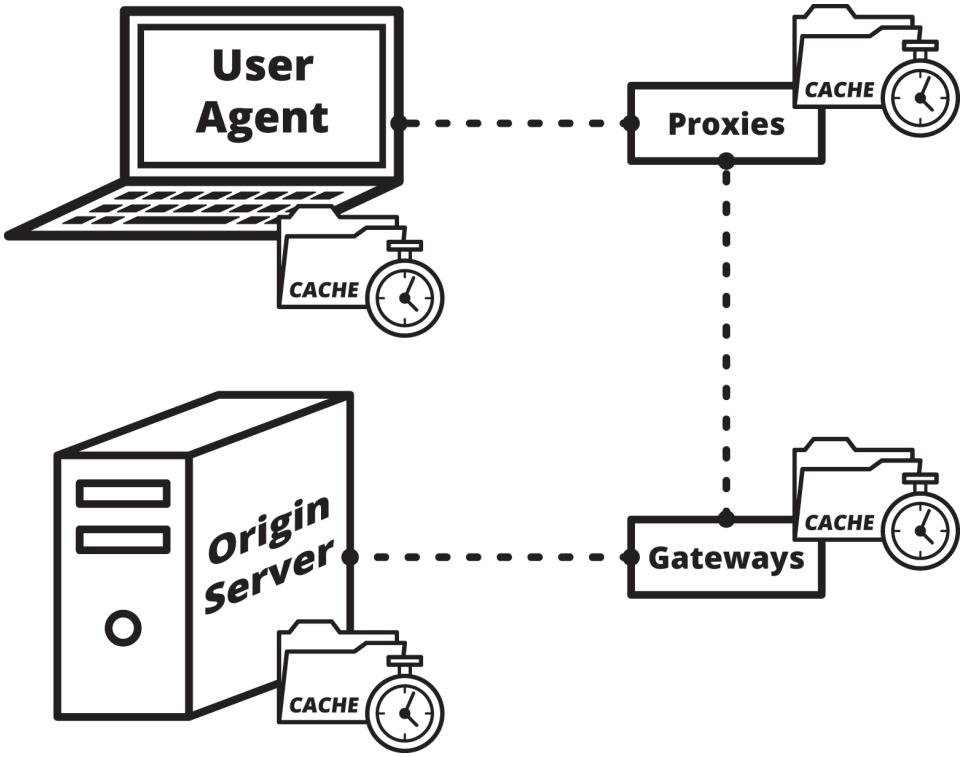

Figure 2.18.: Client-server architecture with intermediates, and, their respective caching systems.

There is a method associated with HTTP's communication messages that are exchanged between the user agent and the origin server. This method is the indicated action of the operation. Some methods indicate safety, idempotence and/or cacheability to possible intermediates between the user agent and the origin server. As illustrated in the figure 2.18, every intermediary in the chain can have a caching system associated with it. If the response has a flag indicating that the response can be cached for a certain time, then the next request for that particular resource with that particular method (*e.g.* HTTP's GET), doesn't have to go all the way from the user agent to the origin server. When an occurrence of the previously mentioned scenario happens, the resulting situation is commonly known as a *cache hit* (Ghosh, 2006) and it is represented in the figure 2.19 .

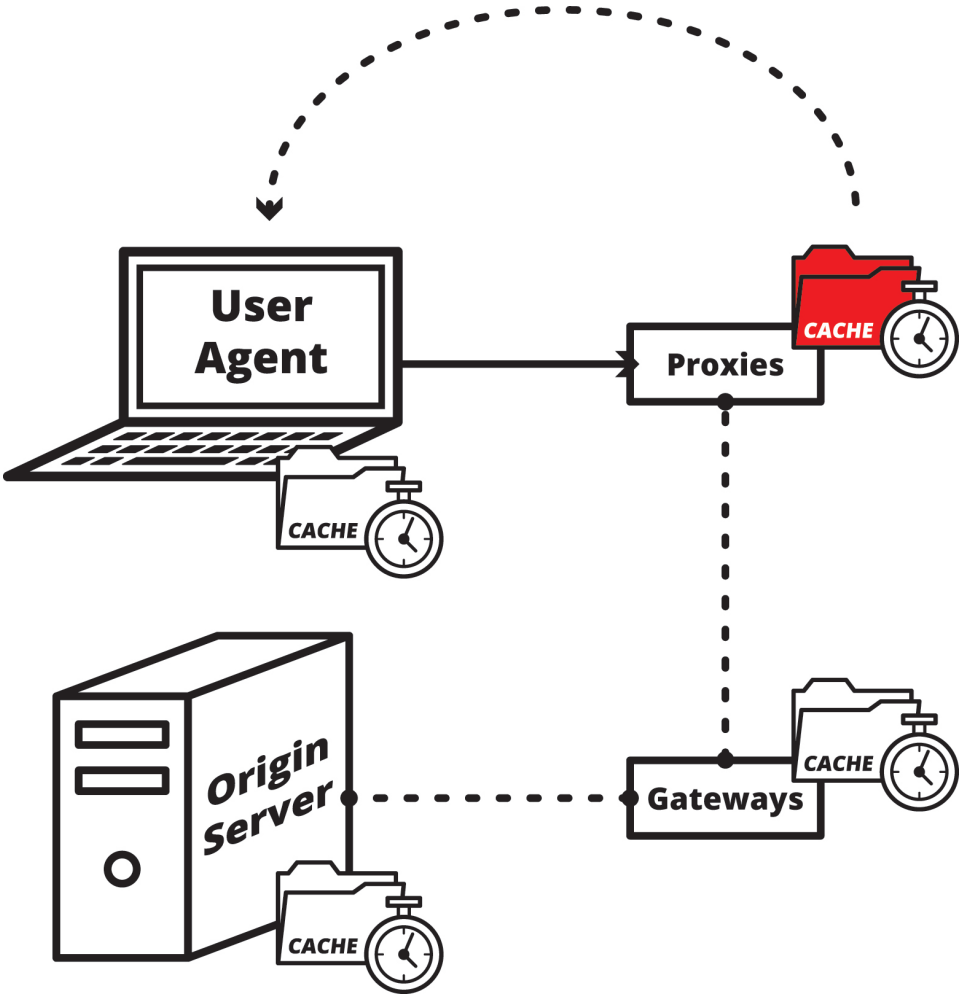

Figure 2.19.: Client-server architecture with intermediates in which, a cache hit occurs.

Some authors (Sousa, 2015a; Rodriguez, 2015; Jendrock et al., 2013; Benatallah et al., 2010) summarize the key principles of REST as:

- Exposed features such as data and functions (*i.e.* the resources) should be identified and accessible by URIs;
- Hypermedia as the Engine of Application State[8] (HATEOAS), meaning an aggregation of everything through links, among other details;
- Interfaces are confined to a uniform interface of "common" methods between entities and/or resources;
- Providence of multiple representations of the resources;
- Stateless communication.

[8]This topic is further detailed in the chapter 3, the section 3.5.

The next picture 2.20 illustrates some RESTful resource interactions (or operations), on some of the available resources and its respective states in a abstract service.

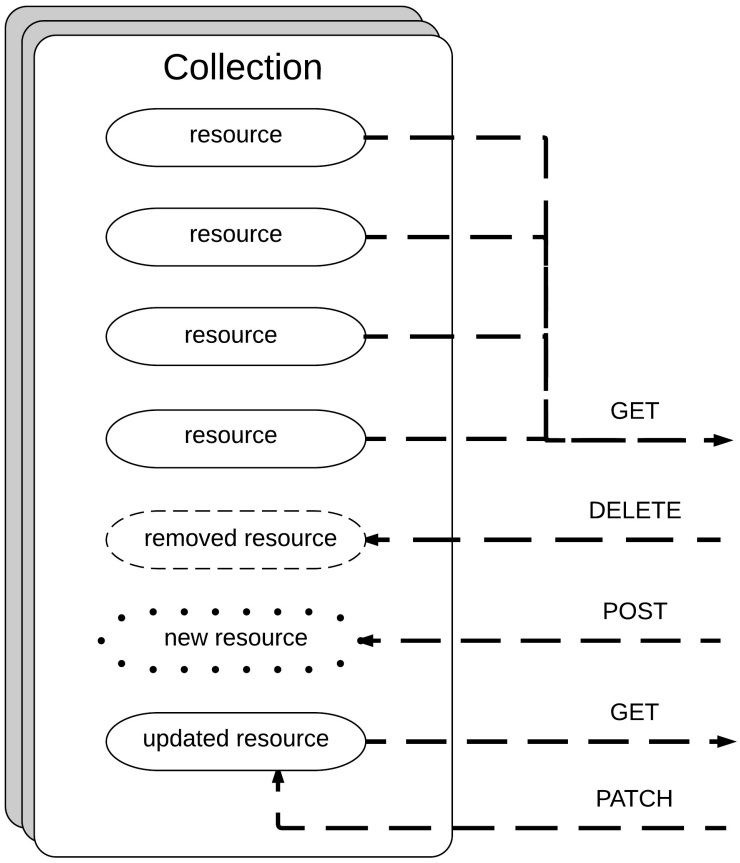

Figure 2.20.: Interaction with REST resources.

The client can request a specific resource, like a book or, a collection of those resources, like a list of books of some theme. Once a client gets a resource, it can usually update it. This is also represented in the figure 2.20, where the client updates a resource via the Patch HTTP method and adds a new item via the POST HTTP method.

2.3.3. Suitable scenarios

The present sub-section analyses the main differences between the implementation styles of the REST and SOAP web services, in order to support the inference of their most suitable scenarios. The following itemizations iterate through their respective advantages and propensities, along with their disadvantages and common issues.

The propensities and advantages of the SOAP approach are:

- It is a mature technology, with good design and well documented;

- SOAP can be used over any transport protocol, even in UDP (OASIS, 2009);
    - Can be used in application layer protocols such as HTTP and SMTP to trade and transmit messages;
- The WS-Security, WS-ReliableMessaging and WS-AtomicTransaction specifications are SOAP extensions which provide important features, specially for sensible enterprise businesses;
- The WSDL and XSD schemas (usually utilized in SOAP services), allow a very precise definition of the data, meaning, the provider/consumer contract data constraints can be accurately defined;
- Provides loose coupling for integrating distributed systems;
- Supports error messaging, with customized codes for various error types;
- Transport neutral, platform neutral and language neutral;
- Client-side artifacts can be generated easily from a WSDL;

However, it can have propensity to the following drawbacks:

- Complex contracts when compared to the REST ones and, it also requires a higher level of learning to fully take advantage of their functionalities;
- The creation of consumer components can be more complex and could take more developing time when comparing with a REST client, meaning, the simplest SOAP service's client is more complex than the simplest REST service's client;
- Complex messages can increase the load on the server and on the network, although it is not that notorious when considering the currently available network and hardware resources, the mobile limited resources should be considered;
- The interfaces are usually specific to the business context and directly represents the functionality intent. This can lead the solution to not align with the software engineering principle of generality [9];
- Limited to XML message formats;
- In implementations of earlier versions, prior to the 1.2, the only HTTP method that is allowed for binding purposes is the POST, so no safety, idempotence nor cacheability of the resource is indicated to the intermediaries. However, in the specification of SOAP version 1.2, the POST and GET methods are supported[10] (Marchal, 2004), and some others might be supported in the future (Mitra/Lafon, 2007);

[9] This principle indicates that the software solutions should be as generalized as possible to be more adaptive to changes and provide re-usability (Jackson/Anderson, 2014).

[10] The POST for SOAP Request-Response message exchange patterns and the GET for the Response message exchange patterns.

- Requests that are made with the POST method repeatedly to the same URI, can also prevent intermediaries to analyze the header and check what is in the actual operation, as it is in the body. This can also result on impeding the intermediaries of giving any additional help;
- Some XML Schema data types do not always have direct mapping to the native data types of some programming languages, for instance the date's data types (Lehmann, 2007).

Nonetheless, SOAP implementations seem the most suitable approach for operations which need the traditional session state. SOAP also has additional specifications to support a better session management as transactions and coordination, which, in a REST architecture, the solution architects would be obliged to create their own alternative tool to respond to these needs or, implement an hybrid solution. A good example of this is the WS-AtomicTransactions specification.

Concerning some SOAP common use cases, we can find them mostly on:

- Financial services;
- Payment gateways;
- Telecommunication services.

Some known SOAP APIs are the Salesforce, Paypal [11] and Clickatell SMS APIs.

As for the REST architecture approach, regardless of the intent, whether it is obtaining data or executing some server processing, REST is recommended when there are limited bandwidth and hardware resources, due to the fact that the resources representation format(s) can be cleaner and simple than the SOAP's XML (*e.g.* JSON), thus making the network payloads lighter. This representation format can be selected by the architect of the solution freely.

Another feature of the REST architecture is the existence of a generic interface, for instance, the already mentioned standard HTTP methods, which can be the "vocabulary" to be used in all resources. This referenced uniform interface, can also have a rich vocabulary as it can consist in verbs such as GET, POST, PATCH, UPDATE, DELETE and OPTIONS. This enables all the components that can use the HTTP protocol to interact with each other, in the same "language". This constraint also aligns with the software's principle of generality. As for caching situations, REST has the advantage over SOAP because, besides the GET method, the OPTIONS method can also be used in this context. In principle, these methods should not create relevant server-side changes and the results of these can be passed to the cache. There are a lot of intermediate components that can enhance the benefits of this feature like HTTP servers,

[11] Although its SOAP API is being classified as legacy and becoming substituted by their own REST API (PayPal, 2014).

Gateways and Proxies. Caching systems have been around and have been utilized for a long time (Abrams et al., 1995) and, besides improving the system's performance and the user's perceived performance, it can even be utilized to reduce costs (Hefeeda/Hsu/Mokhtarian, 2011).

To interact with the resources they receive, the clients will need to know the format of these. In a SOAP environment, there would be a problem if a potential client of a service would not work with XML. REST answers this type of conflict with "redundant" interfaces. Such interfaces are further detailed in the chapter 3 *REST, an Hypermedia approach.*

The following itemization lists some of the main implementation's perks of the REST approach:

- Based on the web architecture principles;
- Some REST services can be consumed by any client, even by an AJAX request from a client-side script of *Javascript*;
- Due to the fact that it is already widely adopted in the web community, more credible documentation, frameworks and developing tools are becoming available;
    - Amazon, Twiiter, Google, Facebook, Flickr and many other renowned entities are using it;
- Data representation formats are not limited to XML in any way, other formats such as JSON and CSV can be utilized;
- When building clients (and still comparing with SOAP), the RESTful approach usually requires a simpler learning level and less development time.

None the less, there are some handicaps that this approach can present:

- Assumed model of communication is point to point, not point-to-multipoint nor broadcast;
- Implementing security in a REST system can be a big challenge when developing the solution. HTTPS can provide transport-level security, however, there are not many documentation concerning the message-level security;
    - Some tools have become available, such as the security modules of the *Java's* framework *Jersey* (Oracle Corporation, 2015), which gives a strong support to the authorization factor, among others;
- When the interactions must be initiated by the server, the technical details can be cumbersome;
- A REST API architecture seems more difficult to design and implement than a SOAP API;

- As not all formats can express (or doesn't have as its main purpose) data type constraints (*e.g.* JSON), it should be pondered if this approach is suitable when there is the need to ensure strict message formats between the clients and the service, when exposing it in multiple formats;
- When coordinating transactions need context on the server, REST can appear as not suitable when this is a major necessity, specially when it involves multiple calls;
    - Some techniques (*e.g. callbacks*) that can solve this problem are presented in the chapter 4;
- Its specification is not considered a standard, however, it can be considered an approach of an interpretation of how the web works.

Concluding, the REST style appears to be suitable in scenarios when:

- The service and its clients are present on a web environment;
- The client can request partial information of the resources;
    - Some examples of this condition are presented in the chapter 3;
- There is the need of a wide scale deployment;
- The API is targeted to be consumed by mobile applications or other systems with limited resources.

REST and other hypermedia-based styles of services can be easily found in use cases such as:

- Social Media services;
- Social Networks;
- Web Chat services;
- Mobile Services.

Some practical examples of use cases of the Hypermedia APIs are the Twitter, Slack [12] and LinkedIn APIs.

By placing the target project scope and its respective use cases in the strengths and weaknesses trends of the presented styles, the decision of the which kind of architecture should be implemented can be helped and/or achieved. The following figure 2.21 summarizes the propensities of each approach.

[12] Although its resources resemble RPC invocations (Slack, 2015).

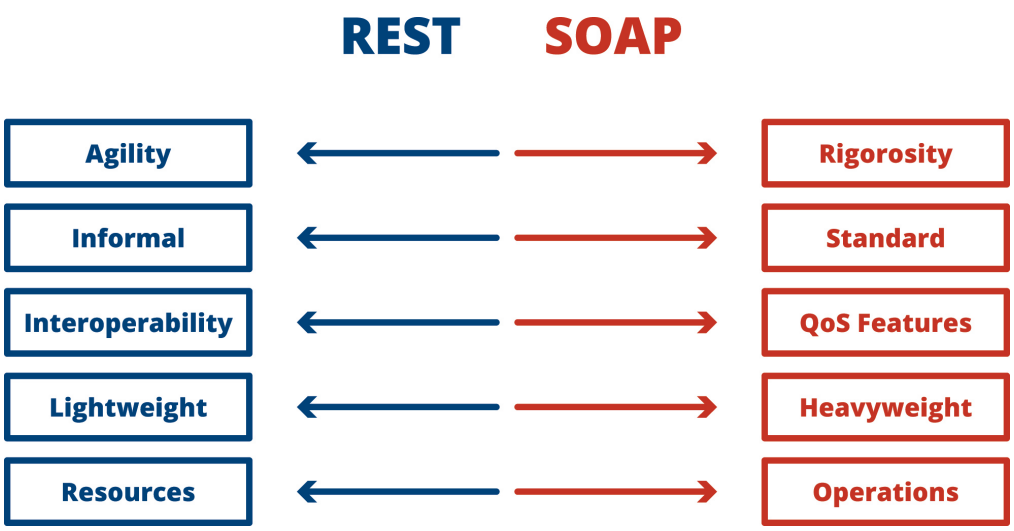

Figure 2.21.: Summarized approach propensities.

On account of the fact that most of the times there are available workarounds to achieve additional goals (even if they are not as direct as they were supposed to), any implementation style could probably be applied in any scenario. There are probably some cases that an hybrid approach could fit better.

2.4. Research question status

Most of the sections of the present chapter, partially and/or fully answered some of the research sub-questions enumerated in the section 1.3. The following itemization summarizes those answers:

- **RSQ1** *Why there is the need to combine information systems?*
  - In the sections 2.1.3 and 2.2.3, some of the needs for those integrations were given (*e.g.* the achievement of top-notch features);
- **RSQ2** *Which environments usually surround integration solutions?*
  - The answer to this question was obtained in the sections 2.1.2, 2.1.3, 2.1.4 and 2.1.5, where some deployment scenarios were illustrated and, some commonly found elements of those environments were exemplified (*e.g.* databases and application servers);
- **RSQ3** *Which models and architectures are utilized to solve system integrations?*
  - The sections 2.1.1, 2.1.3, 2.1.4 and 2.1.5 gave answers to this question and highlighted some of the utilized models (*e.g.* the client-server model) and architectures (*e.g.* SOA);
- **RSQ4** *Which technologies and techniques can be applied in architectures found in RSQ3?*

- The answer to this questions was partially given in the sections 2.2.1, 2.2.2 and 2.2.3 (*e.g.* web services);

- **RSQ5** *Which characteristics should have the desired system?*
  - Some of those characteristics were mentioned in the sections 2.2.3 and 2.3 (*e.g.* interoperable and scalable);

- **RSQ6** *Which metrics can be utilized to classify the final solution as viable?*
  - This question was partially answered in the section 2.3.3, where some metrics were mentioned (*e.g.* network performance and amount of development effort);
  - The approach of these metrics was classified as being qualitative or similar-to-qualitative and the specific mentioned ones appear to be very subjective, imprecise and dependent on the specific context, for instance the security level has various degrees of satisfaction dependently on the business specific necessities and on the deployment scenario;

Chapter 3

REST, an Hypermedia approach

In this chapter, the REST architectural style introduction is extended and detailed. As this chapter is intended to be technical, some examples of its characteristics are exposed via diagrams and source code.

3.1. Contract description and documentation

There can be found a lot of human-only processable textual documentation while searching public APIs, like on web pages or readme-like PDFs, however, and as mentioned in the section 2.2.3, WSDLs and WADLs can be utilized to improve the service description, due to the fact that both of them are machine-processable.

A Web Service Descriptor (WSD) formatted in the WSDL specification format, was the first type of a machine-processable web service descriptor there was [1]. The WSDs describe an API contract (*i.e.* it describes the messages that the service is prepared to receive and send to some potential service consumer). Any web service, regardless of the type of system communication, needs a service descriptor to expose the contract provided by the service and, to better solve this, it can be used a document and/or web page to further detail the service in a more human-readable format, alongside with the WSDL. These descriptors detail which features are offered by the service, where they are exposed and how they can be utilized. In practice, this means it defines the suppliers and customers behavior and the messages they exchange. The publication of the descriptor is the responsibility of the supplier, and the analysis and discovery is the responsibility of the consumer. The figure 3.1 illustrates the WSDL definition and these two participating entities.

[1] IBM and Microsoft combined some service description languages into one, NASSL (Network Application Service Specification Language, SCL (Substation Configuration description Language) and SDL (Service Description Language) (Christensen et al., 2000) .

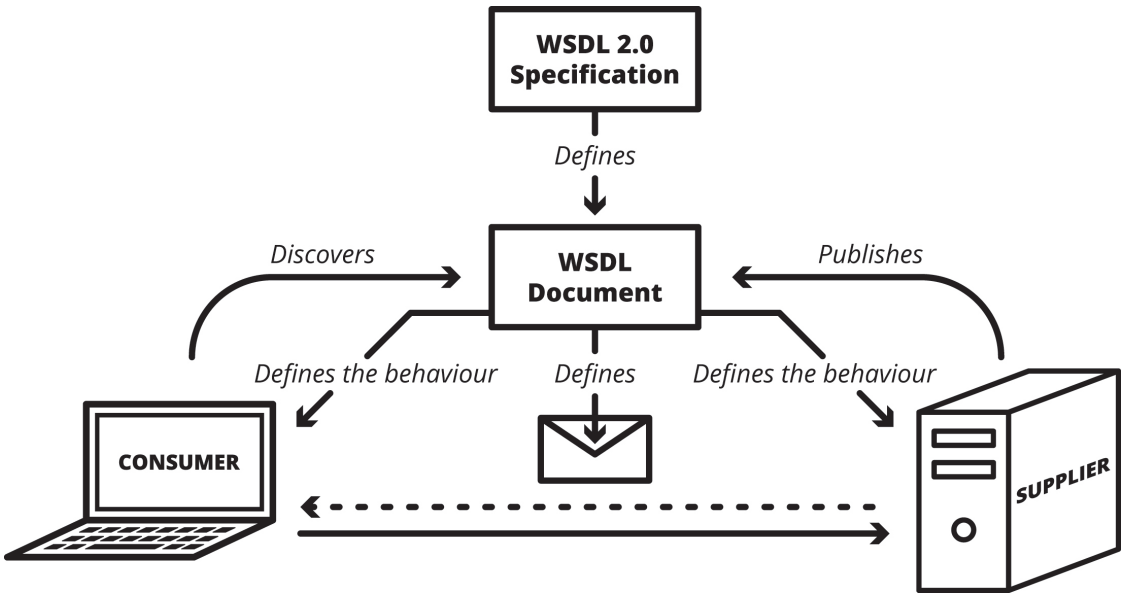

Figure 3.1.: The role of WSDL and of the participating entities.

The WSDL document is composed by some elements, each one has its own purpose. The listing 3.1 represents the skeleton of a WSDL 2.0 document and an enumeration its respective elements.

Listing 3.1: The structure of a WSDL 2.0 document.

```
<wsdl: description xmlns: wsdl = "http://www.w3.org/ns/wsdl">
<wsdl: types />
<wsdl: interface />
<wsdl: binding />
<wsdl: service />
</wsdl: description>
```

Only the element "*type*" is mandatory to actually describe the service, however, all the elements can be conjugated to offer a more rich service description besides the data schemas. The following table 3.1 sumarizes the WSDL elements and their respective purpose.

Table 3.1.: WSDL elements

Element	Necessity	Description
Interfaces	Optional	Describes the service features such as input and output
Binding	Optional	Combines the web service address with a specific binding and interface
Services	Optional	Defines how a client can interact with the service. In a REST scenario, HTTP is specified
Type	Mandatory	Data description

There are four types of elements that play a role in the web services paradigm, which are commonly referred as Agents, Services, Providers and Requesters. The Agents are software

or hardware components whose only purpose is to send and receive messages, whereas the Services are the supplied functionalities. Therefore it is possible to have one or more Agents invoking one or more Services. The Providers are the entities that supply Agents, whose purpose is to answer requests to a specific service. The Requesters are the entities that own one or more Agents that utilizes one or more Provider Agents. The figure 3.2 illustrates the four elements found in the web services paradigm.

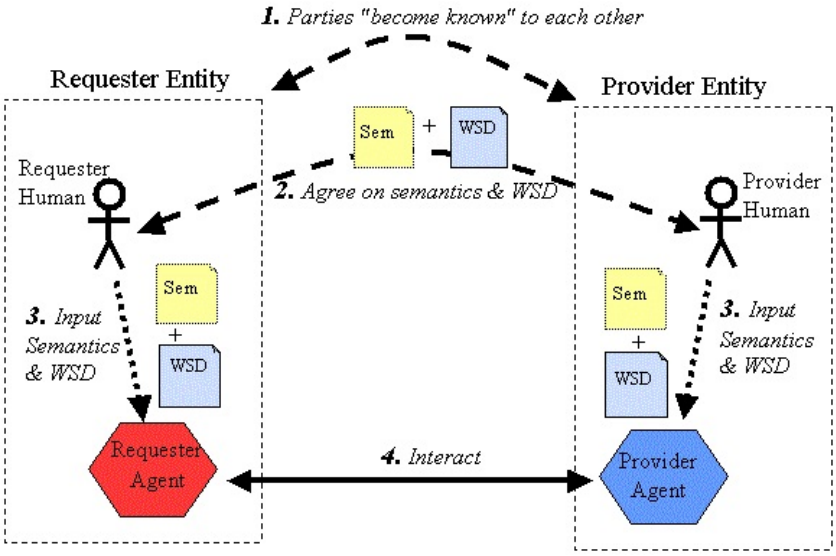

Figure 3.2.: *"The General Process of Engaging a Web Service"* Booth et al. (2004)

Although the most common scenario is to the Requesters to initiate the communication, in some cases the Providers are the ones that actually trigger the Requester-Provider interaction, relating to the publish-subscribe model, where the publisher's message broker forwards the message to the subscribers.

As for WSDL, it is currently in the version 2.0 and it is to note that WSDL 1.1, was not suitable to support REST on account of the specification that a PortType (*Interface* in the WSDL 2.0 terms) could access four different resources but with only one HTTP method Apache Software Foundation (2014), also, since REST services usually expose more than a GET and POST methods on the same resource (*e.g.* PUT), the WSDL 1.1's *binding* element was inadequate, as it only accepted GET and POST. However, even WSDL 2.0 has some characteristics that lead some authors and developers to find it hampering for its utilization in REST services, such as its complexity and its bigger distance from the web architecture when compared with WADL (Rodriguez, 2015). Notwithstanding, the current version does not have most of the previously mentioned impediments of the earlier versions.

Continuing in the service descriptors topic, there is the WADL document. WADLs are designed to provide a description of web applications based on HTTP and, similarly to the WSDLs, they are meant to be processed by machines (Hadley, 2009a). They are designed to be used by web applications that can handle content whose representation semantics are clear (*e.g.* XML or JSON) and, which generally are:

1. Available to integrate other applications;
2. Meant to be re-usable from other origins besides the browser;
3. Fully unbundled from the platform and the programming language.

So an application can integrate a web application via its WADL, the latter resulting document will have to be composed by the provided application resources, their relationships (described by links), the allowed HTTP methods that can be applied to each of the resources, their expected inputs and outputs, the utilized data schemas and the supplied types of the representations (*i.e.* the Internet media types [2] also know as MIME and Content-types).

It is to note that WADL and WSDLs share one common fact: there are some available frameworks which generate clients automatically based on the WADL/WSDL document. However, when the WADL or the WSDL itself is generated automatically, some problems can rise with some data types, due to the lack of data type correspondence between the WADL/WSDL generator tool and the respective client generator tool, for instance the date's data type is prone to generate conflicts on account of the distinct pattern that its format can follow between the programming languages and/or frameworks.

WADL provides the URIs operations, resources and formats that are generated by the services. The WADL use cases that are present in its specification are (Hadley, 2009b):

- *"Application Modeling and Visualization"*
  - Grant support for development tools to model, manipulate and interact with the resources and their relationships;
- *"Code Generation"*
  - Allow the generation of code templates and the generation of resource representation handlers;
- *"Configuration"*
  - Provision of portable settings for the server and client.

[2] Internet media types are identifiers of the type of the data that a file contains, they were originally defined in the RFC 2046 and were referred as MIME types (Freed/Borenstein, 1996), while describing the Multipurpose Internet Mail Extensions (MIME), a specification of a non-ASCII component of a email message.

The listing 3.2 represents a simple skeleton of WADL document.

Listing 3.2: WADL document

```
<application xmlns="http://wadl.dev.java.net/2009/02">
    <resources base="http://bookSearchApi.com/api">
        <resource path="themes">
            <method name="GET"/>
            <method name="POST"/>
                <resource path="{themeId}">
                    <param required="true"
                     style="template"
                     name="themeId"
                    />
                    <method name="GET">
                        <param name="size"
                                required="false"
                                default="5"
                                style="query"/>
                    </method>
                </resource>
        </resource>
    </resources>
</application>
```

In the listing 3.2 , it can be observed that the presented sample WADL indicates that there is a resource named *themes* that exposes interactions with two HTTP methods, the GET and the POST. It can be also observed that inside the *themes* resource (that indicates a possible list or collection of theme items), can be queried a resource for a particular theme or, an item of the themes collection resource. This particular item can be retrieved also, via the GET method and it is highlighted that there is a required parameter for this particular sub-resource, the *size* of the results. In the context of the listing, an item of the collection themes is also a collection itself (a list of books of a particular theme), which, if this example would be extended, the sub-resources under the resource represented by the attribute *path="themeId"*, could be a single item, in this case, a book.

The listing 3.2 also presents two kinds of parameters, one with the *style="template"* attribute, and other with the *style="query"* attribute. The parameter configured with the style defined as ”template”, indicates that the parameter is a segment of the URI and does not make part of the

query string[3], whereas the latter parameter's configuration with the style defined as "query", indicates the opposite of the former (*i.e.* that the request is present in the query string). An example of a request to the API that holds the service description presented in the listing 3.2 is given in the listing 3.3.

Listing 3.3: WADL document consumption

```
1 GET /themes/Science?size=3 HTTP/1.1
2 Host: bookSearchApi.com/api
3 Accept-Encoding: gzip
4 User-Agent: Python-httplib2
```

It can be concluded that, although the WSDL and WADL formats have very similar objectives (*i.e.* the service description), they differ in the fact that WSDL is a technique more focused in detailing the messages and operations via the consumption and the production of them, whereas the WADL technique is more oriented to describe the hierarchical relationship of the resources, although it can still indicate the required parameters and the allowed operations.

3.2. Resources

Resources are identified by a URI. The URI is divided in two sections, the HOST and the PATH. In the following listing 3.4, the PATH is indicated in the first line, and the HOST is in the second line.

Listing 3.4: URI sections in a HTTP request excerpt.

```
1 GET /xptoResource HTTP/1.1
2 Host: www.example.org
3 Accept-Encoding: gzip
4 User-Agent: Python-httplib2
```

3.2.1. Identification and distribution

The resource items can be categorized as collections or as actual single items (*e.g.* *Orders* and *Order*). Depending on the actual business logic, they can be defined in a hierarchical arrangement or, be placed at the same level. Such organizations are depicted in the listings 3.5 and 3.6.

[3] Query string is a set of words that is separated from the URI by a question dot (?), which is utilized to refine the contents of the results (Berners-Lee/Fielding/Masinter, 2005).

Listing 3.5: Items distributed in a hierarchical arrangement.

```
www.example.com/orders
www.example.com/orders/order
```

Listing 3.6: Items distributed at the same level.

```
www.example.com/orders
www.example.com/order
```

The arrangement itself should not dictate how the clients should operate or interact with the service, but it should follow the actual business logic.

To identify and distribute the resources in a RESTful manner, the URIs must expose a perceivable understanding of what the represented resource is and/or does. By avoiding verbs and selecting nouns in the URI skeleton, the former style of representation can be accomplished. URIs with verbs are considered to be RPC style, whereas URIs that do no have verbs in it, are considered to be RESTful. The notion of what can be done to a resource or, of what the resource does, can be achieved by the HTTP methods that the resource exposes and by its actual name. The following listings on the left side, depict RPC-like interface requests, whereas the listings on the right side, depict RESTful ones.

Listing 3.7: Update RPC-like URI

```
POST /update-user?name=jack
Host: www.rpc-like.com
Content: application/xml
<user>
 <name>jack</name>
 <age>25</age>
</user>
```

Listing 3.8: Update RESTful URI

```
PATCH /users/jack
Host: www.restful.com
Content: application/xml
<user>
 <age>25</age>
</user>
```

Listing 3.9: GET RPC-like URI

```
GET /orders/getList?id=1
Host: www.rpc-like.com
```

Listing 3.10: GET RESTful URI

```
GET /orders/1
Host: www.restful.com
```

3.2.2. Representation

REST allows the consumer to choose the display format it prefers and its preferential order, this can be done in the HTTP header *Accept* property or, in the query string itself, although the latter implementation technique would be redundant since the former was already established for this purpose.

In the eventuality of the header *Accept* property not being defined by the requester, the assumed value will be *, meaning all media types are accepted by the client.

In a scenario where the request header's *Accept* property value is not provided by the server, it should send a response to the client indicating that it is not acceptable, *406 Not Acceptable*. This standard response codes are further detailed in the section 3.3 *Methods and responses*.

The header's *Accept* property value can consist in more than one media type, and it can also indicate preferences between them. This can be achieved by the specification of the following rules:

- The preference value ranges from 0 to 1, where 1 is the highest (100%);
- If a media type preference is specified with the 'q = X', then the preference will be represented in X value;
- If a media type preference is not quantified (by the parameter "q") , the default is assumed, which is 1;
- The media types should be separated by commas;
- There should be a semicolon between the media type and the respective "q" parameter;
- The preferences should be sorted by descendant preferential order.

Due to the fact that the "q" parameter was chosen to quantify the preference of the media type, it is not advised to register "q" as a IANA media type (Fielding et al., 2004a) , as it could hamper the interpretation of the header *Accept* property. It is to note, that the "q" parameter is also utilized to quantify the preference of other header properties, such as the *Accept-Charset* and the *Accept-Encoding*.

It can be considered that each URI has redundant interfaces associated with each resource, as each resource can be represented, in the same URI, in various media types. All the HTTP methods that are allowed in the resource should be able to independently comply to any with the provided media types. To accomplish this, the serialization of the resource to the requested media type should be decoupled from the actual business logic. The listing 3.11 represents some examples of multiple resource representation requests to these interfaces.

Listing 3.11: Example of multiple resource representation requests.

```
1  GET /xptoResource/1234
2  HTTP/1.1
3  Host:xpto.com
4  Accept: text/html
5
6  GET /xptoResource/1234
7  HTTP/1.1
8  Host:xpto.com
9  Accept: text/html+application/xml
10
11 GET /xptoResource/1234
12 HTTP 1.1
13 Host: xpto.com
14 Accept: text/html, application/xml; q = 0.9, */*; q = 0.8
```

In the lines 11, 12, 13 and 14 of the last example present in the listing 3.11, it is indicated that the client has preference for:

- HTML, as it is not defined q = X, the highest is assumed;
- XML is the second preference, flagged in the q parameter with the value 90%;
- Other formats (*/*), are accepted and categorized by 80% of preference.

As for the actual consumption, there are some programming languages whose implementations cannot easily handle XML, like *Javascript* that has much more support and flexibility with JSON (among other issues, for instance compatibility and requisites related with the utilization of standard formats such as CSV). Therefore, multiple formats should be provided by the services, which implies that the architecture of the application should be relatively agile to provide representation broadening over time. A service with features such as this, is also more open to use representations standards that don't use XML as its representation language, such as *vCard*, which is an accepted standard format specification to describe individuals and other entities (Perreault, 2011).

Besides the format preference indication, the actual content of the format can be also specified (*i.e.* the detail's depth), a good example with a service with this capability is the Facebook's graph API. Some URIs of it are given as an example in the listing 3.12.

Listing 3.12: URIs exposed in the Facebook's API Facebook, Inc. (2015)

```
1 https://graph.facebook.com/youtube
2
3 https://graph.facebook.com/youtube?fields=id,name,likes
```

In the line 1 of the listing 3.12, the "*youtube*" segment of the URI represents the whole resource. However, the requests can be refined, such as represented in the line 3. This simplifies the request and, can also improve the performance by reducing the network's payload and the actual server-side process weight and duration, due to the fact that the service is asked for fewer information. These types of implementations can be classified as request-refining techniques.

Some APIs can also provide compatibility with *Javascript* consumption, this relates to the *JSON with Padding* (JSONP) keyword and, with the already mentioned "Code-on-Demand" REST constraint (Fielding, 2000c). Resources that have representations with this capabilities differ from the others due to the fact that besides their actual information, they can also provide logic execution to clients that, in this case, are able to handle *Javascript*.

What it actually allows, is that the clients can request for *Javascript* code and add it to their client-side pages. Usually this happens by pointing the *source* attribute of a script to a web service URL. Usually when requesting it, and after a possible preflight negotiation[4], the client also includes a callback function to handle the response. An example of its utilization in illustrated in the listing 3.13.

[4] "Preflighted" requests are a procedure where the client firstly sends an HTTP request with the OPTIONS method to a resource established in another domain, in order to evaluate if the requests are safe to send. Since a request can have requisites such as authentication or other user data, these requests are usual in inter-domain requests.

Listing 3.13: Example of JSONP utilization

```
<!DOCTYPE html>
<html>
  <head>
  </head>
  <body>
    <script>
        function myFunction(data){
                // parse and handle the response;
        }
        var script = document.createElement('script');
        script.src =
        'bookSearch.com/api/theme/jsonp?callback=myFunction';
        document.head.appendChild(script);
    </script>
  </body>
</html>
```

JSONP consumption implies that the client must not only execute *Javascript* but it should also be cagey about what it is consuming, due to the fact that currently there is no easy way to control or modify the HTTP headers sent when requesting *Javascript* and also, it can be used as a malicious code injection point (Hafif, 2014).

3.3. Methods and responses

Generally in most contexts such as in SOAP or CORBA, to obtain or add information to a resource, it is utilized a specific interface for such intent. The listings 3.14 and 3.15 depict two examples of it.

Listing 3.14: Operations present in a excerpt of a WSDL.

```
<wsdl:portType name="BookResourcePortType">
    <wsdl:operation name="addBook">
      <wsdl:input name="bookInput" message="ns:bookInput"/>
      <wsdl:output name="bookOutput" message="ns:bookOutput"/>
    </wsdl:operation>
    <wsdl:operation name="removeBook">
      <wsdl:input name="bookInput" message="ns:bookInput"/>
      <wsdl:output name="bookOutput" message="ns:bookOutput"/>
    </wsdl:operation>
</wsdl:portType>
```

Listing 3.15: Operations present in a OMG IDL.

```
module Book
{
  interface BookUI
  {
  string addBook( in string bookInput );
  string removeBook( in string bookInput );
  };
};
```

Despite their different contexts and applicabilities, the WSDL and the Object Management Group Interface Definition Language (OMG IDL) present a similar structure. They both enumerate business objects (a ”book” object in the 3.14 and 3.15 listings), and several methods (or operations) associated with each of them. These exposed methods generally have a name that describes the actual processing request, like adding or removing something. In a RESTful resource interface, such methodology is not ”acceptable” my many authors.

In a REST environment, one can obtain and manipulate information by using standard methods (*e.g.* the HTTP's methods). It is only exposed a generic interface to be used for all resources. If this interface is defined by the HTTP methods, this uniform interface allows all components that can use the HTTP application protocol to interact with the system. Although the tendency of associating the HTTP standard methods to a CRUD operations, REST aims to expose more than CRUD operations of a resource. The OPTIONS method, will "question" which of the HTTP methods can be used in certain resource. Note that OPTIONS is utilised for determining the options and/or requirements associated with a resource, as well as server capacity without leading resource utilization. These types of informations are considered very suitable

for caching.

As to the responses, HTTP brings another perk through the possible and recommended usage of the default HTTP return codes (FIELDING et al., 2004b) . This response codes are widely known by Web developers and are linked to the success or failure of a particular request. The return codes are divided by groups, each group represent a category of the response, as shown in the following table 3.2 .

Table 3.2.: HTTP response codes.

Code range	Abstract Category
1xx	Informational
2xx	Success
3xx	Redirection
4xx	Client Error
5xx	Server Error

The listings 3.16 and 3.17 represent hypothetical responses to the request represented in the 3.8 listing.

Listing 3.16: 200 OK status response code.

```
HTTP/1.1 200 OK
Content-Type: text/html;charset=utf-8
Content-Length: 13

User <Jack> Updated!
```

Listing 3.17: 404 Not Found status response code.

```
HTTP/1.1 404 Not Found
Content-Type: text/html;charset=utf-8
Content-Length: 13

User <Jack> not found!!
```

Of course the responses usually are far more richer in information and metadata than the ones shown in the 3.7 and 3.8 listings, however, the response's endowment topic is more thoroughly detailed in the section 3.5 *HATEOAS*. In the present sub-section, the focus is on the actual semantics of the available status response codes.

In a RESTful environment, the HTTP response codes can be placed and used in the service's business logic itself. They possess very specific and different symbolic values, even between

the ones inside the same category. The codes are returned, after a HTTP request is done with a combination of a certain method, resource and some optional parameters. These requests can have different results and, if the service gives more clues about what happened besides the success or failure of the request, the services themselves become more perceivable and transparent. The same HTTP response status code can be applied in more than one resource response, for example, two completely different resources can return the status code 405 - Method Not Allowed if a POST is requested on them. Also, there is usually some description accompanying the status code in the response body, such as "*POST is not allowed, please use the PUT method*".

In the following paragraphs, the mentioned semantical differences are briefly detailed via an example where the *Redirection's* 3xx status code category is applied in two possible responses. The 3.18 listing represents a request where the client explicitly requests for the resource collection that represents books categorized in the *Science* theme. For the example's sake, it should be considered an hypothetical service, where the client can search for books, in a book theme collection.

Listing 3.18: GET request to a Book search API

```
GET api/v1/themes/Science
HTTP/1.1
Host: bookSearchApi.com
Accept: text/html
```

In a scenario where the *Science* theme has been grouped with the *Technology* theme, at the *Science and Technology* resource, a suitable response would be the 301, informing the request participants that the resource has permanently moved to the *Science and Technology* resource's URI. In this scenario, the requester agent can index or register this new URI and clear or update its respective entries, so that in the next requests, the old URI is not called anymore. This 301 status code can even be utilized in other issues, such as in a situation where the API has been moved to other domain or if the resource tree becomes redesigned or re-structured.

In another scenario where the *Science* theme still exists, but the service's main API (represented by the "*v1*" in the 3.18 listing) is temporarily on maintenance, the service normally would want to inform the request participants that the requested resource still exists but is currently only available at other URI. In this scenario the 302 response code would be suitable. This status code can be also utilized in situations where the main interface is "overcrowded" with requests and there is a policy implemented to redirect the client to a secondary one. The request participants can follow the redirection and ignore its URI, in a manner that in the next request, the original URI is utilized by preference.

Concluding this example, it can be noted that the different status codes can be applied in a variety of contexts and, can give more accurate responses to the requesting client developers, support teams, applications, intermediates such as proxies and search engines and, to the actual end users.

In the section 2.3.3 of the chapter 2, it was mentioned that the utilization of the REST architectural style can benefit scalability and performance. These benefits are some of the contributions of the self-descriptive characteristics of RESTful messages. These characteristics are briefly detailed in the section Self-descriptive messages 3.4.

3.4. Self-descriptive messages

After establishing the supported media types by the service and its respective allowed operations, the actual structure of the message becomes quite self-descriptive, but it can be improved even more . This self-description allow intermediates to help in the client-server interaction, which can result in a very scalable and cacheable final system. Cacheability can reduce access times, reduce latency and improve input/output (I/O) operations resulting in a system with superior performance. In a REST environment the cacheability can be provided or improved by some elements present in the HTTP's request header.

One of the elements is the already mentioned HTTP methods, which in the case of it being the GET or the OPTIONS method, usually indicates the interaction's safety and its idempotence nature, and these conditions directly relate to cacheability to the intermediates. One other element is the *Content-type* header field, which can be utilized by intermediates which are capable of recognizing it, and then practice compressing endeavors, such as downsampling[5] image or audio content. In a mobile environment, this last element's cost of implementation can prove itself very easily.

The *ETag* property can also enrich the message itself, by indicating the current "version" of a given resource. Although the use of this property in the HTTP header is optional, this property can be utilized to improve the system's performance, by preventing unneeded network payloads and server processing. This can also be combined with the *last-modified* property, which should indicate when the content of a resource has been changed. These two properties, can be utilized to differentiate the already brought data with the one currently residing on the server. An example of these two properties is given in the listing 3.19, extracted from the section 14, of the *RFC2616* (Fielding et al., 2004a) .

[5]Downsample is the practice of decreasing the bit depth of a digital image or the lowering of the bit-rate (bits per sample) of a digital audio signal.

Listing 3.19: The *ETag* and the *Last-Modified* header properties.

```
1 ETag: "xyzzy"
2 Last-Modified: Tue, 15 Nov 1994 12:45:26 GMT
```

The ETag property value, should be a string generated by the server[6] each time a given URI resource's content is updated. This string should always be different within the same resource representation, so it can be utilized to distinguish two versions of it. However, it should not create entropy if a given ETag is the same in two or more different resources. One known flaw of the usage of the *last-modified* property alone, as observable in the line 2 of the listing 3.19, is that it does not indicate differences if the resource's content has been updated multiple times within the same second.

Other elements that directly relate to cache, are some optional fields that make the actual operation conditional (Fielding et al., 2004a). These are constituted by *If-Match*, *If-Modified-Since*, *If-Range* and *If-Unmodified-Since*. These are headers that are defined in the user agent and combined with the ETag value. In the case of the *If-Match* property, the server should only proceed to extra processing if the values do not match, however, if they match, the server should respond with the HTTP status code 304, "Not Modified".

The actual generation of the *ETag* string value on the server-side, was not specified in the HTTP specification (Fielding/Reschke, 2014), as it is an implementation detail.

It is also important to note that some web engines, such as web page monitoring systems utilize the *ETag* values to check if the content of a resource (*e.g* a web page) has been updated. If these *ETag* headers are not present in the resource, this implies that both the analyzed and the analyzer servers must use computing resources in order to retrieve and evaluate the contents.

3.5. HATEOAS

As mentioned previously, the REST architectural style is defined by some constraints, and one of them is the usage of Hypermedia As The Engine Of Application State (HATEOAS) (Fielding, 2000a). This constraint means that the representations should be enriched with hypermedia links, to allow the clients to find their way and navigate through the API. Further more, these links can be categorized by its significance or its target's destination symbolism. The HATEOAS constraint also relates to the Self-descriptive messages characteristic mentioned in the section 3.4 of the present chapter.

To approach this topic, a brief example of it is given with a merely illustrative business logic. The listing 3.20 exemplifies a school's course grade resource representation, with some HATEOAS characteristics and the figure 3.3 illustrates an excerpt of its respective business logic.

[6] There can not be found yet a defined standard for ETag generation in the HTTP specification.

Listing 3.20: Example of a school's course grade resource representation.

```
GET /grades/science
HTTP/1.1
Host: school.com
Accept: application/xml

<?xml version="1.0"?>
<grade>
   <link rel="self" href="/grades/science/A1"
                    type="application/xml" />
   <id>A1</id>
   <type>essay</type>
   <value>15</value>
   <link rel="up"   href="/grades/science"
                    type="application/xml" />
   <link rel="next" href="/grades/science/B2"
                    type="application/xml"  />
   <link rel="prev" href="/grades/science/C3"
                    type="application/xml" />
</grade>
```

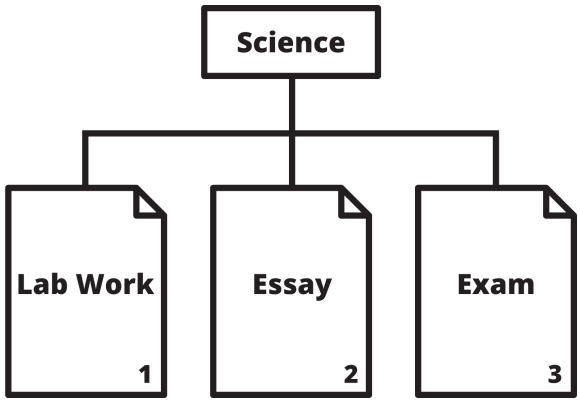

Figure 3.3.: School's course grade resource hierarchy.

The exemplified link *rel* attributes present in the listing 3.20 (*self*,*up*,*next* and *prev)* belong to a list of standard relationship types and they should do what they are meant to and behave as expected and documented in their specification (Nottingham/Reschke/Algermissen, 2015). These types of links are further divided in two categories (or H-Factor categories), depending on the type of their hypermedia support and functionality: the Link Support and the Control Data Support.

These mentioned links enrich the resource representation, by indicating its position relatively to other resources. If it is considered that the business logic is the one represented in the figure 3.3, the third link with the attribute value *"next"* in the listing 3.20, could be improved to *"last"*, as the next resource of the sequence (the exam) would be the last grade evaluation factor of the Science course. There is also the possibility of leaving the *"next"* link and add the *"last"* targeting the same resource representation.

Despite the existence of the list of standard relationship types, the semantics of the *"rel"* attribute value can indicate more than resources relationships, it can also point out custom available actions (Sousa, 2015a). If the business logic allowed a student to survey or, to see what he did right and what he failed in course evaluation, a link with a custom *"rel"* attribute value could be added to that particular resource, for instance:

<link rel=" school.com/api/survey" href="/grades/science/A1" /> .

When encountering with a custom *rel* attribute value in a link, the client must know how to handle it. Therefore, the *rel* attribute values of custom relationship types should be a URI itself or, an URI of it should be given in another attribute present at the same level of the *rel* attribute. Some documentation about the custom relationship type should be in this link target, to allow the clients to check up the actual link functionality (Thijssen, 2015).

However, when it is possible, the usage of the link types should be restrained mostly to the proposed, accepted and standard types, on account of their already established symbolism [7].

Some services place IDs of other resources, in the resources representations instead of links. There are two main disadvantages that are highlighted in the "ID instead of Link" alternative:

- Practicality
  - The client will have to know how to formulate the URI of the target resource;
  - If the hierarchy or the position of the resource changes, and if there is no redirection provided by the service, the clients can result in error;
- Heightens coupling
  - Although ID values or other key identification attributes, are usually not volatile, they can change. External IDs present in resources increase the level of direct knowledge between them, unnecessarily.

[7] Entities can check up the already accepted and approved relation types of links at the IANA website (Nottingham/Reschke/Algermissen, 2015) and can also register there new types for analysis and approval of experts.

As for the definition of which links should be presented in each resource, it depends on the business logic. When a developer of the client application starts to analyze a REST API for a possible consumption, he probably doesn't know anything about the service architecture, about its resource hierarchy nor about particular and possibly important functionalities. This is where the links can also help, by representing the resources and states in documents and provide relationships between them all, via links to other resources or documents. By exploring them, the clients will know how to parse and handle it, by its common metadata format, like a browser interpreting a HTML document.

HTML documents, also provide links which results in redirection, inclusion or integration. They can include style sheets which indicates the browser to include a particular document to style the page, or *Javascript* documents, which gives the ability to load code into the browser and execute it on the client. This particular technique is called Code On Demand. The following listing 3.21 illustrates the mentioned redirection, inclusion and integration.

Listing 3.21: HTML redirection, inclusion and integration of other resources.

```
<a href="/home/link-target">link target </a>
<link href="/css/base.css" type="text/css" rel="stylesheet">.
<script src="extScript.js" type="text/javascript" ></script>
```

Concluding the HATEOAS section, a brief description of the Richardson Maturity Model is given. According to some authors, the ultimate state of a system built on a truly REST architectural style, is when it applies the HATEOAS constraint (Richardson, 2008; Fowler, 2010). This model tries to classify and categorize the level of "RESTfulness" of web services and it is presented in the figure 3.4.

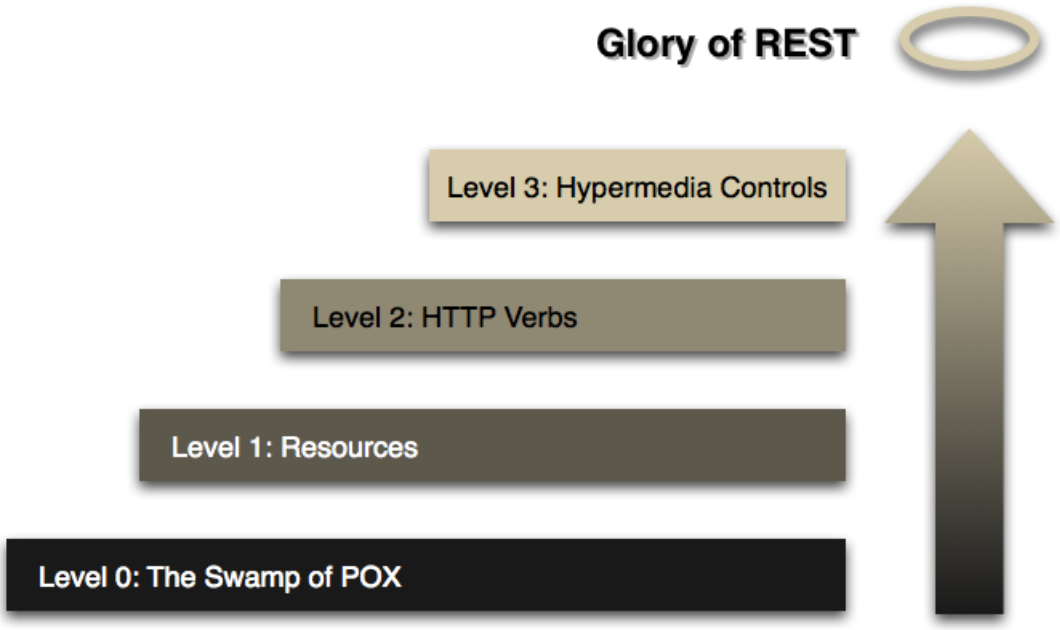

Figure 3.4.: *"Steps toward REST"* Fowler (2010)

As presented in the figure 3.4, the Richardson Maturity Model consists in four levels. Each level is briefly described in the following itemization:

- Swamp of POX [8]
  - Services in which there is only one allowed method in each resource or URI;
  - The usage of the transport protocol, such as the HTTP, without using further advantages of it, for instance, not utilizing it to specify the current action or to indicate the application state; This is usually found in XML-RPC and (in most) SOAP Services (Richardson, 2008).

- Resources
  - Services that differentiate each resource or data entry points with distinct resources;
  - Services classified in this level still utilize one method throughout the resources;
  - In a way, this can related to the Object Oriented (OO) programing paradigm.

[8] POX stands for Plain Old XML (POX), but the mentioned "Swamp" can be made of other formats such as JSON Fowler (2010).

- HTTP verbs
    - In this level, services use protocol properties, for instance the HTTP methods and response status codes, in order to distinguish the executing actions and their respective results.

- Hypermedia controls
    - Services that detail in each of its resources, their own capabilities, connections and adjacencies;
    - Services that have the implementation of the mentioned HATEOAS constraint.

3.6. Security

When the possibility or, the necessity of exposing business through services arises, the security topic becomes a major concern. It is a concern due to the fact that, according to many authors, there is no flawless way of achieving security in an application (Schneier, 2000), let alone in a web API which is, by definition, exposed. There are standard protocols, procedures and politics, however there is always imperfections, failures, mistakes or bugs in them plus, the human error factor. What can be done is an establishment of a comfortable level of security requisites, and an accomplishment of it.
Many authors also consider that security cannot be seen as a product, it should be looked over as a loop process of evaluation, identification of the flaw origin and, the implementation of the correction (Hope/Walther, 2008). There are some subtopics of security, such as authentication and authorization, these along with others topics, their respective techniques and other security challenges are briefly explored in the present chapter.

A large variety of web service security challenges and key elements have been identified by certain field experts (Brose, 2009; Singh, 2008; Lockhart, 2003). Some of them are itemized next:

- Message tampering
    - The modification of the information of a message in transit;
- Spoofing
    - Somehow similar to the previous item, but with the intuition of making the receiver ”think” that the sender is a valid one by endeavouring in message falsification;

- Unauthorized access
    - When information and/or access is given to someone that was not supposed to receive it;
- Man in the Middle
    - When an attacker intercepts messages, reads and/or manipulates before forwarding them to a third party;
- (Distributed) Denial of Service
    - When an attacker overloads a system's resources and capacity, making it unavailable to interact with valid requests. Instances of this attack have particularly become very common and powerful[9];

As mentioned previously, the security topic can be divided a bit, such as in Confidentiality, Integrity and Availability (CIA) (Park/Chen/Atiquzzaman, 2009). By having this concepts decoupled in this manner, the evaluation of vulnerabilities and exploits, can be done in a "divide and conquer" procedure:

- Confidentiality
    - Authentication;
    - Authorization;
- Integrity
    - Data trustworthiness;
    - Non-repudiation;
    - Audit;
- Availability
    - Environment fault tolerance;
    - Input fault resistance.

As for authentication, it relates to the identification of the end user or entity. The origin of the interaction, directly or indirectly starts with an entity, and so, its identity should be validated to check if it is allowed on the system. The user name (or login) and password are usually, and possibly, the most utilized authentication process. The system checks if the provided credentials (the user name and password pair) have a match in a configuration file, code, database or in a user repository, such as in the organization's LDAP system. One example of a login/password pair authentication, is the HTTP protocol's Basic Authentication, which is

[9] In the year 2014, the two biggest DOS attacks ever registered took place, the first with a 400 gigabytes per second (Gbps) rate and the second with 300 Gbps (Akamai, 2015; Arbor Networks, Inc, 2015)

detailed further in this chapter.

The authorization is another item that concerns the confidentiality and, usually follows the authentication. This item is solved by a procedure in which the system verifies the permission level of an authenticated entity on a particular resource. Some systems are prepared to face this problem via the setting of roles and/or permissions, following it with an association of them to entities. Another common way, that can be applied in parallel with the previous one, is the usage of a token or key, which the entity must possess in order to execute the desired action. These tokens are meant to indicate the user agent's authentication and authorization and are usually temporary, which is commonly resolved by having a life span and/or a maximum number of requests associated with it. An example of the latter control is given in the subsection 3.6.2 .

Two common types of authentication/authorization techniques in REST services is the Basic Authentication and the Token Based Authentication. Both of them are briefly detailed next.

As mentioned previously, there is an header property that is defined in the HTTP protocol that can be used in the authentication process, the *Authorization*. One of its utilizations is the Basic Authentication technique and it is commonly utilized, however, it is advised that this should only be used over an external secure system such as SSL or TLS (Franks et al., 1999), due to the fact that its biggest weakness, is that so the verification can occur, the password must be transmitted in the message and the critical data (the login/password pair) is not encrypted nor hashed. So, at least some cryptographic processing should take place in the message and/or in the channel.

In the Basic Authentication context, if an unauthorized request is received by a server, it should respond with a challenge, asking the user agent to provide valid credentials in a given realm (Berners-Lee/Fielding/Frystyk, 1996). To do this, the authentication scheme and the realm should be indicated in the server's response on the HTTP's *WWW-Authenticate* header property (Franks et al., 1999). The user agents that intend to utilize it, join the user name and password pair with a colon (:) in the middle, parse the resulting string in a Base64 encoding[10], append it in the *Authorization* header (Berners-Lee/Fielding/Frystyk, 1996) and respond to the challenging server with the generated string on the response's header. A brief example of this interaction is given in the figure 3.5.

[10] Base 64 Encoding or base64, is an encoding that results in arbitrary sequences of octets (Josefsson, 2006). Allows the use of uppercasing or lowercasing in the input, but the result is not supposed to be human readable. For instance "Book" becomes "Qm9vaw" and "book" becomes "Ym9vaw".

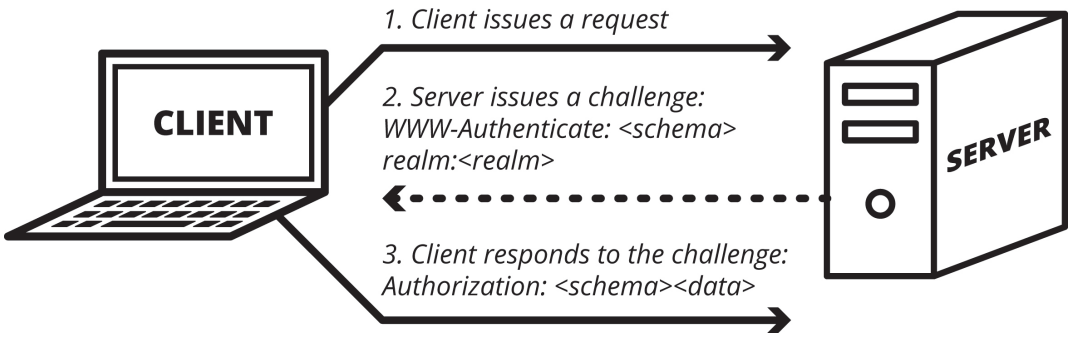

Figure 3.5.: Client-Server first security flow

The first response of the server in the figure 3.5, can be interpreted as *"I need you to authenticate before i handle any of your requests"*. Following that, the user agent submits its credentials for validation.

In the case that the server positively validates the credentials of a request, it should proceed with its respective processing, otherwise, if it does not successfully validates the credentials, it should respond with a 401 status code, indicating that the request is unauthorized.

One of the traditional Basic Authentication implementation's advantages besides the wide framework and browser support, is its simple implementation when comparing with other authentication/authorization techniques. Nonetheless, in a REST context, the traditional Basic Authentication mechanism presents further disadvantages, on account of the stateless constraint. As no traditional session is maintained, each time a user agent sends a request, the client should, besides indicating its intuit, openly identify the requesting entity in the HTTP header, thus increasing the exposure of critical information, the chance of an attacker to get hold of it and the possibility for Cross Site Request Forgery (CSRF) (Open Web Application Security Project, 2015).

As for the Token Based Authorization, it is usually an encrypted key generated and provided by the server and supposedly, the server should be the only one capable of interpreting it. After the server passes the token to a user agent, the latter usually includes it in its requests, which, the server fetches and validates in each interaction.

If there is the need to persist the token, and it usually is, for instance to access it for future references or post-validations, the token itself can be stored in a file, memory or database after its creation and client assignment. The actual size of the token is another concern in Token Based Authorization techniques, as it is sent in every request.

Although some services have custom solutions for this kinds of problems, there are standard frameworks to help with the access to third-party applications. Some of them are considered to be a good practice by many authors. For what concerns authorization, the *OAuth 2.0* Authorization Framework appears to be a good example of it (Hardt, 2012) and, for authentica-

tion plus authorization purposes, *SAML* and *OpenID* come up as safe alternatives (Campbell/ Mortimore/Jones, 2015; Lear et al., 2012).

3.6.1. Channel

So the message exchange of the communication can happen, a channel must be defined and, concerning its security in a REST environment, as both the service and client usually communicate via HTTP, it is possible to make the message exchange more secure under the aegis of HTTPS, also known as HTTP over TLS, HTTP over SSL, and HTTP Secure. Although it has its flaws [11], it is clearly more secure than HTTP (Chen et al., 2010), as it is a HTTP communication over an encrypted connection. HTTPS is commonly utilized in scenarios where there is the need for confidentiality, such as in online banking and online shopping.

Usually HTTPS connections rely on one of two secure protocols to encrypt the communications, the Secure Sockets Layer (SSL) or the Transport Layer Security (TLS). Both of this protocols utilize Asymmetric Public Key Infrastructure (PKI) systems. This systems differ from the symmetric keys due to the fact that it also can improve, besides confidentiality, the integrity and authentication of the sender.

Certificate Authorities or Certification Authorities verify the services ownership of public keys, which allows third parties, for instance a web service client, to rely on the service's trustworthiness. When a resource is accessed by a browser, such as in the Firefox or Google Chrome, they usually indicate its certification status. The figure 3.6 illustrates an untrusted certificate and the figure 3.7 illustrates a trusted certificate.

Figure 3.6.: Website with an untrusted certificate.

Figure 3.7.: Website with a trusted certificate.

Despite the HTTPS advantages, the providing server can accept communications via both HTTP and HTTPS. When utilized in parallel, these two types of channels have their entry points exposed in different ports. Some services (or their gateways or web servers) usually redirect requests that are sent to the HTTP port, to the port assigned to the HTTPS requests. The figure 3.8 illustrates an example of these two channels configured in distinct ports.

[11] As reported in the September 13th, 2011(Galperin/Schoen/Eckersley, 2011), more than 300,000 users had their Google accounts compromised, despite the utilization of the HTTPS protocol.

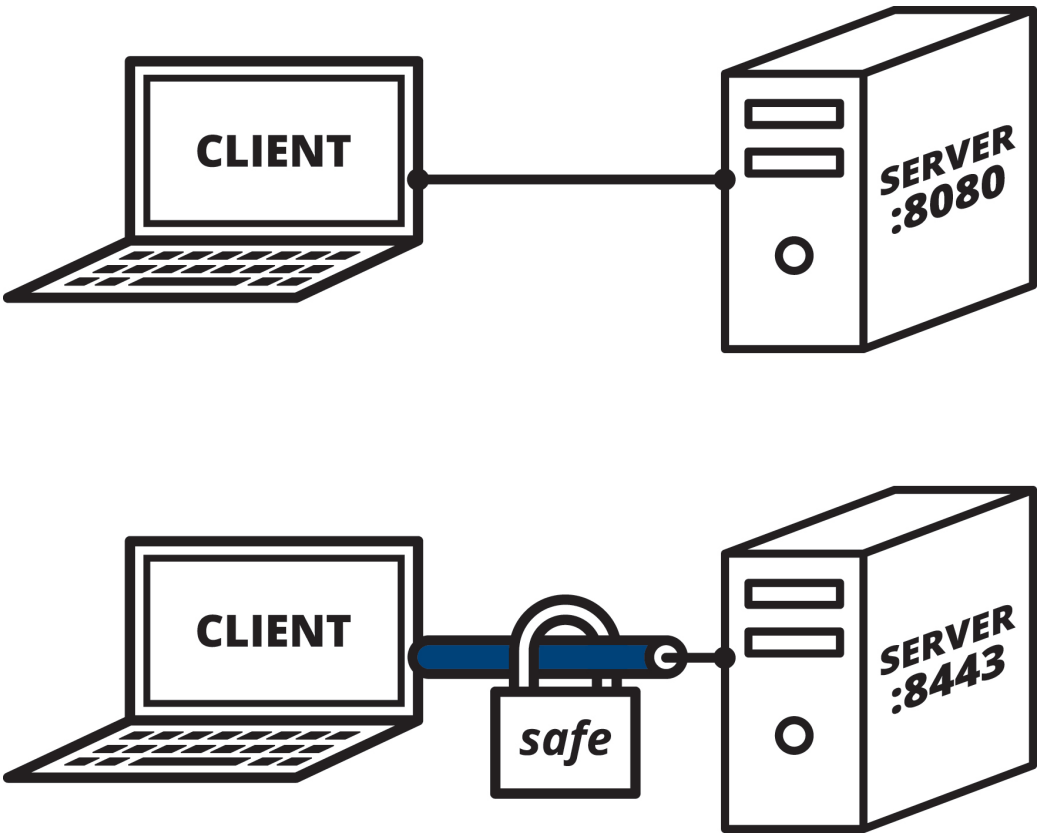

Figure 3.8.: HTTP and HTTPS channels configured in distinct ports.

3.6.2. Access Control

When analyzing the security aspects of an API, one of the first things to decide is to whom should be given access to the provided services. Should the API only respond to applications inside its own domain? Should it be globally available to authenticated servers? Or, should it even respond to client-side requests hosted outside its domain? The last scenario is commonly known as Cross-Origin Resource Sharing (CORS) (van Kesteren, 2014), and is largely observable in public APIs (*e.g.* the Flickr's API[12]).

The actual server-side's configuration of a CORS enabled service, depends specially on the application and/or web server. An excerpt of an example configuration file, of an *Apache's Tomcat* application server with enabled CORS, is listed in the following listing 3.22.

[12]There is a blog post mentioning it at *https://code.flickr.net/tag/cors/* .

Listing 3.22: CORS configuration in the Apache's Tomcat application server APACHE SOFTWARE FOUNDATION (2015)

```
1  <filter> <filter-name>CorsFilter</filter-name>
2        <filter-class>
3                  org.apache.catalina.filters.CorsFilter
4          </filter-class>
5    <init-param>
6      <param-name>cors.allowed.origins</param-name>
7      <param-value>*</param-value>
8    </init-param>
9    <init-param>
10     <param-name>cors.allowed.methods</param-name>
11     <param-value>GET,POST,HEAD,OPTIONS,PUT</param-value>
12   </init-param>
13 </filter>
14 <filter-mapping>
15   <filter-name>CorsFilter</filter-name>
16   <url-pattern>/*</url-pattern>
17 </filter-mapping>
```

A service that has CORS implemented usually includes the following header property in the response "*Access-Control-Allow-Origin: **". This indicates that the server allows requests of all origins (the corresponding configuration is presented in the lines 6 and 7 of the 3.22). However, the server can refine the allowed origins better with "whitelisted" origins, which results in a response with a list of origins, composed by protocol, host and port, separated by coma, such as : "*Access-Control-Allow-Origin: http://isep.ipp.pt:9080, https://isep.ipp.pt:9043*".

The server can also include an header property such as "*Access-Control-Allow-Methods: GET*" in the response, to indicate the allowed HTTP methods (lines 10 and 11 in the 3.22). This last property can also be configured as a whitelist, such as the *Access-Control-Allow-Origin* value.

Depending on the application and/or web server support, there is also more response header properties that can be configured in the response, to better refine the exposed headers (*i.e. Access-Control-Allow-Headers*), as well as to indicate user credential support (*i.e. Access-Control-Allow-Credentials*).

The CORS topic concerns the "reach" of the access control topic, however, there can be identified another concern, its period. For how long or how many times can the client do something? HTTP provides such answers with other properties. We can improve the access control by associating a limit to the allowed number of user agent's requests made with access tokens, during a rate limit period. An example of a service that controls the allowed number

of requests, is given in the listings 3.23 and 3.24. The listing 3.23 illustrates a request and the listing 3.24 an excerpt of the respective response.

Listing 3.23: A request with controlled access times.

```
1 GET /v1/search?Param1=x&param2=y&access_token=12345 HTTP/1.1
2 Host: bookSearchApi.com/api
3 Accept-Encoding: gzip
4 User-Agent: Mozilla/5.0
```

Listing 3.24: A response of a request with controlled access times.

```
1         HTTP/1.1 200 OK
2         Content-Type: text/html;
3         x-Ratelimit-limit : 5000
4         x-ratelimit-remaining: 4994
```

3.7. Service discovery

The discovery can be made in two ways, manually and dynamically. A manual discovery can be done via websites such as the one supported by the *ProgrammableWeb API* journal [13]. As for dynamic discovery, some registries exist to help it, such as Universal Description, Discovery and Integration (UDDI) (OASIS, 2004). Its utilization is depicted in the figure 3.9.

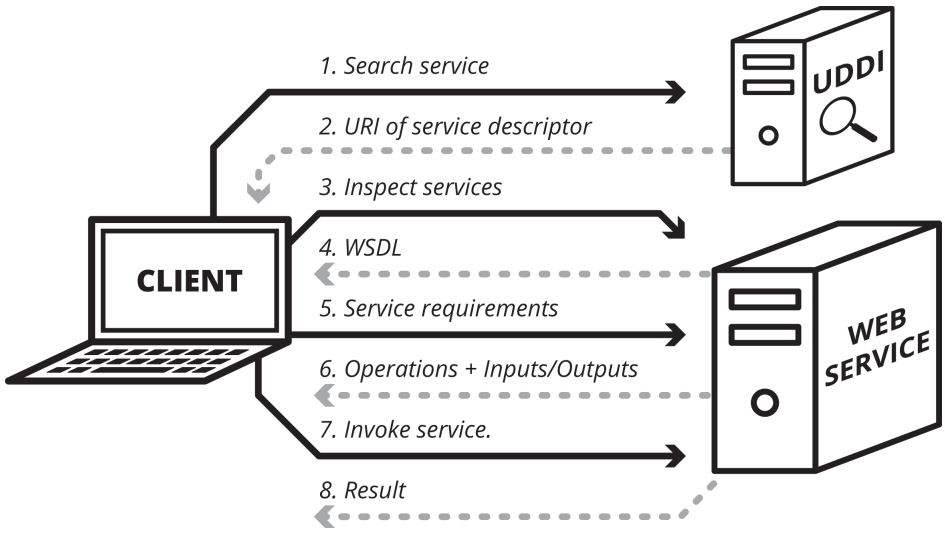

Figure 3.9.: Interaction with UDDI.

[13] The discovery support that this journal provides can be consulted in the web address *www.programmableweb.com/apis/directory* .

Chapter

4 Case study

The requirements of the projected design artifact's instantiation are detailed in the present chapter. The two following sections cover the functional and non-functional requirements of the desired final product and, they are followed by its analysis and specification, where some sketches of the weighted approaches are shown. To better specify the architecture of the approaches, some highlights of the applied architecture patterns are consecutively given. Some of the REST's design-specific features are documented next, such as the contract and its resources.

As this dissertation has an important hands-on purpose embedded in it, some more technical details are briefly explored further in the *Experimentation* section 4.3, such as the chosen development framework and the conceived access control and security aspects.

4.1. Requirements

This section briefly details some of the requirements and purposes that were defined for the desired resulting product of this case study. Due to the fact that all use cases of the experimentation product are not needed in order to describe the API in the web services context and, on account of the fact that it would only create complexity on the documentation, only the required use cases are mentioned.

4.1.1. Functional requirements

- The system must provide answers to enquiries executed over active, completed, terminated and failed BPM processes;
    1. These enquiries must be usable in any BPM process, meaning, the final product must be compatible with the already on-going process definition instances in the

production environment, the present in the test and stage environments and to the future processes as well;

2. Users can view, edit and add enquiries into the API;

- The enquiries must be optionally rich in data, meaning, when it is asked, the system should provide not only the requested BPM's metadata (*e.g.* the process due date), but also some business data flagged as process key informations (*e.g.* a request process's approval decision);
    1. The system must also be able to incorporate additional data into this optional information that was not included into the BPM process or task (*e.g.* user's company);
- The monitoring of all accesses and actions must be possible;
    1. As most of the time no high scrutiny will be required and, to also prevent performance issues due to the implied I/O routines, the respective monitoring levels must be adjustable in real time;
    2. In the case of an error occurrence, such as a repeated not allowed authentication, the support team must be warned in real time, with information detailing the situation with the requester origin, the access request body and the status code returned by the system;
    3. Other errors and technical faults should also trigger notification routines.

4.1.2. Non-functional requirements

To ensure some items of the software's quality metrics, some requirements were highlighted. Such highlights are briefly itemized bellow.

- Usability
    1. The system must be easily accessible and used by its client applications. In a case that a bad request is made, such information must be given back to the client application, detailing what went wrong;
- Interoperability
    1. The produced system must be able to interact with most of the HTTP-capable solutions;
- Security
    1. Although the deployment scenario of the case study system is an intranet environment [1], there should be client application authentication, channel security and user authorization implementations;

[1] The scenario illustrated in the figure 2.3.

- Maintainability
  1. The system should provide easiness to functionality broadening;
  2. The server-side code and project structure must be easily readable, manipulable and adjusted by the assigned support team;
- Performance
  1. The API should be as light as possible, as it will not have an exclusive hosting machine and, it will share the processing resources with other already deployed systems;
  2. The API should respond on an adequate time frame;
- Availability
  1. The API must have an uptime equivalent to the other APIs;
  2. As the API has a harsh single point of failure (*i.e.* the BPM API's server), it should be fault-tolerant. If the BPM server is offline for some reason, the API should indicate the faults origin to the requesting client application.

4.2. Analysis and specification

The following sections enumerate some of the weighted architectural approaches, specifically, how and were the enquiries to the BPM API should be executed.

4.2.1. Case study environment

The current state of the architecture in which the case study is carried out, is defined by two main distinct environments. One is a regular web application's environment, which is constituted by business and auxiliary databases and, various web applications, hosted on multiple application servers. The other one is composed by the BPM suite, which besides having its auxiliary databases, has an adjacent web application whose strict purpose is to interact with the BPM suite. A single application server hosts the BPM suite, its auxiliar databases and the adjacent web application. Such architecture is depicted in the figure 4.1, whose right side concerns the former environment (the regular web application's) and, the left side represents the latter environment (the BPM suite's).

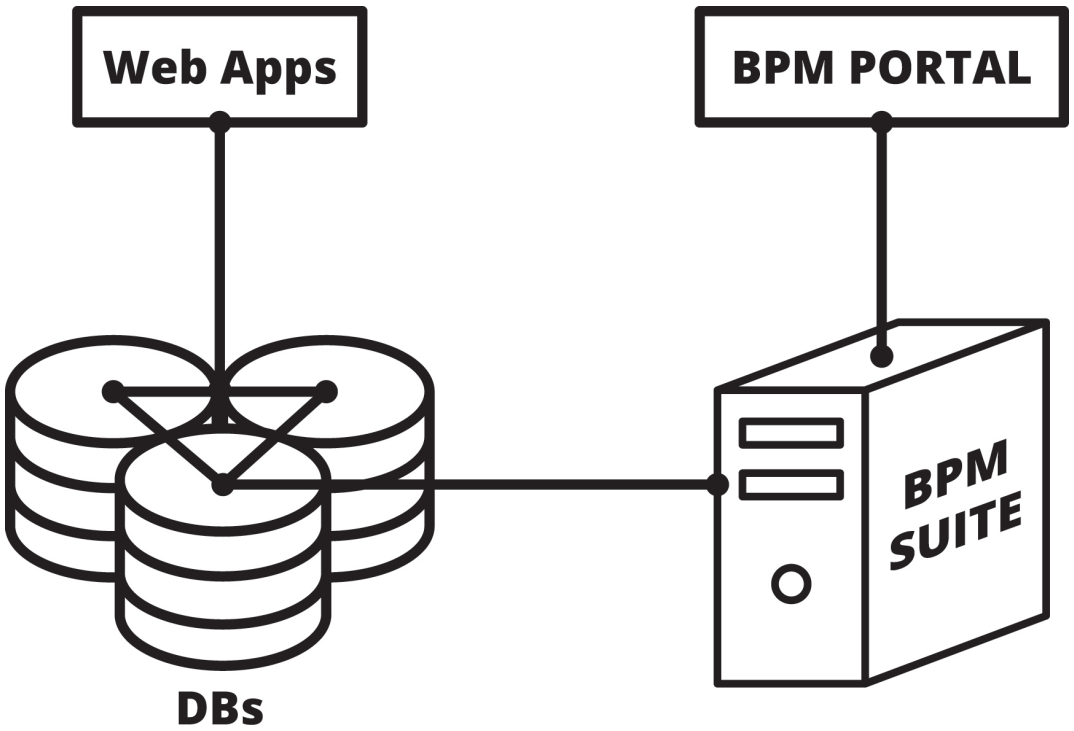

Figure 4.1.: Current state of the architecture.

It is to note that this architecture is similarly replicated in all other environments (*i.e.* the stage and the production's environment).

A possible approach to give the *WebApps* elements access to the BPM suite resources, would be through direct communication with the BPM suite's datasources. This approach is illustrated in the figure 4.2.

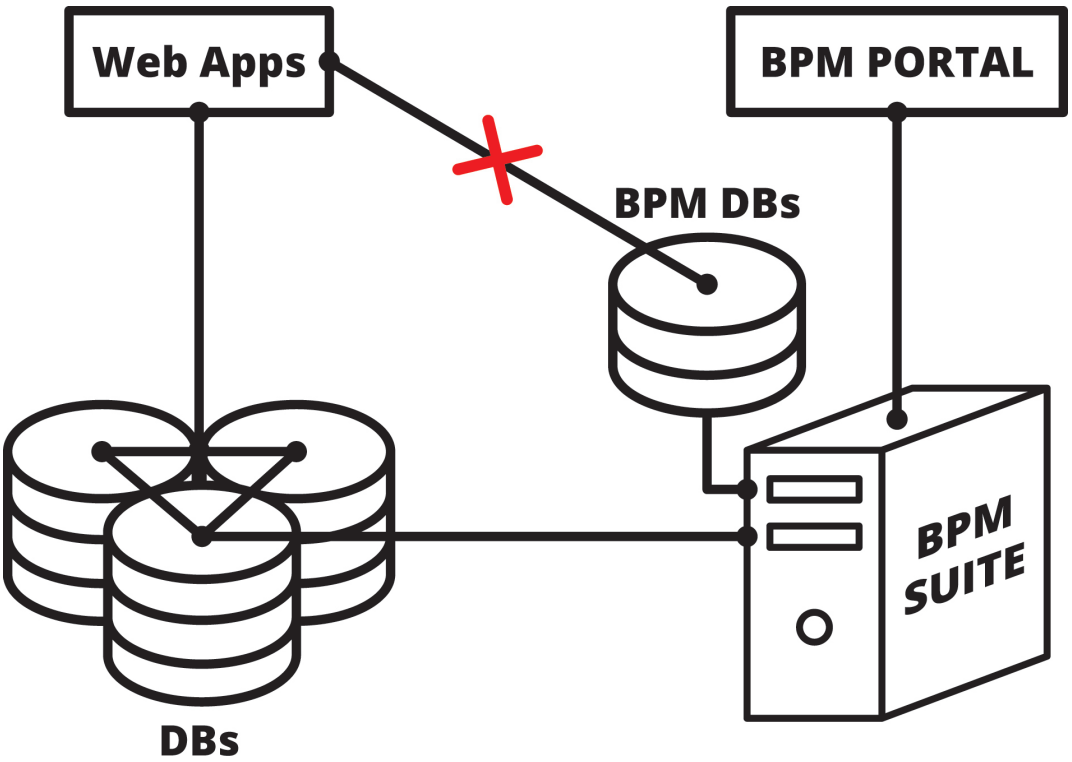

Figure 4.2.: BPM Suite's database access.

The previous alternative was rapidly classified as not proper on account of the facts that:

- Each new *WebApps* element would need to replicate the access steps;
    - If new *WebApps* elements were developed in different programming languages, no library-like toolkits could be re-used (*e.g.* JARs[2]) to encapsulate the access steps;
- Each new *WebApps* element's application server would have to be properly configured to reach the *BPM DBs*'s datasources;
- If the *BPM SUITE*'s version element is updated, there are no guarantees that its database structure would still be the same.

Another weighted alternative was the one illustrated in the figure 4.3.

[2] A JAR is a *Java* file format that enables the aggregation (or bundle) of multiple files into a single archive. The resulting JAR can then be imported as a dependency into another *Java* project. They are usually utilized for auxiliary purposes (*e.g.* code re-use).

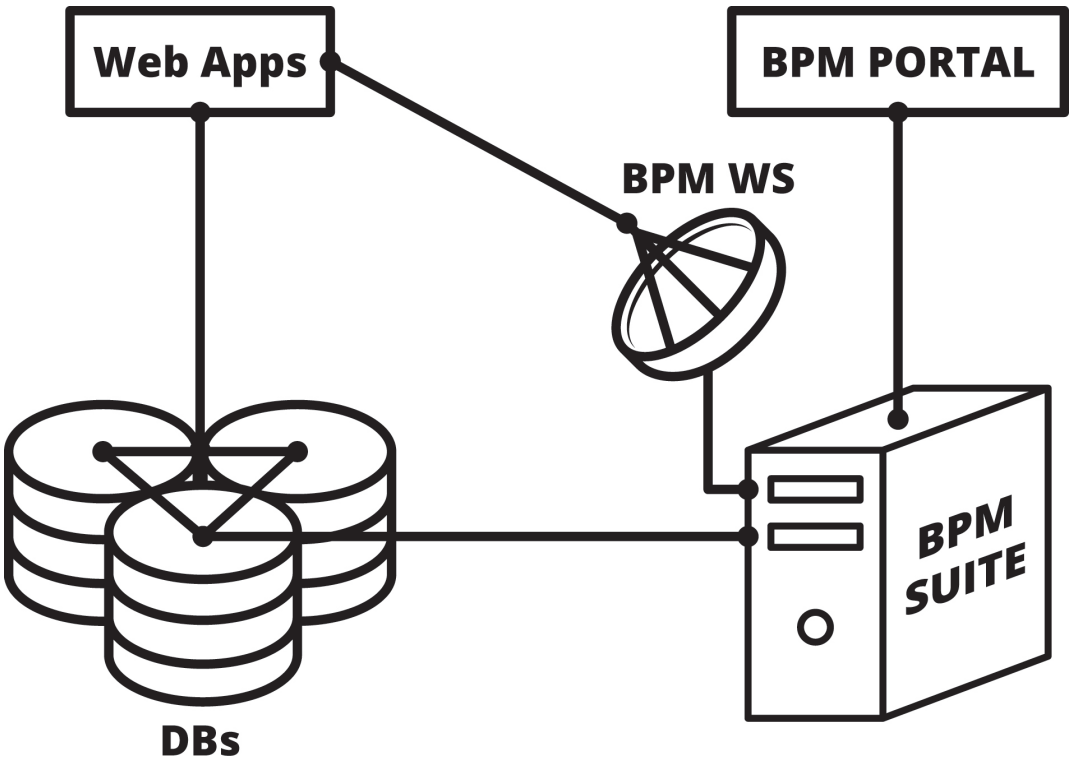

Figure 4.3.: BPM Suite's API access.

In the figure 4.3, the *BPM WS* element which is connected to the *WebApps* element, would interact with the latter via HTTP requests. The *BPM WS* element represents the BPM engine's API (represented by the *BPM SUITE* element). The API presents some RESTful characteristics however, it lacks representation media type varieties and, almost completely avoids the HATEOAS principles. Other disadvantage of this approach is the fact that the *BPM WS* consumption logic would also be repeated in new elements homologous with the *WebApps* elements. Nevertheless, despite the previous disadvantages, the *BPM WS* consumption would be relatively agile when comparing with the previous approach and was very considered.

Following the two previous approaches represented in the figures 4.2 and 4.3, another alternative was pondered. A middleware, or intermediate, could be added to the architecture in order to fill some gaps of the previously mentioned approaches. This third alternative seemed more suitable to respond to the problem at hand and, it is illustrated in the figure 4.4.

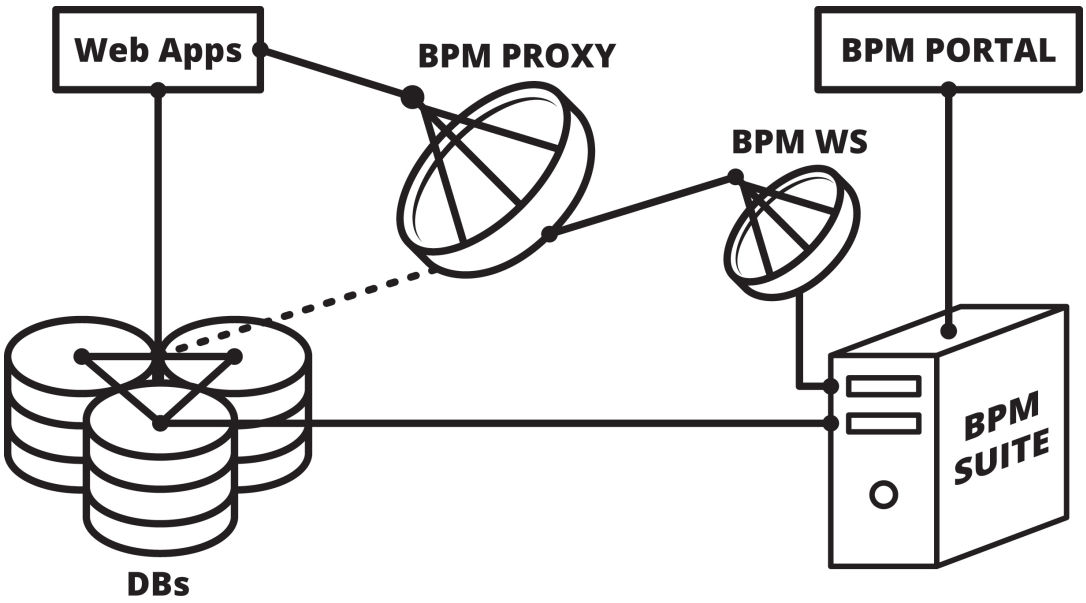

Figure 4.4.: Access via a middleware.

The alternative illustrated in the figure 4.4 is distinct from the previous alternatives, on account of the fact that there is a middleware agent (the *BPM PROXY* element) between the *WebApps* element and the *BPM WS* element. This alternative presents advantages and disadvantages. The considered advantages are listed in the following itemization.

- If the functionality implemented in the *BPM PROXY* element became necessary in another element suchlike the *WebApps* element (*e.g.* a mobile application or a desktop application), the presentation layers of this new client would be the only thing that would have be implemented for what concerns the BPM information consumption, meaning the source of the information, its parsing and processing would be centralized in the *BPM PROXY* element;
- When comparing with the alternative presented in the figure 4.3, this one wins in implementation time on the long run as well, due to the fact that the new clients, in the previous alternative, would also need to conjugate the *DBs* element provided information with the one provided by the *BPM WS* element;
- At least for the *DBs* element's conjugated information, no datasource configurations would be needed on the new client's application servers, as they would already be configured in the *BPM PROXY* application server;
- Responsibility desegregation;
- Agile openness for new client applications;
- Encapsulation of the responsibilities and functionality logic needed for the interaction with the BPM Engine, this would prevent any interference from external sources (*e.g.*

client applications) and possible misuses;

- Streamlining the provision of the clients, as it would be faster and safer, on account of the fact that the API will be tested by previous clients.

The considered disadvantages were:

- The development time needed in order to make the *BPM PROXY* element as generic as possible was estimated to be relatively big;
- The architecture illustrated in the figure 4.4, indicates at least two major points of failure, the *BPM SUITE* and the the *BPM PROXY* itself;
- If the *BPM PROXY* element was to be deployed in a exclusive machine (*i.e.* not on the *Web Apps*'s machine nor on the *BPM SUITE*'s machine), the communication packages would have be transmitted in at least four segments of the network (*i.e.* from the *Web Apps* to the *BPM PROXY*, then to the *BPM WS* element and finally, all the way back to the *Web Apps*);
    - The request-refining technique detailed in the listing 3.12 of the section 3.2.2 *Representation*, presented itself as a strong countermeasure to the disadvantages that were presented in the previous item, by avoiding the degradation of the system's performance and the actual user perceived performance.

4.2.2. Architecture and Design Patterns

As for the architecture and design patterns that the presented solution relates, it can be considered that it follows some principles of the EAI patterns, for instance, the Data Federation pattern. The role of the case study's API relates to the data federation pattern, on account of the fact that one of the goals of the required system is to handle and enrich information from multiple and heterogeneous information sources, and then provide it to the invoking clients (Sauter et al., 2012).

The API can also be associated with the Aggregator pattern, as it collects information originated from different but related messages and outputs a complete response. The following figure illustrates the latter pattern.

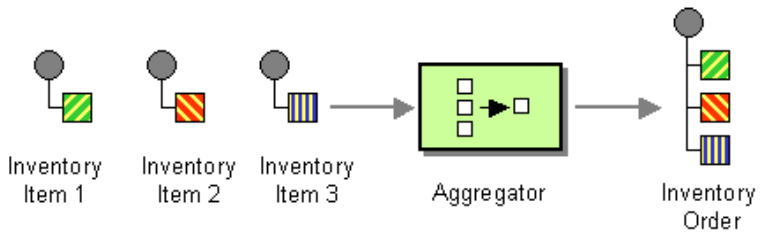

Figure 4.5.: "Aggregator pattern" Hohpe/Woolf (2003) *Enterprise Integration Patterns*

The BPM Proxy API can be related to another pattern, the Data Virtualization pattern, in which, an intermediary system hides the actual information providers and, enables the resulting services to join data from diverse sources without the need of creating redundant and unnecessary data copies (Morgan, 2012).

The output of the requests done to the BPM Proxy API, can also relate to the Data Transfer Object pattern, as it aggregates related information from various singular business entities.

The BPM Proxy element also has components that can be related with the Service Gateway pattern, as the responsibility of the interaction with the various datasources was delegated to particular software components (*e.g. Java* classes).

If the presented API is analyzed from a faraway "distance" and, we add other existing middlewares into the analysis scope, the conclusion that this API is established in a SOA-like architecture can be achieved. To reach the previously mentioned conclusion, some systems that are present in this API scenario are illustrated next, for instance, another middleware that is present in this API environment, which also has its own API, is the *Document Generator* API. This API, along with its respective use case's data flows is illustrated in the figure 4.6.

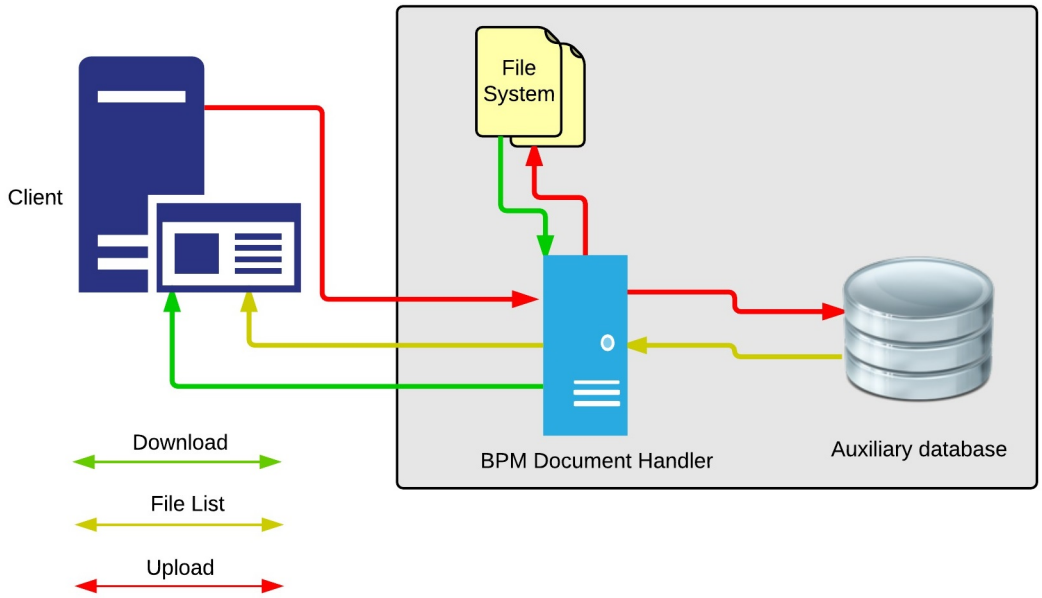

Figure 4.6.: Message exchange flows in use cases related to the document handling resources.

There are other elements that share similar architectural positions of the API presented in the figure 4.6, and it is expected that other elements hierarchically similar to it, will be added in this environment in the near future. On account of the fact that these elements have very specific purposes (*e.g.* upload and download files associated with BPM process instances), they align perfectly with the SOA constraint of the possibility to provide composite services. This SOA principle's alignment is illustrated in the figure 4.7, where another perspective of the architecture is given and, the service's interactions and its respective targets are depicted via differentiated color codes.

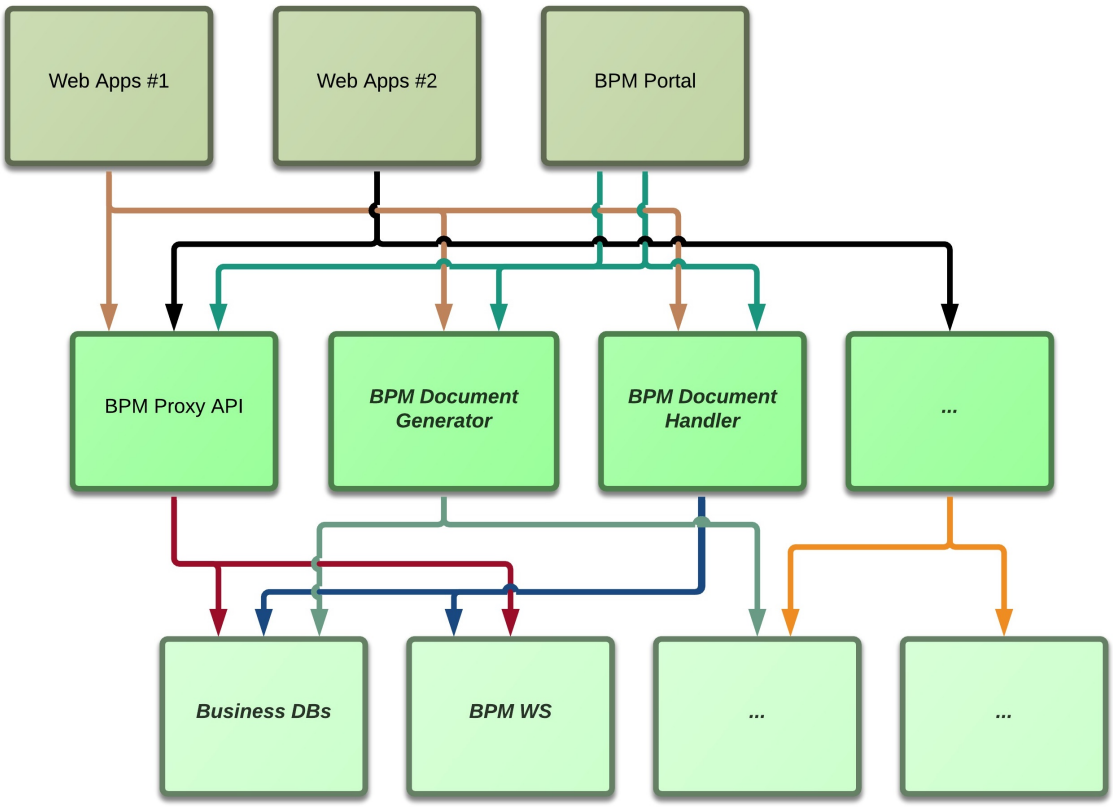

Figure 4.7.: BPM Proxy in a SOA architecture.

Besides the fact that all of this architecture's services can be re-used to achieve new composite services and new functionalities, this architecture also shows explicit boundaries, autonomous services, exposed APIs reachable by standard protocols and loosely coupled implementations, some of the core principles of SOA.

4.2.3. Resources

Following the architecture and some of the design patterns present on the API (described in the section 4.2.2), the present section starts by highlighting some of the API's main resources,

which are exposed to the surrounding services and applications.

There were two kinds of resources that were established to be a part of this API, which are listed in the following itemization:

- The business resources
  - these resources include BPM-specific elements, such as process definitions, process instances and process tasks;
- The control resources
  - these resources are consisted of API auxiliary functionalities such as the API versions and resource trees.

Other types of resources, such as *Token* resources and *User* resources, were also pondered to be included in the API's control resources. However, on account of the fact that they were not considered as crucial as the other main resources for the current state of the project and, as it could possible break the explicit boundary principle (*i.e.* not suitable to the API's functionality scope) they were not included in the current API's resource tree, which is illustrated in the figure 4.8.

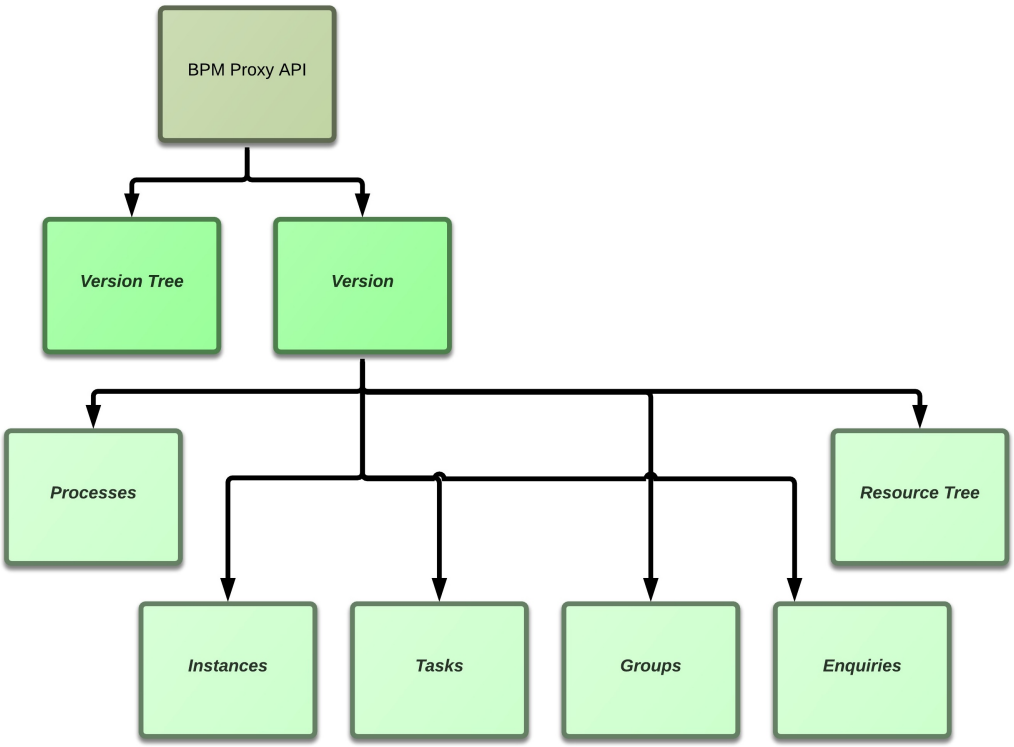

Figure 4.8.: BPM Proxy resource tree.

The purpose of the *Version* resource illustrated in the figure 4.8, is to allow future clients which will be linked to a specific version of the API, to be able to share the same API with other clients linked to earlier or later versions. This *Version* resource can also be considered a layer that provides compatibility handling if, a possible version would become not compatible with previous versions. This *Version* resource can also be utilized to allow *beta* versions of the API to be published alongside the production-ready versions.

There are also two similar resources illustrated in the figure 4.8, the *Version Tree* and the *Resource Tree* resources. In the current state of the project, these resources are meant to be used only as documentation providers (*e.g.* list all available API versions), however, these resources could easily escalate its functionality to act as resource ”road-maps”, thus enriching the service description and complete the respective WADL's provided description. This descriptor is further detailed in the section 4.3.2 *Contract and service description.* The other resources, whose purpose places them in the ”business resource” category, are more detailed in the section 4.2.4.

There is another goal associated with the *Resource Tree*, it should also include, in each of its item's representations, all the respective documentation of the links relationships and, the possible API's custom link relationships[3]. These documentations are also further detailed in the section 4.2.4.

4.2.4. Methods and Functionalities

With the exception of the *Version Tree*, *Version* and *Resource Tree* resources which have only control-oriented goals (therefore mentioned from the present section on as ”control resources”), all other resources will be referred as being classified under the ”BPM business resources” category (mentioned in the section 4.2.3), on account of the fact that each of them have specific business purposes (*i.e.* the BPM's logic), as opposed to the former resources.

Most of the resources illustrated in the figure 4.8, have sub-resources associated with them. This distribution of resources and sub-resources representations can also be considered actual functionalities, for instance, a HTTP GET method invoked in the *Tasks* resource, should result in a representation of a collection consisted by *Task* (singular) item representations. These types of requests's results can also be classified as use case solutions. Some of the use cases that can be fulfilled by the BPM Proxy's *Instances* resources are represented in the figure 4.9. The other use cases that do not concern the *Instances* resources are hidden in the gray ellipses which represent other use cases dependent on other resources such as the *Tasks* or the *Enquiries* resources.

[3]This topic of the link's custom relationships was mentioned in the section 3.5 *HATEOAS.*

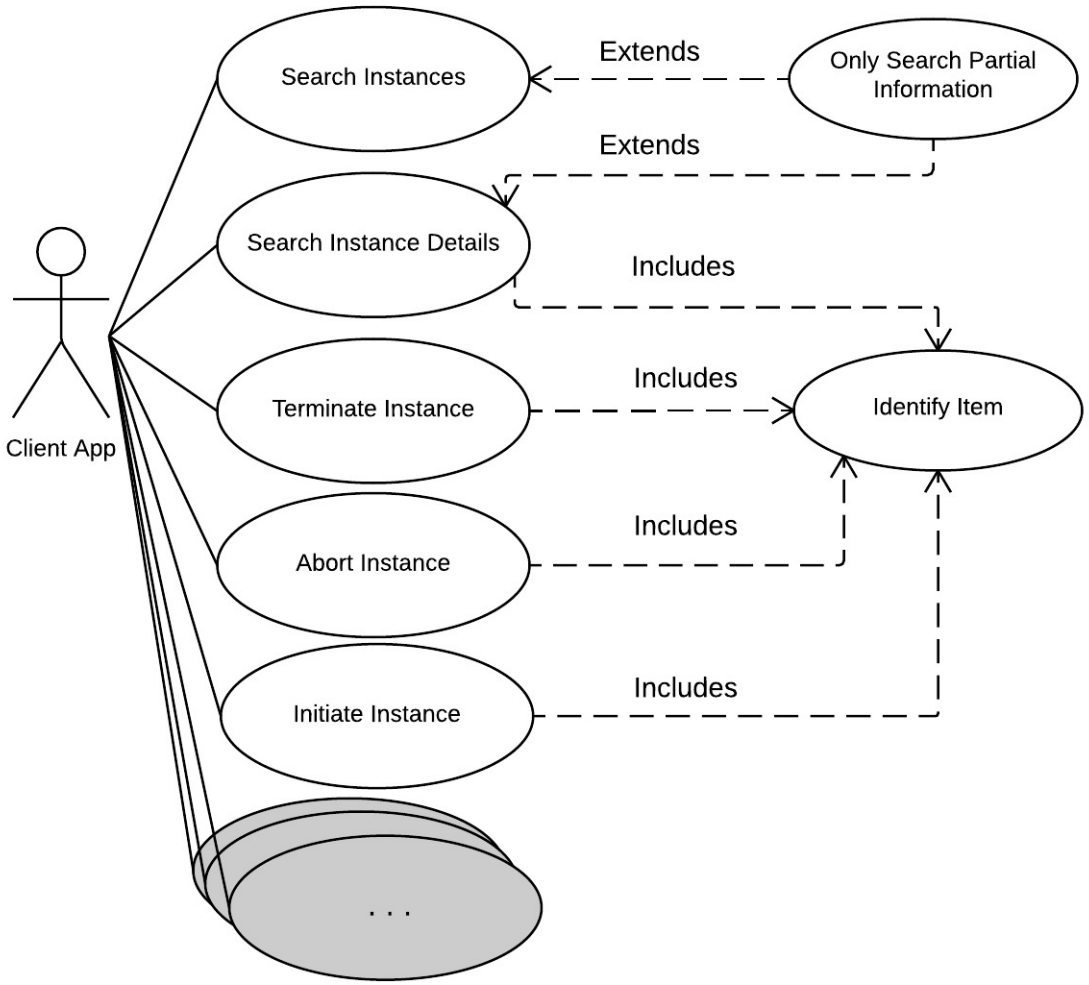

Figure 4.9.: Some of the use cases that can be fulfilled by the BPM Proxy's *Instances* resources.

There are other use cases that the API can respond, such as the definition of new queries (or searches) into the *Enquiries* resource. Similarly to the *Tasks* resource, a successful GET request to the root *Enquiries* resource should return all the available resources to the authenticated user agent, however, it should also expose other uniform interfaces besides the GET, such as the POST or PUT to create a new *Enquiry* resource. As for the actual single items (*e.g.* a Enquiry's resource representation), another variety of methods should complement its interface, such as the ones listed in the following itemization.

- The PATCH method
  - A request with a HTTP PATCH method on a *Enquiry* resource, should partially update it (*e.g.* change a *Enquiry*'s column or label);
- The GET method
  - A successful request that identifies a single *Enquiry*, made with a HTTP GET

method on the *Enquiry* sub-resource, should trigger the API's system to execute the due processing and return its representation (*i.e.* the actual *Enquiry*'s results);

- The PUT method
    - A request with a HTTP PUT method on a *Enquiry* resource, should fully update it;
    - On the other hand, if the HTTP PUT method is executed on the *Enquiries* root resource, the API should create a *Enquiry* resource or, if the client provides a parameter that indicates an identification of a single item of the collection, the API should fully replace it;
- The DELETE method
    - A request with a HTTP DELETE method on a *Enquiry* resource, should delete it and make it not accessible to users;
    - There will be access control procedures present on the API processing logic, for instance, only authenticated users with the same profile (or higher) of the one that created it, should be able to execute the previously mentioned action (*i.e.* the DELETE method on a *Enquiry* resource).

The previously listed methods should also be exposed in the *Groups* resource with similar purposes, for instance, a *Group* resource representation that consists of all the assignees of a *Task*'s representation, should be editable in real time if the HTTP's respective method is executed successfully (*e.g.* removed or added).

To align with the REST's HATEOAS principle, all of the business resources should have links in their representations. These links should indicate the current hierarchical position of the resource and, provide access to the adjacent resources. An example of it is given in the following listing 4.1, where an excerpt of a representation in a JSON format is illustrated.

Listing 4.1: An excerpt of a dummy representation enriched with links.

```
1  {
2    "href" : "https://BpmProxy.gsc/api/v1/instances/66666",
3    "process" : "Budget approval",
4    "originator" : "Luciano Teixeira",
5    (...)
6    "resourceNavigation" : [
7      { "href" : "https://BpmProxy.gsc/api/v1/instances",
8        "rel"  : "up" },
9      { "href" : "https://BpmProxy.gsc/api/v1/instances/55555",
10       "rel"  : "prev" },
11     { "href" : "https://BpmProxy.gsc/api/v1/instances/66666",
12       "rel"  : "next" },
13     { "href" : "https://BpmProxy.gsc/api/v1/instances/99999",
14       "rel"  : "last" },
15     { "href" : "https://BpmProxy.gsc/api/v1/instances/66666",
16       "rel"  :
17       "https://BpmProxy.gsc/api/v1/resourceTree/cancel" },
18   ]
19   (...)
20 }
```

Besides providing resource navigation functions, the resource's adjacent links should also consist of other links related to the available additional actions or functionalities, with the proper customized relationships identified in the *rel* attribute, such as the one represented on the lines 15, 16 and 17 of the listing 4.1. The actual relationship's link (indicated in the *rel* value), should redirect user agents to the respective *Resource Tree*'s element of the control resources. The previous resource representation (*i.e.* the URI illustrated in the link 17 of the listing 4.1) should have a description of what the relationship's resulting action would do, among with other useful control information, such as a possible icon or warning to include in the user interface when an *onmouseover* Event is triggered on a HTML's element which instantiates a resource's representation. This resource's representation is illustrated in the following listing 4.2.

Listing 4.2: A resource's representation of a *Resource Tree*'s element which describes a custom link relationship.

```
{ "href"    : " (...) /api/v1/resourceTree/cancel",
  "info"    : "Cancel the execution of a Task or Instance.",
  "icon16" : " (...) /api/v1/resourceTree/cancel/icon?px=16",
  "icon32" : " (...) /api/v1/resourceTree/cancel/icon?px=32",
  (...)
  "resourceNavigation" : [
    { "href" : " (...) /api/v1/resourceTree",
      "rel"  : "up" },
    { "href" : " (...) /api/v1/resourceTree/delete",
      "rel"  : "prev" },
    { "href" : " (...) /api/v1/resourceTree/survey",
      "rel"  : "next" },
    { "href" : " (...) /api/v1/resourceTree/prioritize",
      "rel"  : "last" }
  ] (...) }
```

The *resourceNavigation* JSON element represented in the listing 4.2 is an array of elements whose items represent the adjacent action resources present in the *Resource Tree*. As these mentioned resources of the adjacent actions have equivalent functionality categories as the one that contains the *resourceNavigation* (*i.e.* the *cancel* resource), all of their link relationships were defined as being the standard ones (*e.g.* the *prev* or *next*).

The following figure 4.10 illustrates an excerpt of a dummy HTML document, where a few representations of *Instance* resources are listed. In the figure figure 4.10, it is also demonstrated some functionalities of the control resource's representations that can be utilized in different manners such as complementing client-side metadata, for instance the generation of the *src* value of a possible action on the *Instance* resource.

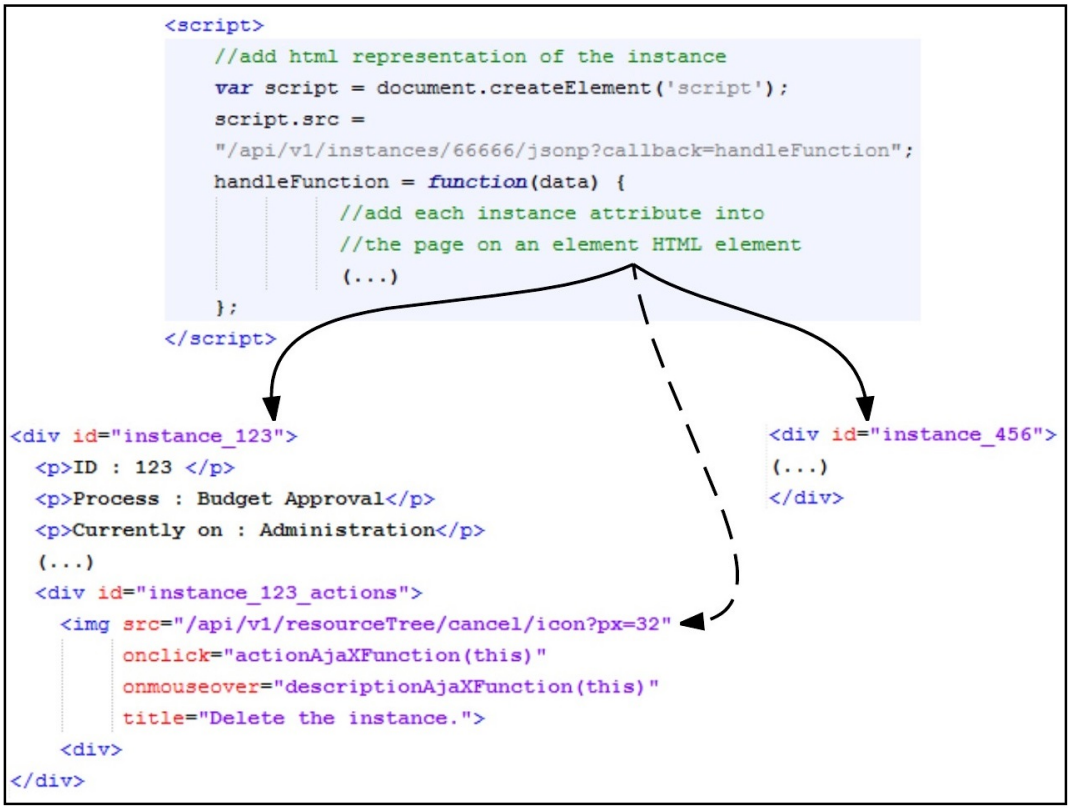

Figure 4.10.: Control resources and business resources utilized in a single representation.

Although the figure 4.10 illustrates only a theoretical prove of concept, it shows a wide range of possibilities of the REST approach applied in this API design architecture.

4.2.5. Planned tests

A considered element of the test plan was to implement, an actual "*API Tester*". This tester should provide a fairly view of the API's resource tree and its respective functionalities. Some other advantages that a "*API Tester*" development would provide when built in parallel with the API, is that it can be used to put the API to the test and provide the observation of new use cases of the API, or even new shortcuts between resources. The following figure 4.11 illustrates the main layout of the proposed *API Tester* application.

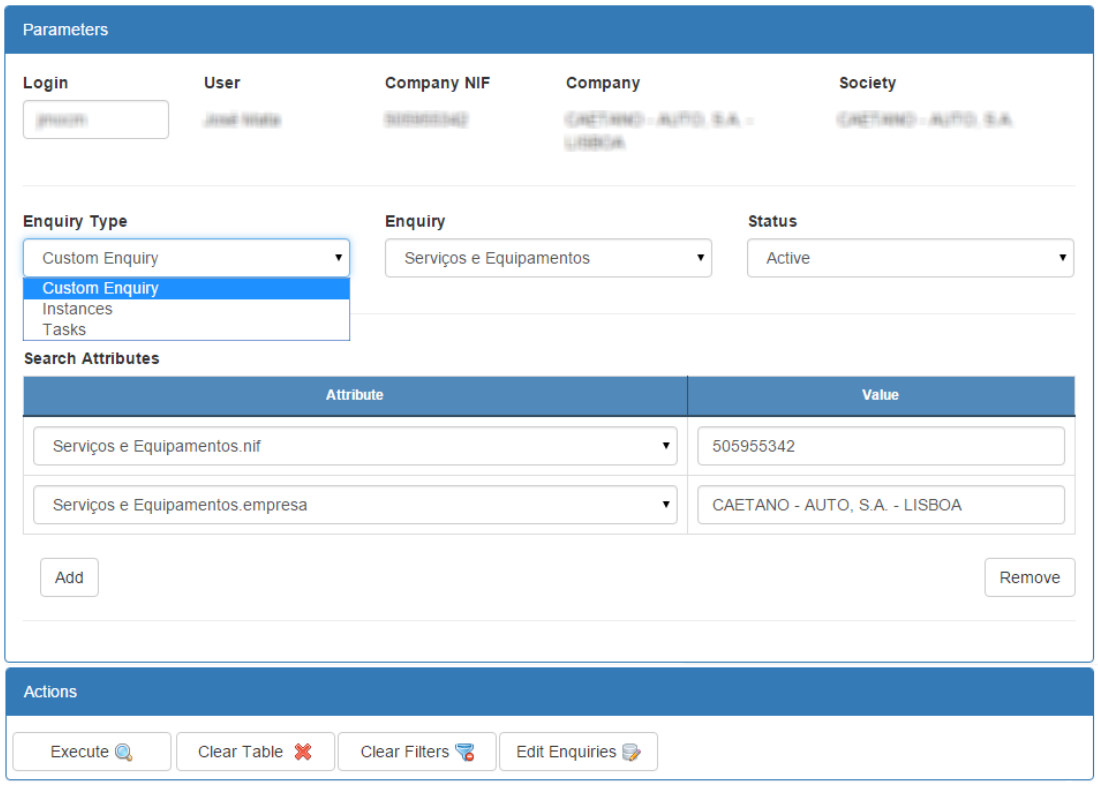

Figure 4.11.: Layout of the *API tester* application.

The select box present on the the left side of the figure 4.11 (*i.e. Enquiry type*) represents the three types of queryable resources present on the API. The information that is given in the upper side of the figure (*e.g. Company* and *Society*) represent information retrieved from the business databases. All the other information present on the user interface illustrated in the figure 4.11, was retrieved from the BPM suite's web service (*e.g.* the *Enquiry* select box is a sub resource of the *Custom Enquiry* resource).

Due to the fact that unit tests are crucial during and after the development of enterprise applications, such tests were planned to be taken in the API's infrastructure and functionalities. These types of tests were executed from the classes that represent programing objects to the actual resource interfaces. The tests that were executed in the smallest pieces of the API application were made via the *jUnit* framework which is briefly detailed and exemplified in the chapter 4.3. The tests that were made in the highest scope of the application's functionalities were made via the *Advanced REST Client* tool which is further detailed in the following section 4.3 and with the *API tester* application.

On account of the facts that browsers could be direct clients of the BPM Proxy API, the fact that different browsers can implement the handling of standards inconsistently, and also that browser-based user agents engines, such as the *WebKit* engine of *Safari* and the *Firefox*'s *Gecko* engine can handle the results of the HTTP requests differently, the API consumption's tests

should be made in different browsers.

4.3. Experimentation

The present section covers the development technologies that were researched and experimented during this case study and, the API's contract. Following that, some of the implementation's security and control aspects are briefly detailed, such as authorization and logging features. Two possible types of communication that are usually found in APIs are also detailed, meaning the synchronous and asynchronous communications. Two finalize the present section, some remarks of the procedures that usually follow the development and the tests of APIs (and other softwares) are given in the final sub-sections (*e.g.* versioning management).

4.3.1. Development technologies

The technologies that were chosen to proceed with the development were the *Apache Tomcat* version 8 to be the application server, the *Java* to be the utilized server-side programming language along with the utilization of the JAX-RS specification via the *Jersey* implementation. The selection of the *Jersey* implementation was made through the analysis of the respective documentation, whose development paradigms and available security mechanisms seemed suitable to meet the case study's requirements. An example of a suitable feature of the JAX-RS, is the way that the handling of dynamic content negotiation can be done. The following listing 4.3 exemplifies a service entry point code, whose clients can choose between two available formats, in XML or in JSON.

Listing 4.3: Dynamic content negotiation via the JAX-RS specification.

```
@GET
@Produces({MediaType.APPLICATION_JSON,
           MediaType.APPLICATION_XML})
public Response versionsGetImpl(@Context Request req) {
  MediaType types[] =
    {MediaType.APPLICATION_JSON_TYPE,
     MediaType.APPLICATION_XML_TYPE};
  List<Variant> variants =
    Variant.mediaTypes(types).add().build();
  Variant oVar = req.selectVariant(variants);
  if(oVar == null)
   return Response.notAcceptable(variants).build();
  String responseBody = (...);
  return Response.ok().entity(body)
         .type(var.getMediaType()).build();
}
```

As observable in the listing 4.3, the dynamic handling of the response's body format that can be achieved via this JAX-RS specification is very agile. If none of the available formats are indicated in the request's *Accept* header property, the API will respond with a status code of 406 *Not Acceptable*.

Another tool that was utilized was the *Advanced REST Client*, which is a Google Chrome's extension. Its interface is depicted in the following figure 4.12, where a request simulation is executed to the service that resulted from the code present in the listing 4.3.

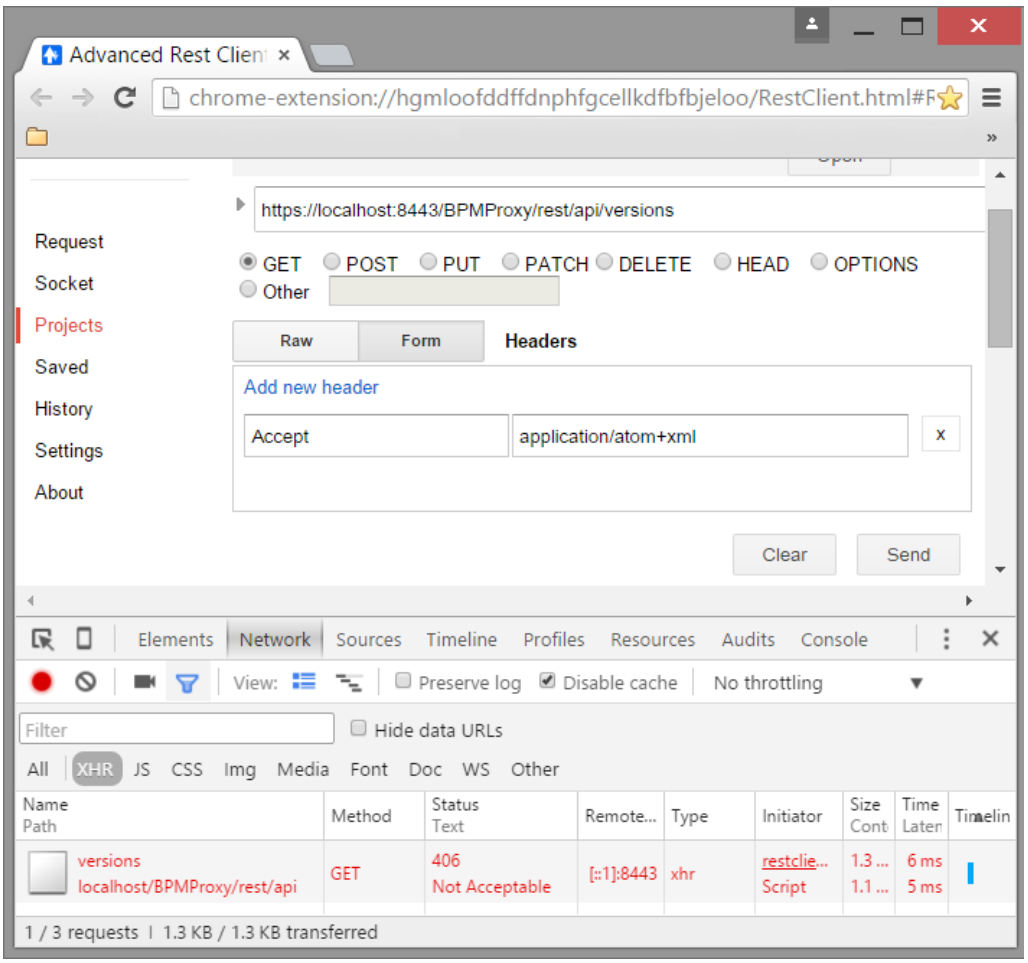

Figure 4.12.: HTTPS request simulation on the *Advanced REST Client*.

The simulated HTTP request had the header's property *Accept* with the value "*application/atom+xml*" (observable in the middle area of the figure 4.12). On account of the fact that the service detailed in the listing 4.3 only supported the "application/json" and the "application/xml" media types, the service responded with the status code 406 *Not Acceptable*, which is observable in the lower area of the figure 4.12 and whose respective code is detailed in the line 12 of the listing 4.3.

Another framework that was utilized, was the *jUnit* framework. The *Jersey* implementation's test functionalities are based on the *jUnit* framework and the *Jersey* class that extends it is named *JerseyTest*. The developers can create other classes that extend the *JerseyTest* class in order to execute the desired unit tests. An excerpt of a class that was put to the test is given in the following listing 4.4.

Listing 4.4: Excerpet of the *Version Tree* resource code.

```
1 @Path("/api/versionTree")
2 public static class VersionTree {
3         @GET
4         public String versionTreeGetImpl() {
5                 return "<resourceTree/>";
6         }
7 (...)
8 }
```

On the excerpt that is given in the listing 4.4, it is observable that a method that is called *versionTreeGetImpl* is implemented. This method name is a ”camel-cased” version of ”HTTP GET implementation of the *Version Tree* resource”. An example of a class that extends the previously mentioned *JerseyTest* class is given in the listing 4.5, which tests if the XML string ”*<resourceTree/>*” is truly the expected output of the *versionTreeGetImpl* method.

Listing 4.5: Unit test programming logic integrated in the *Version Tree* resource code.

```
1  @Override
2  protected Application configure() {
3   return new ResourceConfig(VersionTree.class);
4  }
5  @Test
6  public void testVersionTreeGetImpl() {
7   final String resourceTree =
8   target("/api/versionTree").request().get(String.class);
9   assertEquals("<resourceTree/>", resourceTree);
10 }
```

The executed code present in the line 9 of the listing 4.5, is a *jUnit* method that verifies if the two passed arguments are equal. These arguments can be *Java* data primitives (*e.g. char*) or *Java Objects* (*e.g.* a media type object of the class *javax.ws.rs.core.MediaType*).

4.3.2. Contract and service description

The chosen contract format to achieve the service’s description of the case study’s API was the WADL. Due to the research that was made to find a suitable framework to develop the case study’s experimentation (*i.e.* a *Jersey* implementation), the contract and the resulting service description development was eased up thanks to its inner functionalities. One of the utilization’s perks that the *Jersey* provides, is the service description assembling and its respective development time effort. The WADL service descriptor is updated each time a resource is

re-configured in the API's resource structure. An excerpt of it is given in the listing 4.6.

Listing 4.6: Excerpt of the service descriptor.

```
<application xmlns="http://wadl.dev.java.net/2009/02">
<doc xmlns:jersey="http://jersey.java.net/" jersey: (...)"/>
<resources base="https://localhost:8443/BPMProxy/rest/">
<resource path="/api/versions">
 <method id="versionsPutImpl" name="PUT">
 <response>
  <representation mediaType="application/json"/>
  <representation mediaType="application/xml"/>
 </response>
 </method>
 <method id="versionsGetImpl" name="GET">
  <response>
   <representation mediaType="application/json"/>
   <representation mediaType="application/xml"/>
  </response>
  </method>
</resource>
<resource path="/api/versionTree">...</resource>
<resource path="/api/versions/{VersionId}">...</resource>
</resources>
</application>
```

The request-refining technique is also available in the service description, if the request is made with the query parameter *detail=true* in the WADL resource, a WADL with additional information such as the representation's media type description and expected inputs, is returned to the requesting client. An excerpt of a service description of an expected input parameter is given in the listing 4.7, which represents the service that adds or updates a *Version* resource when a successful HTTP's PUT method is requested.

Listing 4.7: Excerpt of the service descriptor resulted from the request-refining technique.

```
1  <resource path="/api/versions/{versionId}">
2  <param xmlns:xs="http://www.w3.org/2001/XMLSchema"
3   name="versionId" style="template" type="xs:string"
4  />
5  <method id="versionsPutImpl" name="PUT">
6    <response>
7      <representation mediaType="application/json"/>
8      <representation mediaType="application/xml"/>
9    </response>
10 </method>
```

It is to note that in the line 2 of the listing 4.7, there is a XML schema definition that is referenced and utilized in the resulting WADL. This schema reference is a mechanism that is mentioned in the section 2.3.3 of the chapter 2, when describing the commonly utilized WSDL service descriptors that are used in SOAP services.

4.3.3. Security aspects and access control

The security is always a major concern in enterprise applications. This section focus on how one can implement a secure channel between clients and the server and also, how a custom token based authentication technique can be made to make the API's utilization more secure.

In all the environments (*i.e.* the development, staging and production environments), the utilized protocol for the channel is the HTTPS. As previously mentioned, the application server that was chosen for the development scenario was the *Apache Tomcat* version 8, in which the needed steps to implement HTTPS communication can be done. The following described steps concern a development environment with the *Windows* OS.

The first step is the creation of a *keystore* file to keep the server's private key and certificate information. It is to note that, as opposed to the normal staging and production environment's requirements, the signature of the certificate of the case study's development environment is only self-signed, and it is not verified by a internal nor external Certificate Authority (*e.g. Symantec* or *GoDaddy*). The executed commands of the *Windows command prompt* that should be ran to accomplish the first step are fully detailed in the listings 4.8 and 4.10.

Listing 4.8: Prompt's command to generate a private key and keystore file.

```
"%JAVA_HOME%\bin\keytool"
-genkey -alias tomcat
-keyalg RSA -keystore \development\keystore_directory\KSfile
```

The following step is to setup the generated *keystore* file in the server configuration, which in the case study development scenario is the *web.xml* file of the *Apache Tomcat* application server. This setup is detailed in the listing 4.9, which is an excerpt of the server's *web.xml* file.

Listing 4.9: Configuration *Apache's Tomcat* server's web.xml file.

```
<!-- Define a HTTP/1.1 Connector on port 8443 -->
<Connector protocol="org.apache.coyote.http11.Http11Protocol"
  port="8443"
  maxThreads="150"
  scheme="https" secure="true" SSLEnabled="true"
  keystoreFile="D:\development\keystore_directory\KSfile"
  keystorePass="xxxxxx"
  clientAuth="false" sslProtocol="TLS"
/>
```

After these two steps, the server already accepts communications via the HTTPS channel, which is illustrated in the figure 4.13.

Figure 4.13.: HTTPS connection test in the development environment.

The next step is the creation and installation of the certificate from a Certificate Authority. The first stage of this step is the creation of a local Certificate Signing Request (CSR). This CSR file is utilized by the Certificate Authority to create the certificate itself, which identifies the website and flags it as secure. The creation of the CSR file is detailed in the listing 4.10.

Listing 4.10: Creation of the certificate file.

```
keytool -certreq -keyalg RSA -alias tomcat
-file certificateRequest.csr
-keystore D:\development\keystore_directory\KSfile
```

The command detailed in the listing 4.10 outputs a file *certificateRequest.csr* which should then be submitted to the Certificate Authority in order to receive the actual certificate. After the certificate has been generated, the next step is to map it into the *keystore* configuration which is done by importing the Chain Certificate[4] of the Certificate Authority and finally, im-

[4]The Chain Certificate can also be referenced as Root Certificate.

porting it into the *keystore*.

After the establishment of a securer channel, the next concern was the access control. There were two types of accesses identified for this case study, which are detailed in the following itemization.

- Own domain access:
  - The *API tester* application, which is established in the same domain of the BPM Proxy API, should have complete access to the API's resources;
  - The *API tester* was defined to be a basic web application, mainly consisted of HTML, *JavaScript* and CSS components.

- External domain access:
  - In this scenario, the requests are originated from applications established in domains which are external to the BPM Proxy API;
  - Due to cross-site scripting security issues, no web pages hosted in external domains should be able to interact with the API;
  - Clients established in this scenario, should only be server-side clients so no CORS mechanisms should be implemented.

Other weighted concerns of the security topic were the authentication and the authorization issues. The authentication concerns were approached via the designing of a *Token Based Authentication* mechanism, whose paradigm follows the steps illustrated in the following figures 4.14 and 4.15. The authorization concern was met with the security annotations of JAX-RS via the verification of *Roles*, whose description follows the authentication diagrams.

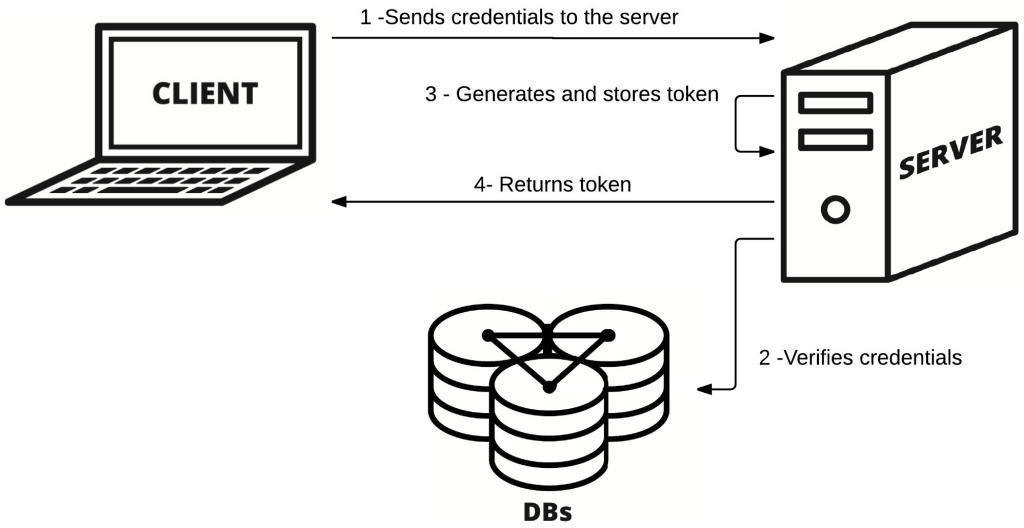

Figure 4.14.: First steps of the *Token Based Authentication.*

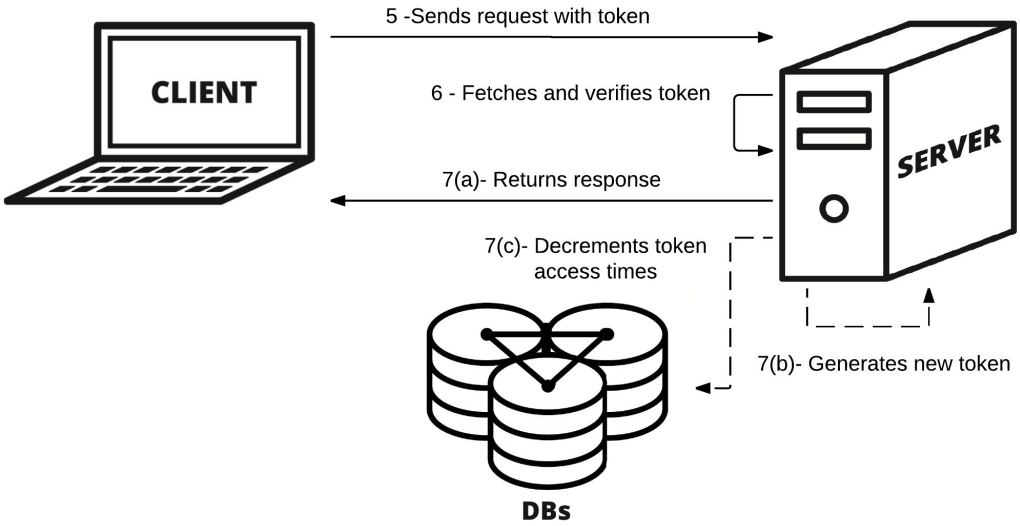

Figure 4.15.: Second steps of the *Token Based Authentication.*

The steps illustrated in the figure 4.15 with the dotted arrows [*i.e.* steps *7(b)* and *7(c)*], are optional and/or conditional steps. The step *7(b)* only happens if the token that the client sent in the step *5* was classified as expired in the step *6* or, the step *7(c)* has looped until its end. The step *7(c)* only takes place if the token has a limit of access times associated with it.

It is to note that the persistence illustrated by the *DBs* element present in the figures 4.14 and 4.15, could also be memory-stored data or placed in files established in a file system reachable by the API's server.

The designed authorization mechanism is detailed in the listing 4.11, where an excerpt of the *Java* class that represents the API *Versions* resource is detailed.

Listing 4.11: Excerpt of the *Java* class that represents the API *Versions* resource.

```
@Path("/api/versions")
@PermitAll
public class Versions {
        @RolesAllowed("AllClients")
        @GET
        public Response getVersions(@Context Request req){
          /* handle version list (...)*/
          return response;
        }

        @RolesAllowed("AdminClients")
        @PUT
        public Response put(@Context Request req) {
          /*handle version input (...);*/
          return response;
        }
}
```

The resulting authorization of the mechanism detailed in the listing 4.11, defines that requests originated from applications in the role *AllClients* can access the *Resource Tree* via the GET method and, that only requests originated from applications in the role *AdminClients* can access this resource via the PUT method.

These mechanisms provided by the JAX-RS API can be utilized to make the application's architecture to align with the principle of least privilege, which is considered to be a good practice.

4.3.4. Logging

As it is important to know who (or what) did and when it happened, logging frameworks have become important support tools during the software's development and production life cycles. The experimented logging framework that was implemented in the case study was the *log4j* (Apache Software Foundation, 2012). Its working mechanism allows the definition of various logging and notification procedures depending on the actual environment (*i.e.* the development, stage and production environments). The notification procedures can be as passive as a new entry in a log file established in a file system reachable by the application or, more active such as an email message.

Log4j also provides four main logging levels, the *trace*, *debug*, *info* and *error*. The resulting logs originated from the invocations of the *log4j* methods that implement the previous levels can be adjusted in real time, which answers to a requirement defined in the section 4.1 of the present chapter. These logs include messages that can consist of anything that is passed by the exception handler functions as an argument and, other additional information that is predefined in the respective environment's *log4j* setup file (or configuration file). These predefined informations are defined via the usage of patterns, such as the one indicated in the following listing 4.12 which is an excerpt of the utilized configuration file.

Listing 4.12: *Log4j* message information defined via a format pattern.

```
1  # SMTP appender
2  log4j.appender.myMail=org.apache.log4j.net.SMTPAppender
3  log4j.appender.myMail.Threshold=error
4  log4j.appender.myMail.BufferSize=3
5  log4j.appender.myMail.To=abcdefg@gsc.com
6  log4j.appender.myMail.From=webmaster@gsc.com
7  log4j.appender.myMail.SMTPHost=myEmailHost.gsc
8  log4j.appender.myMail.Subject=[Production] BPM Proxy Error
9  log4j.appender.myMail.layout=org.apache.log4j.PatternLayout
10 log4j.appender.myMail.layout.ConversionPattern=
11 [%d{DATE}]-[%-5p]-[Production]-[BPM Proxy]-[%C]-[%p] %d %c %M - %m%n
```

The pattern detailed in the line 11 of the listing 4.12, produces an entry describing the log level, the current timestamp, the respective *Java*'s class and method and, the information passed by the exception handling method. In the present case study's context, it was defined as crucial that the message of the log entry should be consisted of the resource, the resource's HTTP method interface, the returned status code and the requesting client identification. An excerpt of a sample SMTP log entry is detailed in the listing 4.13.

Listing 4.13: BPM Proxy's *Log4j* message.

```
1  [08 Out 2015 18:40:20,915]-[ERROR]-[LocalHost]-[BpmProxy]
2   -[com.gsc.bpmproxy.apiClient.BpmRestClient]-[getCallRestApi(421)]
3
4  GET  http://bpmproxy.gsc/api/versions/v1/abc
5
6  Failed: HTTP error Status Code: 404; Message:  Not Found;
7
8  Exception: Source error;
9
10 BPM Api response: 404 - resource abc Not Found;
11
12 BPM Api origin client: API Tester App;
```

4.3.5. Synchronous and asynchronous communication

There are usually two types of communications that can be found in web services, the synchronous communications and the asynchronous communications. This asynchronous type of communications usually involve one of the techniques that are detailed next.

The first type of asynchronous communications is the *callback*, that already has been partially exemplified in the chapter 3 on the listing 3.13. In this paradigm, the requesting client somehow indicates its own endpoint that the server should utilize to respond to the request. The client's endpoint that the server should utilize is usually indicated in the client's first interaction, and it is detailed in the line 12 of the listing 3.13. In the asynchronous communication types, the clients can generally continue its processing without waiting for the server response, which is the case of the 3.13 listing, where the browser can continue the page rendering or event-handling routines while waits for the server's response.

The other type of asynchronous communication is the *polling*, which differs from the previous asynchronous communication type on account of the fact that the client has the responsibility to check if the server has already executed its due processing.

There are two main issues or challenges that can be encountered when designing asynchronous communications, the correlation of messages and the handling of out of order messages (Sousa, 2015b).

As the services of the experimentation and their respective results have an "immediate" nature and can directly respond to its use cases, only synchronous implementations were designed.

4.3.6. Installation and versioning management

If the API had been planned to be published in *Internet* or *Extranet* scenarios, after the respective installation, some helpful client libraries with auxiliary functionalities could be offered. There are some public web services that provide representation code generators for popular programming languages such as *Java* and *C#*. A common practical example of this kinds of libraries are the already mentioned *JAR* files, which entities such as *Amazon* and *Google* provide to ease up the development efforts of API consumer clients (Amazon, 2014; Google, 2015).

To handle the compatibility of the clients and service's versions, the *Version Tree* resource could include in each of its representations the current version of the client library and other useful information, such as its respective download link.

In order to provide multiple versions of the API working in parallel, within the same project, the API version was defined to be a resource itself. Such approach is illustrated in the next figure 4.16.

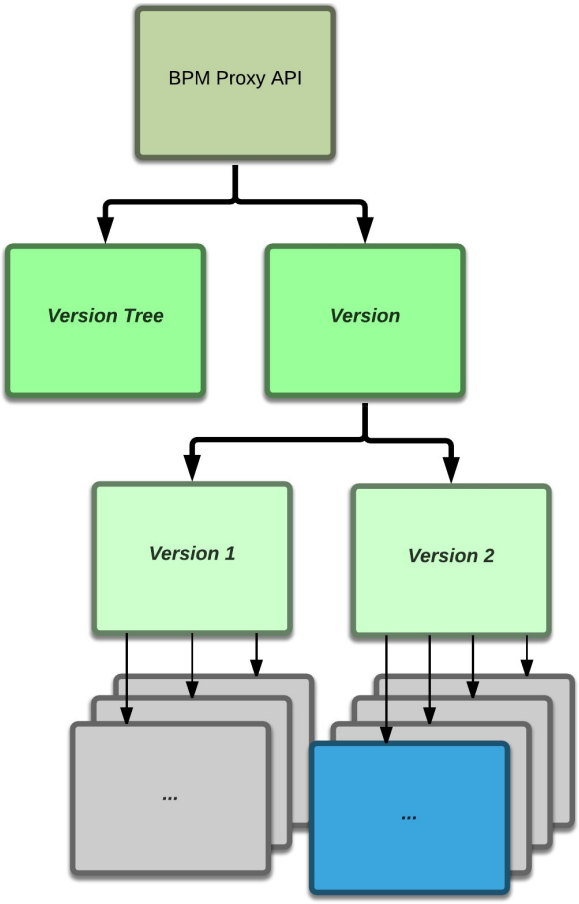

Figure 4.16.: *Version* resources of the API.

To ensure the forward and backwards compatibility as most as possible, the output that is given to the client should be as similar as possible to the previous versions. This will allow clients that were developed in a coupling manner with the API, to gracefully continue its processing that interacts with the new version of the API. To prevent parsing errors in the clients, the responses should highlight the current version of the API that was reached and the one indicated in the client's request.

To prevent other types of conflicting changes between the API versions, a stage server should be available. This types of servers are usually distinct from the development servers however, they should be as similar to the production server as most as possible, to ensure more controlled simulations and that the executed tests are as close to the real executions as possible, without affecting the production versions. This also allows client and supervisor entities to test and approve new versions of the API.

Other mechanisms such as version control systems (*e.g.* SVN) and automated build mechanisms (*e.g. Ant*) can also be utilized during the development stage.

Chapter

5 Analysis of the results

The present chapter is mainly focused on describing the analyzed results of the case study's experimentation that was carried out and documented in the chapter 4. Following it, an overview of the defined issues, characteristics and metrics is briefly detailed. The section 5.1 also finalizes the answers to the *RSQ5* and *RSQ6* research sub-questions that were indicated in the section 1.3.1, whose purpose was to find some of the characteristics and metrics that could classify the architecture as viable.

The present chapter is finalized with certain software design remarks that were concluded and observed during the experimentation.

5.1. Considered issues, characteristics and metrics

The present section iterates through the considered issues and how they were attended during the experimentation documented in the chapter 4. The sub-sections of the present section lists some of the issues that were considered qualitative metrics during and after the development of the case study's experimentation and, a brief overview of the degree of their accomplishment is described. As mentioned in the section 2.4, these metrics are considered subjective and are also very dependent on the specific necessities of the deployment scenario and on the business requirements. Due to the fact that the designed architecture is a new element and, the fact that it did not substituted a previously established architecture, no values are currently available for comparison purposes. However, the instantiation of the designed architecture can be utilized to initiate new comparative studies in order to evaluate other new proposed architectures.

5.1.1. Reliability level of the communication and messages

The communication implemented in the case study is stateless, which is one of the REST architectural style's constraints. The client applications identify themselves in each interaction

and so, no session is kept nor maintained on the server.

The main communication protocols that were utilized were the HTTP and the HTTPS. Despite the fact that the HTTPS mechanism of communication embroils some complex hidden elements, such as the channel encryption, it has a simple implementation effort thanks to the utilized frameworks, the development tools and specially due to the HTTP and HTTPS specification's maturity and standardized positions. This simple implementation is demonstrated in the section 4.3.3 of the chapter 4. The utilized protocols proved to be very efficient and reliable during the global tests.

5.1.2. Degree of heterogeneity

As the developed API exposes its functionality interfaces via HTTP methods, it is highly reachable by various types of clients. Of course these clients must be capable of interacting via the HTTP protocol however, the HTTP and HTTPS protocols allow most of the current enterprise applications to interact with the API independently of its native programing language and residing application server.

The API itself can also be installed in most of the current operative systems, due to the advantages brought by the chosen programming language, which was the *Java* and the fact that it doesn't need any demanding specifications of hardware nor software requirements.

5.1.3. Grade of Transparency

All communication between the designed API and its data sources (*i.e.* the BPM Suite and the business databases) are transparent to the client applications. This transparency characteristic is more observable in the figure 4.7, in which the BPM Proxy element acts as a layer between the client applications and the various information sources that they indirectly utilize.

5.1.4. Classification of the failure handling mechanisms

As mentioned in the section 4.3.4 of the chapter 4, the designed API handles well the faults that can happen in its data sources thanks to the implemented *Try-Catch-Finally* [1] mechanisms in all of the critical points (the components that followed the guidelines of the Service Gateway pattern), such as the HTTP connections to the BPM's suite and the connections to the business databases.

The API also prevents failure points when handling possible errors that are inputed by the client applications in its main interfaces. An example of it is given in the listing 4.3 and in the

[1] *Try-Catch-Finally* is an error handling mechanism that is largely utilized in multiple programming languages, such as *C#* and *Java*. Its logic is divided in three steps: the *Try* where the critical instructions are usually present, the *Catch* in which the fault's message and other related information is handled and the *Finally* block, whose purpose is to revert possible damages made by the fault and close I/O procedures and/or connections.

figure 4.12 of the section 4.3.1, where a simulated client application asks for a unacceptable response media type. This error handling mechanism are also demonstrated in the line 10 of the listing 4.13, where an excerpt of the error message that was passed to the client is detailed in the logging email notification.

The WADL descriptor and the OPTIONS method present on each resource also help clients to avoid calling unimplemented or unavailable resources and other interfaces of the API. In the situations that the user agents try to execute an HTTP method that is not implemented, the service will respond with the HTTP status code 405 *Method not allowed*.

5.1.5. Level of the accomplished performance requirements

As the performance is a major concern when an application is directly or indirectly connected with a user interface, most of the tests mentioned in the section 4.2.5 and demonstrated in the section 4.3.1, had not only the intuit of verifying the data integrity but also to measure how the API would behave when handling the client's requests. The two following sub-sections concern the user perceived performance and the performance related to the actual client application's requests and, are followed by some notes about the cacheability and scalability.

User perceived performance

As some browsers do partial rendering of content-types, such as HTML, as they are downloaded, the usage of reduced and known media-types allows browsers and other clients to handle the responses faster. As GET is idempotent and safe, the user agent can pre-fetch them before they are needed, thus improving the user perceived performance and their experience with the application.

Request performance

In a regular paradigm, when a client wants to update a resource, the resource is fully transported twice over the network. The client pulls the resource, edits the elements that it needs and then pushes it back to the service. This is no problem if the resource is light, however it can be prejudicial to the network performance and on the processing services if the resource is heavier. Even a light resource usually includes elements that the client most of the times doesn't need, like metadata or other related information. A possible solution to the previous problem is to allow clients to execute partial operations, for instance to ask for a partial response, meaning the client gets only the fields that it pretends to handle. This possibility was mentioned over certain sections of the present document and was referred as the request-refining technique. The pretended elements can be indicated by the client with the respective arguments indicated in the query parameters or in the actual request body. The request-refining technique, can also be utilized by the PATCH and PUT method's implemen-

tations in order to prevent full resource's representations to be transmitter over the network A practical example of a light request was given in the section 4.3.2, concerning the WADL retrieval.

Cacheability

Cache systems positioned in the intermediaries, allow a request to be done without actually going all the way through the network or even reach the network if the cache system is positioned within the user agent. An illustration of occurrences where it can happen is given in the figure 2.19 of the section 2.3.2. By utilizing the *Control Data* header property to signal compression, the response can be compressed (*e.g. gzip*), before it is sent to the user agents that handle them. To endorse a more reliable message passing, some control data can be included in the response, such as *Cache-control : max-age= 9800*, whose *max-age* value is the time in seconds. This particular header indicates the cache systems present on the intermediaries (*e.g.* on the proxies or on the user agents) to control how long the response can be cached.

Scalability

Gateways distribute traffic of many origin servers, by its method, URI, content-type or other headers of the request. Cacheable systems helps scalability, as it reduces the number of requests done by the user agents to the origin servers. The statelessness nature of the communication mentioned in the subsection 5.1.1, allows the request to be routed between various proxies and gateways thus avoiding bottlenecks or congestions and, allowing new intermediaries to be added as they are needed.

5.1.6. Research question overview

The main research question indicated in the section 1.3 of the introduction chapter (1) was divided in various sub-questions, which were answered throughout the present document. Concerning the primary research question, which is:

RPQ →" *Is it viable to use an Hypermedia-based REST architecture as a solution to integrate the required information systems along with the BPM engine?*"

It can be considered that the answer is yes, the designed architecture can be considered viable in the studied context due to its characteristics that are highlighted on the previous section and its respective perks that are mentioned throughout the chapter 4.

5.2. Design remarks

Some design remarks related to software development practices are highlighted to synthesize this chapter.

5.2.1. Contract and resources

The experimentation highlighted the possibility that to define and design a RESTful API, all specific technology constraints and aspects should be considered secondary issues, except for the Hypermedia itself, meaning the focus should be only on the pretended qualitative metrics, the protocol (*e.g.* HTTP), on the resources and on the media types.

The definition of the resources, the allowed interfaces in them and, each response's content and status codes, should be designed after a thorough identification of the use cases, with some different purposed clients as the possible actors. Where does the interaction start? And after getting something, what can the client ask for? These kinds of questions can lead to a resource tree diagram, such as the depicted diagram of the figure 4.8.

The use case definition should start from the simplest resource request, like a GET to the root URI. The response of this first call must be enough for the client to "know" what he can or must do next, via the links provided in it and on the service's description. Although there is always the tendency to map the resources only by its business model, so the client can smoothly and "independently" navigate in the API resources, they should also be charted like a city map, with the links metaphorically working as direction signals. This way, the interface processing machines can always know in which state (or where) they are and, to which state they can go without a strict knowledge of the service's resource hierarchy.

Also, so the clients can successfully interpret the service response, the latter's media type must be known. The HTTP content-type header property should always be defined by the service and, always consulted by the client before any parsing. These media types used in the resources representations, should follow standards whenever it is possible, like Atom or v-Card. This way:

- There is no reinventing the wheel;
- It is more probable that the clients will embrace the service due to the support of enterprise/government policies and legislations;
- It facilitates interoperability with other systems.

Besides the machine-readable service descriptor, it is always welcomed by the consumer entity a human-readable describing document. This document can help the interpretation of which resources exist, how they are structured and their possible relations or links. The contract schema and the HTTP header property *Content-Type* alone, can hamper a possible software developer, that would want to combine partial information originated from multiple resources into a local object or "client-side" representation. Also, if there is another document that targets audience beyond the IT personnel, meaning non-technical staff, it may also serve as a good service exposure and help the system's utilization to propagate and scale.

5.2.2. Application's architecture

The present section highlights some approaches that can be followed in order to answer certain programming issues of a RESTful API development.

Logical components

Similarly to the API resources, the application's components should be properly decoupled in order to allow the independent evolution of each of them. The next picture 5.1 illustrates a possible architecture with openness to the adding of new representation formats.

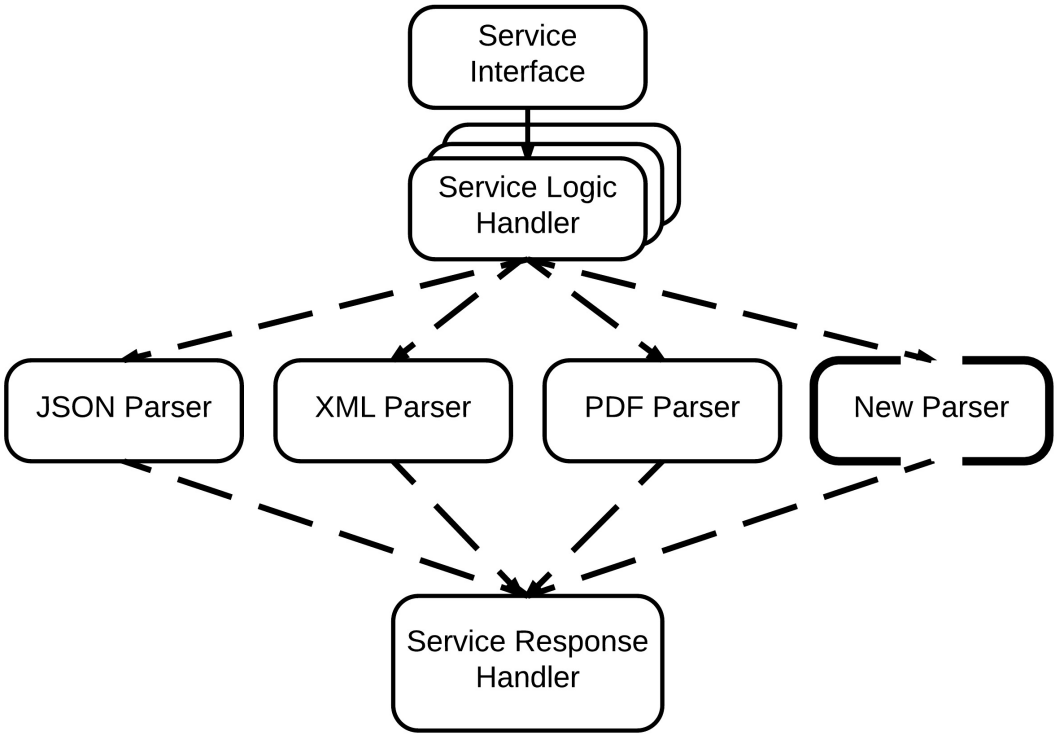

Figure 5.1.: A possible architecture with openness to the adding of new representation formats.

Methods

When some imperative instructions are needed in order to fulfill certain use cases of a RESTful API, the noun-oriented resources and the standard HTTP methods (*e.g.* POST) can seem inappropriate or, can seem that the name of the functionality cannot entirely indicate a perceptible understanding of what it is or does. These kinds of instructions can be implemented via the query parameters. The following figure 5.2 illustrates a possible alternative for an imperative instruction to finish a task of a BPM process instance.

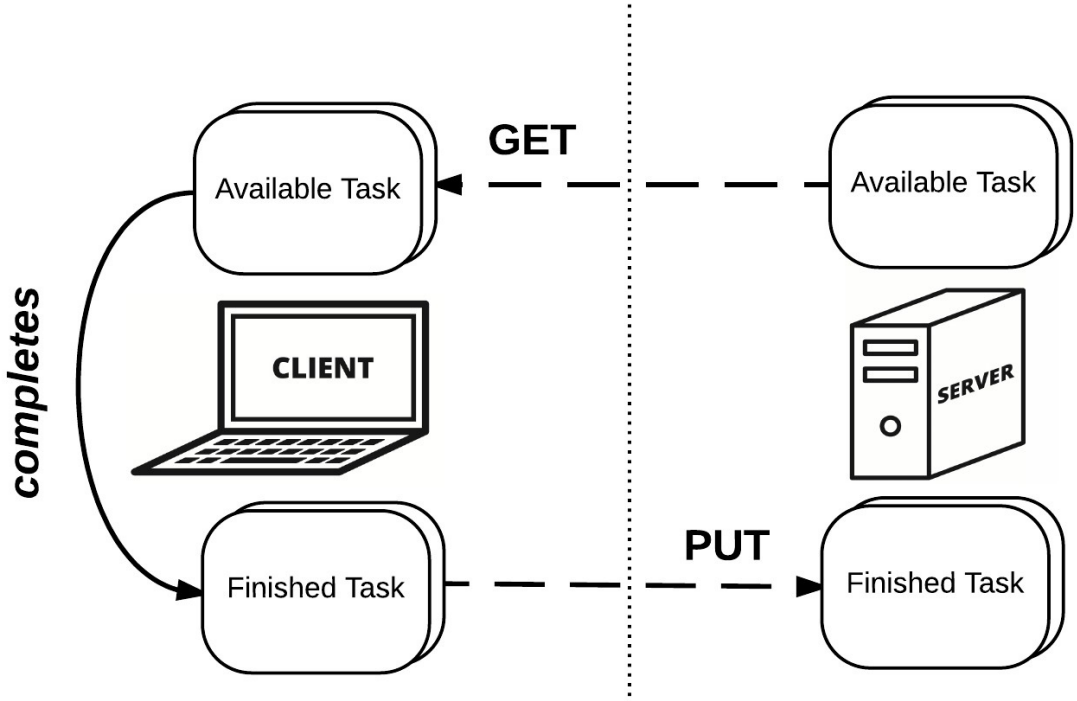

Figure 5.2.: First alternative of an imperative method mechanism.

As observable in the figure 5.2, the client obtained the *Task* resource, manipulated its representation and then submitted it back to the server. Despite the fact that the resource was updated on the server-side, the mentioned *Task* resource will still have the reference (or ID) as the same task representation that was sent to the client, because no new resource was created, its state was just altered. It can also be observable that in the illustration of the figure 5.2, the full resource's representation was transported twice over the network. The next picture 5.3 illustrates another possible alternative of an imperative method with the same intuit as the one illustrated in the figure 5.2, with the difference that the PUT method is combined with a RPC-like method indicated as a query parameter.

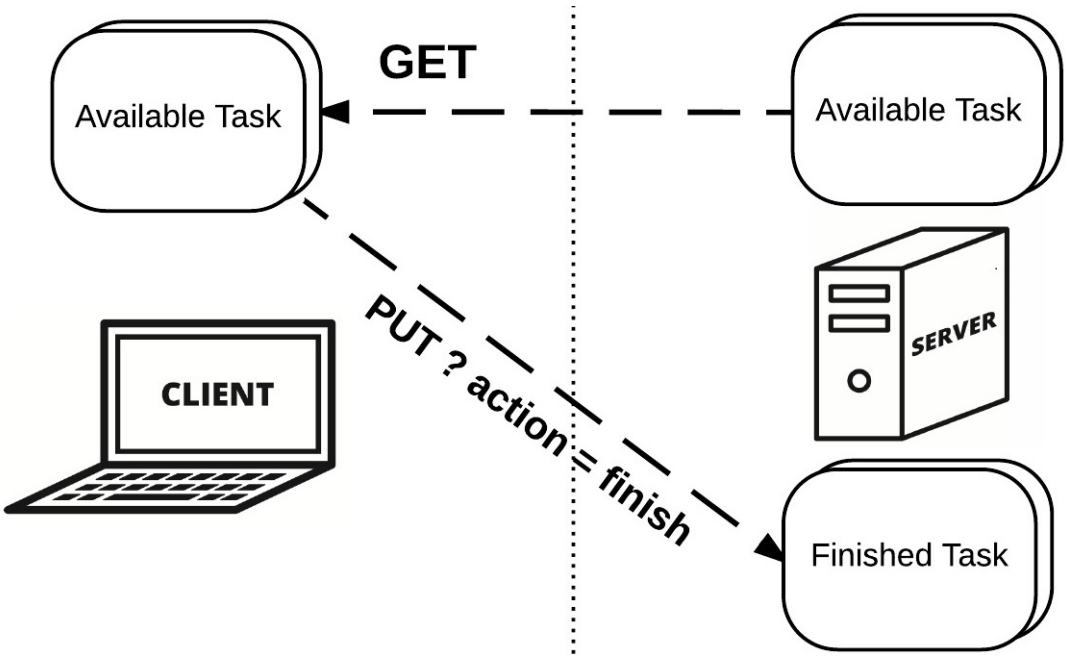

Figure 5.3.: Second alternative of an imperative method mechanism.

As demonstrated in the figure 5.3 the query parameter mechanism can be established in a REST environment and, it can work well in parallel with it when some imperative methods are required or, are more proper or intuitive to the business logic. The PATCH method (or the PUT itself) could also not carry all the resource's representation in order to fulfill the update of a representation. This alternative can also be utilized with the conditional header properties of the HTTP header (*e.g. if-match*), in order to prevent conflicting updates. These conditional header properties are briefly detailed in the section 3.4 of the present document. Such arrangement is described in the following figure 5.4, which is an updated version of the diagram illustrated in the figure 2.20 of the first state of the art chapter (2).

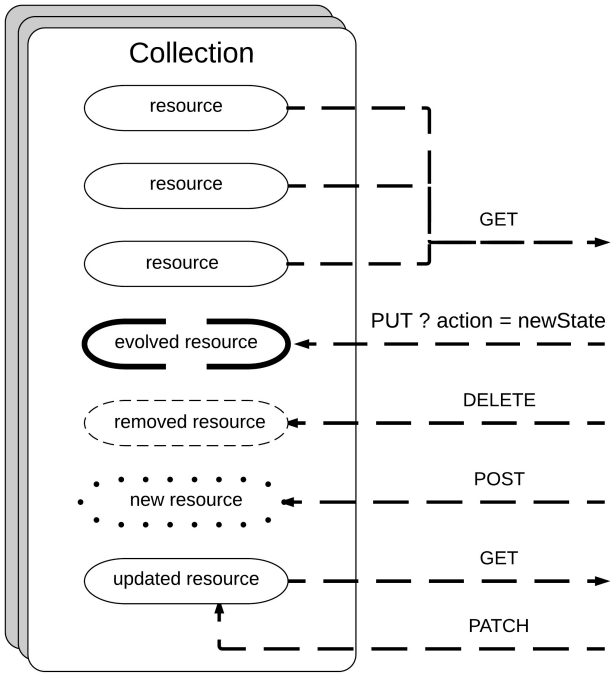

Figure 5.4.: RPC interfaces combined with the uniform interfaces of a REST environment

5.2.3. Application's security

Authentication and Authorization

The authentication should be based on the resource representation's content and on the clients attributes/properties and, not on the resource's path itself. If by some reason the resources are re-arranged in another hierarchy, it could create conflicts. After this is accomplished, the security concerns of the dialog between the server and the client should be distributed between the following three levels:

- Authenticated
  - A state in which the client has submitted valid credentials;
  - The usage of the 401 status code seems proper in situations were the client is still not in this state;
- Authorized
  - When the client has submitted valid credentials and it is utilizing them in order to ask the server to execute its due processing;
  - An adequate status code for the scenarios in which the client has not reached this state is still the 401 status code;

- Forbidden
    - A situation when a client reaches its pretended resource, but it is not allowed to do what it intended to do.
    - An adequate status code for these scenarios is the 403 status code or, the 405 if the pretended action simply doesn't exist;

Token-based authentication

The usage of token-based authentication seems more reliable and it is very utilized in public APIs for that reason. Due to the fact that the token itself is generated by secret and random data, it is surely more difficult to guess than the regular "username plus password" paradigm. The username element has become even more easy to guess due to the fact that in many systems the username is an email address, which is often of public knowledge. Even if the mechanism to decrypt the key is not a "personal guess", the applied algorithmic decryption mechanism (*e.g.* brute force), supposedly takes more time on a token than on a password, due to the entropy created by the mentioned secret and random data and, due to its resulting char sequence length that can significantly increase the time frame in which the decryption could be done. This increased time frame also coordinates well with the possible life span of the token, whose expiration leads to a new token that the decryption mechanism as to process from the beginning. Although the previous fact is true, the decryption mechanism can correlate common patterns of the token creation in order to achieve its decryption, and this is a reason not to include nor utilize exposed IDs that are sequential as it can be traceable.

5.2.4. Implementation steps

The experimented implementation highlighted some patterns and advises that could indicate a REST API development methodology. Its steps are described in the following itemization.

- Implementation of internal services and functionalities
    1. Definition of the abstract resources, its hierarchy and functionality;
    2. Definition of items, collections and their respective interfaces in a uniform manner;
- Configuration of the API composition
    1. Creation of a map-like route for the URI paths, query parameters and possible request-refining techniques that could also be utilized for necessary RPC-like methods;
    2. Improvement of the application with enterprise *QoS* features for security and other concerns, such as authentication, logging and caching;
- Definition of templates to be utilized in the outputs of the service

1. Definition of enriched data representation formats of the internal representations, to provide the required information for external entities;
2. Implementation of procedures related to data enrichment practices such as the instantiation of the Data Transfer Object pattern on related and dependent objects.

Chapter

6 Conclusions

The present chapter describes the conclusions associated with the developed study and experimentations which are documented throughout this dissertation. Starting with a detailed list of the defined objectives and its respective accomplishments, which are followed by the found limitations that could be attended during the proposed enhancing future work, which is also highlighted in the same section. Finally, a summary appreciation of the developed work is described.

6.1. Achievements

Part of the accomplished achievements are the research question's answers that resulted from the carried out research. The main researched question implied sub-questions, which are listed and answered throughout the following paragraphs.

RSQ1 → *Why there is the need to combine information systems?*
The combination of information from distinct types of data sources is needed in order to provide enriched information to third party applications or other systems, that otherwise would have to handle incomplete but related informations from various data sources.

RSQ2 → *Which environments usually surround integration solutions?*
The environments are usually categorized as the Internet, Extranet and Intranet deployment scenarios. The environments of the integrated solutions have elements that commonly consist of distinct types of data sources, such as web services, databases and third party applications or other systems.

RSQ3 ⟶ *Which models and architectures are utilized to solve system integrations?*
The client-server and peer-to-peer models are generally utilized to solve integrations of distinct or hierarchically similar elements within the same distributed system. From a another perspective, their architectures generally follow one of the two or, both of the following approaches: the first involves the presence of a component that acts as an integrator, in which all the architecture participants couple with (*e.g.* ESB) and, the second in which all the participants are decoupled and interact with each other via specific service languages and/or protocols or, via uniform and/or standard interfaces.

RSQ4 ⟶ *Which technologies and techniques can be used on architectures found in RSQ3 ?*
In the present document's context, the main technologies that can be found in the mentioned architectures are the web services. The techniques can be considered the implementation styles of the web services, meaning the style of the SOAP protocol's implementations and the REST architectural style's implementations.

RSQ5 ⟶ *Which characteristics should have the desired system?*
Some of the main characteristics that the system should have, is the capability to answer to each of the distributed system's issues that are mentioned in the section 2.2.3 (*e.g.* heterogeneity), among with other characteristics that are generally considered crucial on most softwares (*e.g. maintainability, fault tolerant and interoperable*).

RSQ6 ⟶ *Which metrics can be utilized to classify the final solution as viable?*
Due to the fact that the level of the issue's accomplishment can be volatile depending on the business requirements and, the corresponding values can be subjective and lack accuracy, the degree of the solution's capability to accomplish the categorized distributed systems issues, can be considered as qualitative or similar-to-qualitative metrics (*e.g.* grade of transparency).

Concerning the primary research question:

RPQ ⟶" *Is it viable to use an Hypermedia-based REST architecture as a solution to integrate the required information systems along with the BPM engine?*"

It can be considered that the answer is yes, the designed architecture can be considered viable in the studied context due to its characteristics and its respective advantages which are documented throughout the document. Its degree of capability to accomplish the categorized distributed systems issues are also considered fairly positive.

There are three primary research objectives highlighted in the section 1.3 and, the following itemization iterates through them while describing their accomplishments.

RPO1 → *Production of a viable architecture design;*

- The chapter 4 described the type of the deployment (*i.e.* an intranet scenario), the followed architectural patterns (*e.g.* SOA) and some of the specific elements of the REST technical implementations (*e.g.* dynamic content negotiation);
    - The chapter 5 highlights some of the researched advantages that are documented in the chapter 3 and, that were experimented and documented in the chapter 4;
- The section 5.2 describes additional notes on the development perspective of the architecture design.

RPO2 → *Execution of tests on an instantiation of the developed architecture design;*

- The planned and carried out tests on the instantiation of the developed architecture design are documented on the sections 4.2.5 and 4.3.1, and consisted of:
    1. unit tests, whose environment was local and whose targets were the POJOs themselves and the class methods;
    2. resource interface tests, which were done via the *Advanced REST Client* (a browser's extension);
    3. global tests, which were carried out via the developed *API tester app*, whose purpose was to test the conciliation of the related resources and the actual overall performance of the designed architecture.

RPO3 → *Documentation of design remarks associated with the development of REST APIs;*

- There are three categories of documentation that were developed during the study:
    1. a background study whose notes are present on the chapter 2;
    2. the overview of the chosen architectural style, which is documented on the chapter 3;
    3. the design remarks themselves, which are described on the chapter 4 and on the section 5.2 of the chapter 5.

6.2. Limitations and future work

A particular good enhancement that could be done was a mechanism to define the routes of the resources, by dynamically defining their *paths* of the corresponding URIs. This mechanism could be made via database configurations, external files or, via the API itself, in which, a new

resource could be added into the same level of the *Version* resources (observable in the figure 4.8), in order to expose a resource representation whose functionalities would be the analysis of the structure and the restructure of the respective resource version.

The experimented token based authentication mechanism was custom made, a possible enhancement would be a experimentation and integration of the *OAuth 2.0* framework (mentioned in the section 3.6) in order to solve the authorization features.

6.3. Final appreciation

Over the years, web services have become very popular and enhanced. Some of them are even discreetly present on our daily routine, such the toll payments and weather forecast news. The study carried out during the development of the present dissertation, clarified some of their aspects and enlightened some of the technical concerns of the chosen web service's implementation style and architectural style.

Bibliography

Open Web Application Security Project – *Testing for CSRF*. OWASP Testing Project, 2015 ⟨URL: https://www.owasp.org/index.php/Testing_for_CSRF_(OTG-SESS-005)⟩ – Last visited in March, 2015

Abrams, Marc et al. – *Caching Proxies: Limitations and Potentials*. MIT Computer Science and Artificial Intelligence Laboratory, OSF Research Institute and World Wide Web Consortium, 1995, Presented in the Fourth International World Wide Web Conference

Akamai – *Akamai's Q1 2015 Internet Security Report*. 2015 ⟨URL: https://www.stateoftheinternet.com/downloads/pdfs/2015-internet-security-report-q1.pdf⟩ – Last visited in May, 2015

Amazon – *Amazon SQS Client Libraries for Java Messaging Service*. Amazon Web Services, 2014 ⟨URL: https://aws.amazon.com/about-aws/whats-new/2014/12/29/amazon-sqs-client-libraries-for-java-messaging-service-now-available/⟩, Last visited in August, 2015

Apache Software Foundation – *Class PatternLayout*. API Docs, 2012 ⟨URL: https://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/PatternLayout.html⟩, Last visited in July, 2015

—— *WSDL 1.1 Extensions for REST*. 2014 ⟨URL: http://ode.apache.org/extensions/wsdl-11-extensions-for-rest.html⟩ – Last visited in May, 2015

—— *Apache Tomcat 8 Configuration Reference*. 2015 ⟨URL: https://tomcat.apache.org/tomcat-8.0-doc/config/filter.html⟩ – Last visited in June, 2015

API Academy – *Web API Architectural Styles*. 2012 ⟨URL: http://www.apiacademy.co/lessons/api-design/web-api-architectural-styles⟩ – Last visited in May, 2015

Arbor Networks, Inc – *Arbor Networks 10th Annual Worldwide Infrastructure Security Report*. 2015 ⟨URL: http://www.arbornetworks.

com/news-and-events/press-releases/recent-press-releases/5351-arbor-networks-10th-annual-worldwide-infrastructure-security-report\-finds-50x-increase-in-ddos-attack-size-in-past-decade⟩ – Last visited in May, 2015

Arsanjani, Ali – *Service-oriented modeling and architecture.* IBM developerWorks - SOA and web services, 2004 ⟨URL: http://www.ibm.com/developerworks/library/ws-soa-design1/⟩, Section - An architectural template for a SOA

Baccala, Brent – *Connected: An Internet Encyclopedia.* FreeSoft, 1997, p. 652 ⟨URL: http://www.freesoft.org/CIE/Topics/102.htm⟩ – Last visited in March, 2015, HTTP Protocol Overview

Baker, Loek et al. – *Integration Architecture With BizTalk 2004.* SYS-CON Media - .NET Developer's Journal, 2005 ⟨URL: http://dotnet.sys-con.com/node/121831⟩

Ballinger, Keith et al. – *Basic Profile Version 1.1.* Web Services-Interoperability Organization, 2006

Banzal, Shashi – *Data and Computer Network Communication.* Firewall Media, 2007, p. 652, Application Layer - HTTP Protocol Overview

Benatallah, Boualem et al. – *Web Engineering: 10th International Conference.* Springer, 2010, p. 144

Berners-Lee, T.; Fielding, R.; Frystyk, H. – *Hypertext Transfer Protocol – HTTP/1.0.* Network Working Group, 1996, p. 48 ⟨URL: http://tools.ietf.org/html/rfc1945#section-11⟩

Berners-Lee, T.; Fielding, R.; Masinter, L. – *Uniform Resource Identifier (URI): Generic Syntax.* Network Working Group, 2005, p. 15 ⟨URL: https://www.ietf.org/rfc/rfc3986.txt⟩

Booth, David et al. – *Web Services Architecture.* 2004 ⟨URL: http://www.w3.org/TR/ws-arch/wsa.pdf⟩ – Last visited in May, 2015

Borghoff, Uwe M.; Schlichter, Johann H. – *Computer-Supported Cooperative Work: Introduction to Distributed Applications.* Springer, 2000

Bovet, Daniel P.; Cesati, Marco – *Understanding the Linux Kernel.* O'Reilly, 2006

Box, Don et al. – *Simple Object Access Protocol (SOAP) 1.1.* 2000 ⟨URL: http://www.w3.org/TR/2000/NOTE-SOAP-20000508/#_Toc478383497⟩ – Last visited in February, 2015

Bragança, Alexandre – *Introduction to SOAP Web Services: WCF and JAX-WS.* ISEP/IPP, 2012

Brose, Gerald – *Web Services Security with SOAP Security Proxies.* OMG Web Services Workshop USA, 2009

Campbell, B.; Mortimore, C.; Jones, M. – *Security Assertion Markup Language (SAML) 2.0 Profile for OAuth 2.0 Client Authentication and Authorization Grants.* 2015 ⟨URL: `https://tools.ietf.org/html/rfc7522`⟩ – Last visited in June, 2015

Carballeira, Félix García – *Llamadas a procedimientos remotos.* Universidad Carlos III de Madrid, 2015 ⟨URL: `http://www.arcos.inf.uc3m.es/~infosd/lib/exe/fetch.php?media=es:tema6\_llamadas\_a\_procedimientos\_remotos.pdf`⟩

Chen, Shuo et al. – *Side-Channel Leaks in Web Applications: a Reality Today, a Challenge Tomorrow.* IEEE Symposium on Security & Privacy, 2010 ⟨URL: `http://research.microsoft.com/pubs/119060/WebAppSideChannel-final.pdf`⟩

Christensen, Erik et al. – *Web Services Description Language (WSDL) 1.0.* Ariba, International Business Machines Corporation, Microsoft, 2000, p. 1 ⟨URL: `http://xml.coverpages.org/wsdl20000929.html`⟩

—— *Web Services Description Language (WSDL) 1.1.* Ariba, International Business Machines Corporation, Microsoft, 2001, p. 1 ⟨URL: `http://www.w3.org/TR/wsdl`⟩

Clark, Kim; Flurry, Greg – *The Enterprise Service Bus, re-examined.* IBM developerWorks - Industrial SOA article series, 2011 ⟨URL: `http://www.ibm.com/developerworks/websphere/techjournal/1105_flurry/1105_flurry.html`⟩, Section - The ESB in SOA

Clark, Kim J. – *Integration architecture: Comparing web APIs with service-oriented architecture and enterprise application integration.* IBM developerWorks, 2015 ⟨URL: `http://www.ibm.com/developerworks/websphere/library/techarticles/1503_clark/1305_clark.html`⟩

Delgado, Nelly et al. – *Microsoft ESB Guidance for BizTalk Server 2006 R2.* Microsoft patterns and practices, 2007 ⟨URL: `https://msdn.microsoft.com/en-us/library/ff647678.aspx`⟩, Section - Common Scenarios

Eles, Petru – *Models of Distributed Systems.* Linköping University, 2007, p. 3

Eliassen, Frank – *System models for distributed systems.* University of Oslo, 2011, p. 4

Endrei, Mark; Keen, Martin; Sadtler, Carla – *Application Integration patterns.* IBM Redbooks Paper, 2004, p. 6, Section - Application Integration considerations

Evdemon, John – *The Four Tenets of Service Orientation.* SOA Institute, 2015 ⟨URL: `http://www.soainstitute.org/resources/articles/four-tenets-service-orientation`⟩ – Last visited in May, 2015

FACEBOOK, INC. – *The Graph API*. 2015 ⟨URL: https://developers.facebook.com/docs/graph-api⟩ – Last visited in May, 2015

FIELDING, R. et al. – *Hypertext Transfer Protocol – HTTP/1.1*. 2004 ⟨URL: http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html⟩ – Last visited in June, 2015

—— *Hypertext Transfer Protocol – HTTP/1.1*. 2004 ⟨URL: http://www.w3.org/Protocols/rfc2616/rfc2616-sec6.html⟩ – Last visited in June, 2015

—— ; RESCHKE, J. – *Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests*. 2014 ⟨URL: http://tools.ietf.org/html/rfc7232\#page-7⟩ – Last visited in June, 2015

FIELDING, Roy Thomas – *Architectural Styles and the Design of Network-based Software Architectures*. Ph. D thesis, University of California, Irvine, 2000, ⟨URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm\#sec_5_2⟩

—— *Architectural Styles and the Design of Network-based Software Architectures*. Ph. D thesis, University of California, Irvine, 2000, ⟨URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm⟩, Section 5.1.5 Uniform Interface

—— *Architectural Styles and the Design of Network-based Software Architectures*. Ph. D thesis, University of California, Irvine, 2000, ⟨URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm⟩, 5.1.7 Code-On-Demand

FLURRY, Greg – *Exploring the Enterprise Service Bus*. IBM developerWorks - SOA and web services, 2007 ⟨URL: http://www.ibm.com/developerworks/webservices/library/ar-esbpat1/index.html⟩, Section - The ESB in SOA

FOWLER, Martin – *Richardson Maturity Model - steps toward the glory of REST*. 2010 ⟨URL: http://martinfowler.com/articles/richardsonMaturityModel.html⟩ – Last visited in June, 2015

FRANKS, J. et al. – *HTTP Authentication: Basic and Digest Access Authentication*. 1999 ⟨URL: http://tools.ietf.org/html/rfc2617\#section-2⟩ – Last visited in June, 2015

FREEBSD FOUNDATION – *FreeBSD Handbook*. 2015 ⟨URL: http://www.freebsd.org/about.html⟩ – Last visited in April de 2015

FREED, N.; BORENSTEIN, N. – *Multipurpose Internet Mail Extensions*. 1996 ⟨URL: https://tools.ietf.org/html/rfc2046⟩ – Last visited in May, 2015

GALPERIN, Eva; SCHOEN, Seth; ECKERSLEY, Peter – *A Post Mortem on the Iranian DigiNotar Attack*. Electronic Frontier Foundation, 9 2011, p. 1 ⟨URL: https://www.eff.org/deeplinks/2011/09/post-mortem-iranian-diginotar-attack⟩

Gao, Shudi et al. – *W3C XML Schema Definition Language (XSD) 1.1*. 2012 ⟨URL: http://www.w3.org/TR/xmlschema11-1/⟩ – Last visited in February, 2015

George, Alex – *Remote Procedure Call (RPC) - Session Layer protocol*. 2009 ⟨URL: http://www.corenetworkz.com/2009/02/remote-procedure-call-rpc-session-layer.html⟩ – Last visited in May, 2015

Ghosh, Neil – *Intro to web services*. 2011 ⟨URL: http://www.slideshare.net/neilghosh/intro-to-web-services⟩ – Last visited in May, 2015

Ghosh, Sukumar – *Altruistic Routing in P2P Networks: Solutions and Problems*. University of Iowa, 2006, p. 10, Section "The Model"

Google – *API Client Library for Java*. Google Developers, 2015 ⟨URL: https://developers.google.com/api-client-library/java/google-api-java-client/setup⟩, Last visited in August, 2015

Hadley, Marc – *Web Application Description Language*. 2009 ⟨URL: http://www.w3.org/Submission/wadl/⟩ – Last visited in May, 2015

—— *Web Application Description Language*. 2009 ⟨URL: http://www.w3.org/Submission/wadl/\#x3-30001.2⟩ – Last visited in June, 2015

Hafif, Oren – *Reflected File Download a New Web Attack Vector*. Trustwave's SpiderLabs, 2014

Hardt, D. – *The OAuth 2.0 Authorization Framework*. 2012 ⟨URL: https://tools.ietf.org/html/rfc6749⟩ – Last visited in June, 2015

Harris, Jim et al. – *Electronic Court Filing 4.0 Web Services Service Interaction Profile Version 2.0*. 2008 ⟨URL: http://docs.oasis-open.org/legalxml-courtfiling/specs/ecf/v4.0/ecf-v4.0-webservices-spec/ecf-v4.0-webservices-v2.0-spec.html⟩ – Last visited in February, 2015

Hefeeda, Mohamed; Hsu, Cheng-Hsin; Mokhtarian, Kianoosh – *Design and Evaluation of a Proxy Cache for Peer-to-Peer Traffic*. Institute of Electrical and Electronics Engineers, 2011, p. 1

Hevner, Alan R. et al. – *Design Science In Information Systems Research*. 2004 ⟨URL: http://wise.vub.ac.be/thesis_info/design_science.pdf⟩ – Last visited in February, 2015

Hohpe, Gregor; Woolf, Bobby – *Messaging Patterns - Aggregator*. Enterprise Integration Patterns, 2003 ⟨URL: http://www.enterpriseintegrationpatterns.com/patterns/messaging/Aggregator.html⟩, Last visited in July, 2015

Hohpe, Gregor; Woolf, Bobby – *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2004, p. 55, Chapter 2 - Integration Styles, Section Messaging

Hope, Paco; Walther, Ben – *Web Security Testing Cookbook - Systematic Techniques to Find Problems Fast*. O'Reilly Media, 2008

Hudson, Scott – *Success with Hub and Spoke Distribution*. NC State University 2003

Iyengar, Sridhar – *Web Services for the Integrated Enterprise*. OMG Web Services Workshop Europe, 2009

Jackson, Paul; Anderson, Stuart – *Seven Principles of Software Engineering*. University of Edinburgh, 2014 ⟨URL: `http://www.inf.ed.ac.uk/teaching/courses/inf2c-se/Lectures/Lectures-2014/lecture-14-SEprinciples.pdf`⟩

Jendrock, Eric et al. – *The Java EE 6 Tutorial*. Oracle, 2013, Chapter 20 - "Building RESTful Web Services with JAX-RS"

—— et al. – *The Java EE 7 Tutorial : Volume 2*. Oracle Corporation, 2014, p. 545

Jiang, Chunqi – *Why Some Theories Fail to Describe, Explain, and Predict: Reconstructing the Future*. 1998 ⟨URL: `http://naulibrary.org/dglibrary/admin/book\_directory/Philosophy/4880.pdf`⟩ – Last visited in February, 2015

Josefsson, S. – *The Base16, Base32, and Base64 Data Encodings*. The Internet Society, 2006, p. 5 ⟨URL: `https://tools.ietf.org/html/rfc4648`⟩

Kalin, Martin – *Java Web Services: Up and Running*. O'Reilly Media, 2009

Kanjilal, Joydip – *An Introduction to Simple Object Access Protocol*. ASPAlliance, 2006 ⟨URL: `http://aspalliance.com/1064_An_Introduction_to_Simple_Object_Access_Protocol.all`⟩, Section 5 - Disadvantages of SOAP

Kesteren, Anne van – *Cross-Origin Resource Sharing*. 2014 ⟨URL: `http://www.w3.org/TR/cors/`⟩ – Last visited in June, 2015

Khan, Jahanzeb; Khwaja, Anis – *Building Secure Wireless Networks with 802.11*. John Wiley and Sons, 2003, p. 41, Chapter 2; Section - Wireless LANs

Kress, Jürgen et al. – *Enterprise Service Bus*. Oracle Technology Network - Industrial SOA article series, 2013 ⟨URL: `http://www.oracle.com/technetwork/articles/soa/ind-soa-esb-1967705.html`⟩

LEAR, E. et al. – *A Simple Authentication and Security Layer (SASL) and Generic Security Service Application Program Interface (GSS-API) Mechanism for OpenID*. 2012 ⟨URL: https://tools.ietf.org/html/rfc6616⟩ – Last visited in June, 2015

LEHMANN, Tobias – *A Framework for Ontology based Integration of Structured IT-Systems*. Bundeswehr University Munich, 2007, p. 2

LINUX KERNEL ORGANIZATION INC – *Sysctl documentation*. 2015 ⟨URL: https://www.kernel.org/doc/Documentation/networking/ip-sysctl.txt⟩ – Last visited in April, 2015

LOCKHART, Hal – *Web Services Security Challenges*. 2003 ⟨URL: http://www.omg.org/news/meetings/workshops/MDA-SOA-WS_Manual/06-1_Lockhart.pdf⟩ – Last visited in June, 2015

MARCHAL, Benoit – *Tip: SOAP 1.2 and the GET request - A first look at the Response MEP*. Pineapplesoft/IBM developerWorks, 2004 ⟨URL: http://www.ibm.com/developerworks/library/x-tipgetr/⟩ – Last visited in June, 2015

MARKS, Eric A.; LOZANO, Bob – *Executive's Guide to Cloud Computing*. John Wiley and Sons, 2000

MARSHALL, James – *HTTP Made Really Easy - A Practical Guide to Writing Clients and Servers*. 2012 ⟨URL: http://www.freesoft.org/CIE/Topics/102.htm⟩ – Last visited in March, 2015, HTTP Protocol Overview

MCGUIRE, Scott – *BSD Sockets API*.

MITRA, Nilo; LAFON, Yves – *SOAP Version 1.2 Part 0: Primer*. W3C, 2007 ⟨URL: http://www.w3.org/TR/soap12-part0/⟩

MORGAN, Gareth – *Data virtualisation on rise as ETL alternative for data integration*. ComputerWeekly, 2012 ⟨URL: http://www.computerweekly.com/feature/Data-virtualisation-on-rise-as-ETL-alternative-for-data-integration⟩, Last visited in July, 2015

NAKAMOTO, Satoshi – *Bitcoin: A Peer-to-Peer Electronic Cash System*. Satoshi Nakamoto Institute, 2008 ⟨URL: http://nakamotoinstitute.org/bitcoin/⟩

NOTTINGHAM, Mark – *Caching Tutorial*. 2013 ⟨URL: https://www.mnot.net/cache_docs/⟩ – Last visited in March, 2015

; RESCHKE, Julian; ALGERMISSEN, Jan – *RFC5988 - Link Relations*. 2015 ⟨URL: http://www.iana.org/assignments/link-relations/link-relations.xml⟩ – Last visited in June, 2015

OASIS – *Introduction to UDDI: Important Features and Functional Concepts*. 2004 ⟨URL: `http://uddi.xml.org/files/uddi-tech-wp.pdf`⟩ – Last visited in February, 2015

OASIS – *SOAP-over-UDP Version 1.1*. 2009 ⟨URL: `http://docs.oasis-open.org/ws-dd/soapoverudp/1.1/os/wsdd-soapoverudp-1.1-spec-os.html`⟩ – Last visited in May, 2015

Oracle – *The Java EE 6 Tutorial*. Oracle Corporation, 2013, p. 367 ⟨URL: `http://docs.oracle.com/javaee/6/tutorial/doc/javaeetutorial6.pdf`⟩

Oracle Corporation – *Jersey 2.22 User Guide - Chapter 16 : Security*. 2015 ⟨URL: `https://jersey.java.net/documentation/latest/security.html`⟩ – Last visited in June, 2015

Papazoglou, Michael – *Web Services: Principles and Technology*. Tilburg University, 2008 ⟨URL: `http://www.cs.colorado.edu/\~kena/classes/7818/f08/lectures/lecture_3_soap.pdf`⟩, Chapter 4 - SOAP: Simple Object Access Protocol

Park, Jong Hyuk; Chen, Hsiao-Hwa; Atiquzzaman, Mohammed – *Advances in Information Security and Assurance: Third International Conference and Workshops*. Springer, 2009, p. 269

PayPal – *API Version History*. 2014 ⟨URL: `https://developer.paypal.com/docs/classic/release-notes/`⟩ – Last visited in May, 2015

Peffers, Ken et al. – *A Design Science Research Methodology for Information Systems Research*. Journal of Management Information Systems, 2008 ⟨URL: `http://wise.vub.ac.be/thesis\_info/Design\_Science\_Research\_Methodology\_2008.pdf`⟩

Perkins, Colin – *Network Programming in C: The Berkeley Sockets API*. 2015 ⟨URL: `https://csperkins.org/teaching/ns3/labs-intro.pdf`⟩ – Last visited in April, 2015

Perreault, S. – *vCard Format Specification*. 2011 ⟨URL: `https://tools.ietf.org/html/rfc6350`⟩ – Last visited in June, 2015

Presidência do Conselho de Ministros – *Resolução do Conselho Ministros nº 91 - 2012*. Diário da República, 2012

Richardson, Leonard – *Justice Will Take Us Millions Of Intricate Moves*. Ph. D thesis, QCon, 2008, ⟨URL: `http://www.crummy.com/writing/speaking/2008-QCon/act3.html`⟩ – Last visited in June, 2015

Rodriguez, Alex – *RESTful Web services: The basics*. IBM, 2015, Section 1 - "The basics"

Rouse, Margaret – *Remote Procedure Call (RPC)*. 2015 ⟨URL: `http://searchsoa.techtarget.com/definition/Remote-Procedure-Call`⟩ – Last visited in April de 2009

Sabbouh, Marwan et al. – *Workshop on Web services*. 2011 ⟨URL: http://www.w3.org/2001/03/WSWS-popa/paper08⟩ – Last visited in May, 2015

Sandakith, Lahiru – *Eclipse WTP Tutorials - Creating Top Down Web Service via Apache Axis2*. WSO2 Inc. 2007 ⟨URL: http://www.eclipse.org/webtools/community/tutorials/TopDownAxis2WebService/td_tutorial.html⟩ – Last visited in March, 2015

Sauter, Guenter et al. – *Data federation pattern*. IBM developerWorks - Information service patterns, 2012 ⟨URL: http://www.ibm.com/developerworks/library/ws-soa-infoserv1/⟩, Last visited in July, 2015

Schlichter, Johann – *Distributed Applications*. 2002 ⟨URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.471.7987\&rep=rep1\&type=pdf⟩ – Last visited in May, 2015

Schmidt, Douglas C. – *Overview of Remote Procedure Calls (RPC)*. 2012 ⟨URL: https://inst.eecs.berkeley.edu/~ee122/sp06/LectureNotes/Socket%20Programmingprint.pdf⟩ – Last visited in April, 2015

Schneier, Bruce – *The Process of Security*. TechTarget, 2000 ⟨URL: https://www.schneier.com/essays/archives/2000/04/the_process_of_secur.html⟩ – Last visited in March, 2015

Shetty, Nikhil – *Socket Programming*. 2007 ⟨URL: https://inst.eecs.berkeley.edu/~ee122/sp06/LectureNotes/Socket%20Programmingprint.pdf⟩ – Last visited in April, 2015

Singh, Magan Pal – *Web Services Security - Challenges and Trends*. 2008 ⟨URL: https://www.owasp.org/images/3/33/Web_Services_Security_-_Challenges_and_Trends.ppt⟩ – Last visited in June, 2015

Slack – *Slack API Documentation*. 2015 ⟨URL: https://api.slack.com/methods⟩ – Last visited in June, 2015

Sousa, Paulo Gandra – *Models of distributed computing*. ISEP/IPP, 2014

—— *Programação de Sistemas Distribuídos - REST web services*. ISEP/IPP, 2015a

—— *Programação de Sistemas Distribuidos - Communication*. ISEP/IPP, 2015b, p. 46

—— *Programação de Sistemas Distribuidos - Decoupled communication*. ISEP/IPP, 2015c, p. 20

—— *Programação de Sistemas Distribuidos - Principles of service design*. ISEP/IPP, 2015d

The Eclipse Foundation – *Service-oriented architecture (SOA) for EGL developers*. EGL Development Tools - Technology Papers, 2009 ⟨URL: `https://www.eclipse.org/edt/papers/topics/egl\_soa\_overview.html`⟩, Last visited in June, 2015

Thijssen, Joshua – *The RESTful CookBook*. 2015 ⟨URL: `http://restcookbook.com/Basics/hateoas/`⟩ – Last visited in May, 2015

Tidwell, Doug; Snell, James; Kulchenko, Pavel – *Programming Web Services with SOAP*. O'Reilly, 2001

Vedamuthu, Asir S et al. – *Web Services Policy 1.5 - Framework*. W3C, 2007 ⟨URL: `http://www.w3.org/TR/ws-policy/`⟩

Villa, Matteo et al. – *SOAP HTTP Binding Status*. European Commission, Joint Research Centre, Institute for Environment and Sustainability, 2008 ⟨URL: `http://inspire.ec.europa.eu/reports/ImplementingRules/network/SOAP_binding_survey.pdf`⟩ – Last visited in June, 2015

Woolf, Bobby – *WebSphere SOA and JEE in Practice*. IBM developerWorks - SOA and web services, 2006 ⟨URL: `https://www.ibm.com/developerworks/community/blogs/woolf/entry/composite_services?lang=en`⟩, Section - Composite Services

XyzWS – *SOAP: Simple Object Access Protocol*. SCDJWS Study Guide: SOAP, 2015 ⟨URL: `http://www.xyzws.com/scdjws/SGS22/8`⟩, Section 8 - The Advantages and Disadvantages of Using SOAP Messages

Appendix

A Resolução do Conselho de Ministros 91-2012

PRESIDÊNCIA DO CONSELHO DE MINISTROS

Resolução do Conselho de Ministros n.º 91/2012

A Lei n.º 36/2011, de 21 de junho, que estabelece a adoção de normas abertas nos sistemas informáticos do Estado, atribui à Agência de Modernização Administrativa, I. P., a elaboração do Regulamento Nacional de Interoperabilidade Digital, doravante designado por Regulamento, a aprovar por resolução do Conselho de Ministros. De acordo com a referida lei, este Regulamento define as especificações técnicas e formatos digitais a adotar pela Administração Pública.

A utilização de formatos abertos (não proprietários) é imprescindível para assegurar a interoperabilidade técnica e semântica, em termos globais, dentro da Administração Pública, na interação com o cidadão ou a empresa e para disponibilização de conteúdos e serviços, criando a necessária independência dos fornecedores ou soluções de *software* adotadas. O Regulamento, alinhado com as diretrizes europeias em termos de interoperabilidade, contribui para a universalidade de acesso e utilização da informação, para a preservação dos documentos eletrónicos e para uma redução de custos de licenciamento de *software*.

Em cumprimento do disposto no n.º 4 do artigo 5.º da Lei n.º 36/2011, de 21 de junho, as matérias abrangidas pelo Regulamento foram sujeitas a discussão pública, tendo sido tomados em consideração, na sua seleção e classificação de obrigatoriedade, os contributos e resultados da mesma.

O Regulamento aprovado pela presente resolução assenta prioritariamente em especificações técnicas e formatos digitais definidos e mantidos por organismos internacionais e está dividido em especificações técnicas e

formatos digitais obrigatórios e recomendados, sendo que o incumprimento das especificações técnicas e formatos digitais obrigatórios tem, para fins de contratação pública, as consequências previstas no artigo 9.º da Lei n.º 36/2011, de 21 de junho, e as especificações técnicas e formatos digitais recomendados são orientações que constituem boas práticas que devem ser aplicadas sempre que possível.

O conceito de «especificações técnicas» adotado no âmbito da presente resolução corresponde à definição prevista na subalínea *i*) da alínea *c*) do artigo 2.º do Decreto-Lei n.º 58/2000, de 18 de abril, bem como no n.º 4 do artigo 2.º do Regulamento PE-CONS 32/12, distinguindo-se do conceito de «especificações técnicas» estabelecido no artigo 49.º do Código dos Contratos Públicos, aprovado pelo Decreto-Lei n.º 18/2008, de 29 de janeiro.

Assim:

Nos termos do disposto no n.º 6 do artigo 5.º da Lei n.º 36/2011, de 21 de junho, e da alínea *g*) do artigo 199.º da Constituição, o Conselho de Ministros resolve:

1 — Aprovar o Regulamento Nacional de Interoperabilidade Digital, doravante designado por Regulamento, constante do anexo à presente resolução e da qual faz parte integrante.

2 — Estabelecer que as entidades, serviços e organismos abrangidos pelo âmbito de aplicação do Regulamento estão obrigados a cumprir as especificações técnicas e formatos digitais obrigatórios e a procurar seguir as especificações técnicas e formatos digitais recomendados de acordo com a respetiva classificação, nos termos definidos na Lei n.º 36/2011, de 21 de junho.

3 — Determinar que a implementação, licenciamento ou evolução de sistemas informáticos tem obrigatoriamente de considerar o disposto no Regulamento, em cumprimento do disposto no n.º 1 do artigo 4.º da Lei n.º 36/2011, de 21 de junho.

4 — Estabelecer que o disposto no número anterior não prejudica a aplicação das condições de exceção, em caso de impossibilidade da utilização das especificações técnicas e formatos digitais previstos no Regulamento, em cumprimento do estatuído no artigo 6.º da Lei n.º 36/2011, de 21 de junho, nela se incluindo as situações em que, fundamentadamente, se comprove que da aplicação do Regulamento resulta um aumento de encargos para o caso em concreto.

5 — Determinar que o Regulamento agora aprovado deve ser revisto num prazo máximo de três anos, sem prejuízo de alterações técnicas pontuais às tabelas que o integram, que são aprovadas pelo membro do Governo responsável pela tutela da Agência para a Modernização Administrativa, I. P., sob proposta desta entidade.

6 — Determinar que a presente resolução produz efeitos 90 dias após a sua publicação.

Presidência do Conselho de Ministros, 25 de outubro de 2012. — O Primeiro-Ministro, *Pedro Passos Coelho.*

ANEXO

REGULAMENTO NACIONAL DE INTEROPERABILIDADE DIGITAL (RNID)

1 — O Regulamento Nacional de Interoperabilidade Digital, doravante designado RNID, define as especificações técnicas e formatos digitais, doravante e abreviadamente designados de especificações técnicas, a adotar pela Administração Pública, nos termos previstos na Lei n.º 36/2011, de 21 de junho.

2 — As especificações técnicas agora adotadas e regulamentadas cumprem os requisitos estabelecidos no n.º 1 do artigo 3.º da Lei n.º 36/2011, de 21 de junho, e estão alinhados com orientações europeias e internacionais.

3 — O RNID aplica-se aos órgãos, serviços e demais entidades previstas no artigo 2.º da Lei n.º 36/2011, de 21 de junho.

4 — O RNID abrange os seguintes domínios:

a) Formatos de dados, incluindo códigos de carateres, formatos de som e imagens (fixas e animadas), audiovisuais, dados gráficos e de pré-impressão (tabela I);

b) Formatos de documentos (estruturados e não estruturados) e gestão de conteúdos, incluindo gestão documental (tabela II);

c) Tecnologias de interface *web,* incluindo acessibilidade, ergonomia, compatibilidade e integração de serviços (tabela III);

d) Protocolos de *streaming* ou transmissão de som e imagens animadas em tempo real, incluindo o transporte e distribuição de conteúdos e os serviços ponto a ponto (tabela IV);

e) Protocolos de correio eletrónico, incluindo acesso a conteúdos e extensões e serviços de mensagem instantânea (tabela V);

f) Sistemas de informação geográfica, incluindo cartografia, cadastro digital, topografia e modelação (tabela VI);

g) Especificações técnicas e protocolos de comunicação em redes informáticas (tabela VII);

h) Especificações técnicas de segurança para redes, serviços, aplicações e documentos (tabela VIII);

i) Especificações técnicas e protocolos de integração, troca de dados e orquestração de processos de negócio na integração interorganismos (tabela IX).

5 — As especificações técnicas e formatos digitais adotados pelo presente Regulamento, classificam-se como «obrigatório» ou «recomendado», cuja aplicação se define nos seguintes termos:

a) Especificações técnicas classificadas de «obrigatório» — são especificações técnicas cuja aplicação é obrigatória por parte das entidades abrangidas pelo presente Regulamento, em todos os processos de implementação, licenciamento ou evolução de sistemas informáticos, resultando nulos e de nenhum efeito todo e qualquer ato de contratação, nos termos do artigo 9.º da Lei n.º 36/2011, de 21 de junho, com exceção dos processos excecionados nos termos do artigo 6.º da mesma lei;

b) Especificações técnicas classificadas de «recomendado» — são especificações técnicas com caráter de orientação que constituem boas práticas a serem adotadas sempre que possível por parte das entidades abrangidas pelo presente Regulamento, nos processos de implementação, licenciamento ou evolução de sistemas informáticos.

6 — As versões mais recentes das especificações técnicas constantes no presente Regulamento e classificadas como obrigatórias, são suscetíveis de serem adotadas, desde que retrocompatíveis com a versão constante no Regulamento, ou sejam disponibilizadas as duas versões, desde que tal seja possível.

7 — São ainda classificados como «recomendado» versões posteriores das especificações técnicas e formatos digitais definidos nas tabelas I a IX.

8 — O RNID aplica-se nos termos previstos no artigo 4.º da Lei n.º 36/2011, de 21 de junho, a «todos os processos de implementação, licenciamento ou evolução de sistemas informáticos na Administração Pública», «em todos os documentos de texto em formato digital que sejam objeto de emissão, intercâmbio, arquivo e ou publicação pela Administração Pública», nos prazos estabelecidos nas tabelas I a IX.

9 — As comunicações e os pareceres referentes às condições de exceção previstos no artigo 6.º da Lei n.º 36/2011, de 21 de junho, bem como o Relatório Anual da Interoperabilidade Digital são publicados em *site web* da Agência para a Modernização Administrativa, I. P., no endereço www.ama.pt.

TABELA I

Formatos de dados, incluindo códigos de carateres, formatos de som e imagens (fixas e animadas), audiovisuais, dados gráficos e de pré-impressão

Domínios de formato de dados	Acrónimo especificação técnica	Especificação técnica	Classificação	Prazo para aplicação	Referência
Interação com sistemas de gestão de bases de dados.	SQL	*Structured Query Language*.	Obrigatório	Entrada em vigor do Regulamento.	http://www.w3schools.com/sql/default.asp
Imagem *Raster*	PNG	*Portable Network Graphics*.	Recomendado		http://www.w3.org/TR/PNG
Imagem Vetorial.	SVG	*Scalable Vector Graphics*.	Obrigatório	Entrada em vigor do Regulamento.	http://www.w3.org/TR/SVG
Linguagem para descrição de documentos e formatação de dados, para interpretação não-humana.	XML	*Extensible Markup Language*.	Obrigatório	Entrada em vigor do Regulamento.	http://www.w3.org/TR/REC-xml/
Transformação de dados para conversão de dados em XML para outro formato.	XSLT 2.0	*XSL Transformations*	Obrigatório	Entrada em vigor do Regulamento.	http://www.w3.org/TR/xslt20/
Definição de estrutura de informação.	XSD	*XML Schema Definition*	Obrigatório	Entrada em vigor do Regulamento.	http://www.w3.org/TR/xmlschema-0/ http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/structures.html http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html
Transformação de dados para apresentação.	XSL 1.1	*Extensible Stylesheet Language*.	Obrigatório	Entrada em vigor do Regulamento.	http://www.w3.org/Style/XSL/
Protocolo baseado em XML para sistemas de mensagens instantâneas.	XMPP	*Extensible Messaging and Presence Protocol*.	Obrigatório	Entrada em vigor do Regulamento.	http://xmpp.org/rfcs/rfc6120.html
Lista de carateres válidos.	UTF-8	*8-bit Unicode Transformation Format*.	Obrigatório	Entrada em vigor do Regulamento.	http://tools.ietf.org/html/rfc3629

TABELA II

Formatos de documentos (estruturados e não estruturados) e gestão de conteúdos, incluindo gestão documental

Domínios de formato de documentos e gestão de conteúdos	Acrónimo especificação técnica	Especificação técnica	Classificação	Prazo para aplicação	Referência
Documentos editáveis para apresentação, gráficos, folhas de cálculo e processamento de texto.	ODF 1.1	*Open Document Format* v1.1 *(Second Edition) specification*.	Obrigatório	*i*) Documentos disponibilizados de e para o cidadão: Entrada em vigor do Regulamento. *ii*) Restantes documentos: 1 de julho de 2014.	http://docs.oasis-open.org/office/v1.1/OS/OpenDocument-v1.1.pdf
Formato para documentos que precisam de ser partilhados, geridos e preservados de forma segura e fiável.	PDF 1.7	*Portable Document Format*.	Obrigatório	Entrada em vigor do Regulamento.	http://wwwimages.adobe.com/www.adobe.com/content/dam/Adobe/en/devnet/pdf/pdfs/PDF32000_2008.pdf
Linguagem para descrição de documentos e formatação de dados, para interpretação não-humana.	XML 1.0	*Extensible Markup Language*.	Obrigatório	Entrada em vigor do Regulamento.	http://www.w3.org/TR/REC-xml/
Linguagem para descrição de documentos para apresentação nativa em *browsers.*	HTML 4.01	*Hypertext Markup Language*.	Obrigatório	Entrada em vigor do Regulamento.	http://www.w3.org/TR/html401/

TABELA III

Tecnologias de interface *web,* incluindo acessibilidade, ergonomia, compatibilidade e integração de serviços

Domínios de tecnologias de interface *web*	Acrónimo especificação técnica	Especificação técnica	Classificação	Prazo para aplicação	Referência
Sindicação de conteúdos *web* . . .	ATOM 1.0	*Atom Syndication Format* 1.0.	Obrigatório	Entrada em vigor do Regulamento.	http://tools.ietf.org/html/rfc4287
Acesso remoto a calendários . . .	CalDav	*Calendaring Extensions to web DAV* (CalDAV).	Obrigatório	1 de julho de 2014 . . .	http://tools.ietf.org/html/rfc4791
Linguagem para descrição da semântica de apresentação de página *web*.	CSS2.1	*Cascading Style Sheets* 2.1.	Obrigatório	Entrada em vigor do Regulamento.	http://www.w3.org/TR/REC-CSS2
Linguagem para descrição de documentos para apresentação nativa em *browsers*.	HTML 4.01	*Hypertext Markup Language*.	Obrigatório	Entrada em vigor do Regulamento.	http://www.w3.org/TR/html401/
Protocolo de hipertexto para disponibilização de página *web*.	HTTP/1.1	*Hypertext Transfer Protocol*.	Obrigatório	Entrada em vigor do Regulamento.	http://tools.ietf.org/html/rfc2616
Protocolo hipertexto seguro para disponibilização de página *web*, utilizando o protocolo HTTP/1.1 com TLS 1.0 (adotado como Especificação técnica aberta no presente Regulamento).	HTTPS	*Hypertext Transfer Protocol Secure*.	Obrigatório	Entrada em vigor do Regulamento.	http://tools.ietf.org/html/rfc2818
Linguagem de *scripting* para página *web*.	Javascript 1.5	*Javascript* 1.5	Recomendado		http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%203rd%20edition,%20December%201999.pdf
Nível de acessibilidade para *sites* Internet que disponibilizem exclusivamente informação e conteúdos, de acordo com a Resolução do Conselho de Ministros n.º 155/2007.	WCAG 2.0 — nível «A»	*Web Content Accessibility Guidelines* 2.0 — nível «A».	Obrigatório	Entrada em vigor do Regulamento.	http://www.w3.org/TR/WCAG20
Nível de acessibilidade para *sites* Internet que disponibilizem serviços *online,* de acordo com a Resolução do Conselho de Ministros n.º 155/2007.	WCAG 2.0 — nível «AA»	*Web Content Accessibility Guidelines* 2.0 — nível «AA»	Obrigatório	Entrada em vigor do Regulamento.	http://www.w3.org/TR/WCAG20
Nível de acessibilidade para *sites* Internet que disponibilizem exclusivamente informação e conteúdos, de acordo com a Resolução do Conselho de Ministros n.º 155/2007.	WCAG 2.0 — nível «AA» ou «AAA»	*Web Content Accessibility Guidelines* 2.0 — nível «AA» ou «AAA».	Recomendado		http://www.w3.org/TR/WCAG20
Nível de acessibilidade para *sites* Internet que disponibilizem serviços *online,* de acordo com a Resolução do Conselho de Ministros n.º 155/2007.	WCAG 2.0 — nível «AAA»	*Web Content Accessibility Guidelines* 2.0 — nível «AAA».	Recomendado		http://www.w3.org/TR/WCAG20
Acesso remoto a sistemas de ficheiros.	WebDAV	*Web Distributed Authoring and Versioning Access Control Protocol*.	Recomendado		http://tools.ietf.org/html/rfc3744
Linguagem para descrição de documentos e formatação de dados, para interpretação não-humana.	XML 1.0	*Extensible Markup Language*.	Obrigatório	Entrada em vigor do Regulamento.	http://www.w3.org/TR/REC-xml/
Linguagem de definição de estilos XML.	XSL v1.1	*XML stylesheet language XSL* v1.1.	Obrigatório	Entrada em vigor do Regulamento.	http://www.w3.org/TR/2006/REC-xsl11-20061205

TABELA IV

Protocolos de *streaming* ou transmissão de som e imagens animadas em tempo real, incluindo o transporte e distribuição de conteúdos e os serviços ponto a ponto

Domínio de protocolo de *streaming*	Acrónimo especificação técnica	Especificação técnica	Classificação	Prazo para aplicação	Referência
Streaming de áudio e vídeo . . .	RTSP	*Real Time Streaming Protocol*.	Obrigatório	Entrada em vigor do Regulamento.	http://www.ietf.org/rfc/rfc2326.txt

TABELA V

Protocolos de correio eletrónico, incluindo acesso a conteúdos e extensões e serviços de mensagem instantânea

Domínios de protocolos de correio eletrónico	Acrónimo especificação técnica	Especificação técnica	Classificação	Prazo para aplicação	Referência
Consulta de *e-mail*	IMAP 4	*Internet Message Access Protocol.*	Obrigatório	Entrada em vigor do Regulamento.	http://tools.ietf.org/html/rfc3501
Formato de mensagens de correio eletrónico.	MIME	RFC 2045, 2046, 2047 — *Multipurpose Internet Mail Extensions*.	Obrigatório	Entrada em vigor do Regulamento.	http://tools.ietf.org/html/
Acesso remoto a uma caixa de correio eletrónico.	POP3	RFC 1939 — *Post Office Protocol*.	Obrigatório	Entrada em vigor do Regulamento.	http://www.ietf.org/rfc/rfc1939.txt
Acesso seguro remoto a uma caixa de correio eletrónico.	POP3S, IMAPS	RFC 2595 *Using TLS with IMAP, POP3 and ACAP*.	Recomendado		http://tools.ietf.org/html/rfc2595
Envio de correio eletrónico . . .	SMTP	*Simple Mail Transfer Protocol* — RFC 5321.	Obrigatório	Entrada em vigor do Regulamento.	http://www.ietf.org/rfc/rfc2821.txt
Envio seguro de correio eletrónico.	SMTPS	RFC 3207 SMTP *Service Extension for Secure SMTP over Transport Layer Security* http://www.ietf.org/rfc/rfc3207.txt	Recomendado		http://www.ietf.org/rfc/rfc3207.txt

TABELA VI

Sistemas de informação geográfica, incluindo cartografia, cadastro digital, topografia e modelação

Domínio de sistemas de informação geográfica	Acrónimo especificação técnica	Especificação técnica	Classificação	Prazo para aplicação	Referência
Web Coverage Service	WCS	*Web Coverage Service*	Obrigatório	Entrada em vigor do Regulamento.	http://www.opengeospatial.org/standards/wcs
Web Feature Service	WFS	*Web Feature Service*	Obrigatório	Entrada em vigor do Regulamento.	http://www.opengeospatial.org/standards/wfs
Web Map Service	WMS	*Web Map Service*	Obrigatório	Entrada em vigor do Regulamento.	http://www.opengeospatial.org/standards/wms
Web Processing Service	WPS	*Web Processing Service*	Obrigatório	Entrada em vigor do Regulamento.	http://www.opengeospatial.org/standards/wps

TABELA VII

Especificações técnicas e protocolos de comunicação em redes informáticas

Domínios de comunicação em redes informáticas	Acrónimo especificação técnica	Especificação técnica	Classificação	Prazo para aplicação	Referência
Envio de pacotes de dados numa rede informática.	IPv6	*Internet Protocol, Version* 6 (IPv6).	Recomendado		http://tools.ietf.org/html/rfc2460

TABELA VIII

Especificações técnicas de segurança para redes, serviços, aplicações e documentos

Domínios de segurança para redes, serviços, aplicações e documentos	Acrónimo especificação técnica	Especificação técnica	Classificação	Prazo para aplicação	Referência
	TLS 1.0	*Transport Layer Security*.	Obrigatório	1 de janeiro de 2014 . . .	http://tools.ietf.org/html/rfc2246

TABELA IX

Especificações técnicas e protocolos de integração, troca de dados e orquestração de processos de negócio na integração interorganismos

Domínios de integração, troca de dados, integração de serviços e orquestração	Acrónimo especificação técnica	Especificação técnica	Classificação	Prazo para aplicação	Referência
Representação gráfica para a especificação de processos de negócio.	BPMN 2.0	*Business Process Model and Notation*.	Recomendado		http://www.omg.org/spec/BPMN/2.0
Canal de transporte para integração entre 2 ou mais sistemas de informação não requerendo segurança do canal.	HTTP/1.1	*Hypertext Transfer Protocol*.	Obrigatório	Entrada em vigor do Regulamento.	http://tools.ietf.org/html/rfc2616
Canal de transporte para integração entre 2 ou mais sistemas de informação requerendo segurança do canal.	HTTPS	*Hypertext Transfer Protocol Secure*.	Obrigatório	Entrada em vigor do Regulamento.	http://tools.ietf.org/html/rfc2818
Acesso a diretórios de informação.	LDAP	*Lightweight Directory Access Protocol*.	Obrigatório	Entrada em vigor do Regulamento.	http://www.ietf.org/rfc/rfc1777.txt
Autenticações, autorizações e troca de atributos entre 2 ou mais sistemas de informação interorganismos da Administração Pública.	SAML 2.0	*Security Assertion Markup Language* 2.0.	Obrigatório	Entrada em vigor do Regulamento.	http://docs.oasis-open.org/security/saml/v2.0/
Estrutura das mensagens trocadas para Integração entre 2 ou mais sistemas de informação.	SOAP 1.1	*Simple Object Access Protocol* 1.1.	Obrigatório	Entrada em vigor do Regulamento.	http://www.w3.org/TR/2000/NOTE-SOAP-20000508/
Comunicação da informação de endereços entre *web services* entre 2 ou mais sistemas de informação.	WS-Addressing 1.0	*Web Services Addressing*.	Obrigatório	Entrada em vigor do Regulamento.	http://www.w3.org/TR/ws-addr-core/
Protocolo para a garantia de entrega de mensagens na integração entre 2 ou mais sistemas de informação interorganismos da Administração Pública.	WS-RM 1.1	*WS-Reliable Messaging* 1.1.	Recomendado		http://docs.oasis-open.org/ws-rx/wsrm/200702/wsrm-1.1-spec-os-01.pdf
Segurança de integridade e confidencialidade da comunicação na Integração entre 2 ou mais sistemas de informação inter-organismos da Administração Pública.	WS-Security 1.2	*Web Services Security* 1.2.	Recomendado		http://docs.oasis-open.org/ws-sx/ws-securitypolicy/v1.2/ws-securitypolicy.html
Segurança de autenticação da comunicação na integração entre 2 ou mais sistemas de informação interorganismos da Administração Pública.	WS-Security Username Token Profile 1.0	*WS-Security Username Token Profile* 1.0.	Recomendado		http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0.pdf

REGIÃO AUTÓNOMA DA MADEIRA

Assembleia Legislativa

Decreto Legislativo Regional n.º 29/2012/M

Adapta ao Sistema Regional de Saúde da Região Autónoma da Madeira a Lei n.º 11/2012, de 8 de março, que estabelece as novas regras de prescrição e dispensa de medicamentos, procedendo à sexta alteração ao regime jurídico dos medicamentos de uso humano, aprovado pelo Decreto-Lei n.º 176/2006, de 30 de agosto, e à segunda alteração à Lei n.º 14/2000, de 8 de agosto.

A Lei n.º 11/2012, de 8 de março, aprovou as novas regras de prescrição e dispensa de medicamentos, destacando-se a obrigatoriedade de a prescrição se efetuar por denominação comum internacional (DCI) da substância ativa, forma farmacêutica, dosagem, apresentação e posologia como regra.

A política do medicamento na Região Autónoma da Madeira tem assumido ao longo do tempo peculiar singularidade com especiais reflexos, denominadamente de cariz social, económico e financeiro, face à existência do Sistema Regional de Saúde, cuja regulação e financiamento é exercida pela Região, na defesa e promoção da saúde.

Por seu turno, o Programa de Ajustamento Económico e Financeiro da Região Autónoma da Madeira determina a adoção na Região de todas as medidas preconizadas a nível nacional no tocante à política do medicamento.

Neste sentido, importa adaptar o predito diploma às especificidades da Região Autónoma da Madeira.

Por fim, não obstante o princípio da prescrição por DCI estar cominado no Decreto Legislativo Regional n.º 16/2010/M, de 13 de agosto, alterado pelo Decreto Legislativo Regional n.º 2/2012/M, de 16 de março, o normativo estabelecido na Lei n.º 11/2012, de 8 de março, que ora se adapta, difere do normativo vertido no sobredito diploma regional, pelo que se procedeu à sua revogação.

Assim:

A Assembleia Legislativa da Região Autónoma da Madeira decreta, nos termos do disposto na alínea *a*) do n.º 1