European Master in Software Engineering
Technische Universität Kaiserslautern
Department of Computer Science

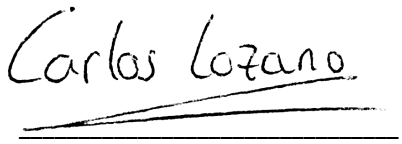
MASTER THESIS

Platform for designing, developing and deploying microservices

Presented	December 15th, 2015
Author	Carlos Lozano Sánchez
Supervisors	Prof. Dr. Paul Müller
	Prof. Dr. Ricardo Imbert Paredes
	Dr. Markus Hillenbrand

Schriftliche Erklärung

Ich versichere hiermit, dass ich die vorliegende Masterarbeit mit dem Thema „Platform for designing, developing and deploying microservices“ selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, habe ich durch die Angabe der Quelle, auch der benutzten Sekundärliteratur, als Entlehnung kenntlich gemacht.



Kaiserslautern, 15.12.2015

A mis padres por darme todo su apoyo
durante esta experiencia inolvidable

A mis sobrinos Naya, Hugo y Asier por
hacerme sentir como un niño

A mi hermana por su confianza
y apoyo incondicional

A mis amigos por estar a mi lado
incluso en la lejanía

A todos vosotros os dedico esta tesis
con mucho cariño y agradecimiento

Acknowledgments

I would like to express my sincere gratitude to those persons, who directly or indirectly, have contributed to the realization of this thesis.

Firstly, I would like to thank my supervisors, Prof. Dr. Paul Müller from Technische Universität Kaiserslautern and Prof. Dr. Ricardo Imbert Paredes from Universidad Politécnica de Madrid, for their confidence in me and in this project. For me, it has been a real honor and pleasure to work with you.

Secondly, I would like to thank my advisor, Dr. Markus Hillenbrand from Technische Universität Kaiserslautern, for his patience, guidance, knowledge, and insightful comments. Now I can asseverate that if I had to start again, I would like to do it with you.

Finally, I would like to thank the members of the Integrated Communication Systems working group for their interest in this project and their constructive comments.

Thank you.

Abstract

Internet has transformed the current society, enabling a further globalization and interconnection between people, organizations and countries. This revolution has forced software engineers to seek new methods and technologies in order to develop more powerful and complex software systems that can be massively used by millions of people around the world every day. During recent decades, many developers have trusted service-oriented computing to develop this type of systems. This thesis continues the service-oriented trend by proposing an innovative platform, named Microrestjs, for designing, developing and deploying microservices-oriented systems. More specifically, Microrestjs provides a methodology, a framework and some core services that work together with the purpose of simplifying the development of powerful software systems. Definitely, this thesis aims to transform the way developers will create the future software.

Zusammenfassung

Das Internet hat die gegenwärtige Gesellschaft verwandelt und eine weitreichende Globalisierung und Verbindung zwischen Menschen, Organisationen und Ländern ermöglicht. Diese Revolution hat Software-Ingenieure dazu gezwungen, neue Methoden und Technologien zu suchen, um leistungsfähigere und komplexere Softwaresysteme zu entwickeln, die von Millionen Menschen rund um die Welt und jeden Tag genutzt werden. In den letzten Jahrzehnten haben viele Entwickler ihr Vertrauen in die Serviceorientierung gesetzt, um diese Art von Systemen zu entwickeln. Die vorliegende Masterarbeit führt diesen serviceorientierten Trend fort und schlägt eine innovative Plattform - Microrestjs genannt - zum Entwerfen, Entwickeln und Bereitstellen mikroserviceorientierter Systeme vor. Genauer gesagt bietet Microrestjs eine Methodik, ein Rahmenwerk und einige Basisdienste, die zusammenarbeiten mit dem Ziel, die Entwicklung leistungsfähiger Softwaresystemen zu vereinfachen. Zweifellos ziel diese Masterarbeit darauf ab, den Weg der Entwicklung zukünftiger Software zu wandeln.

Resumen

Internet ha transformado la sociedad actual en la que vivimos, permitiendo una mayor globalización e interconexión entre personas, organizaciones y países. Esta revolución ha obligado a los ingenieros de software a buscar nuevos métodos y tecnologías para conseguir desarrollar sistemas más potentes y complejos que puedan ser utilizados masivamente por millones de personas, alrededor del mundo, cada día. Durante las últimas décadas, muchos desarrolladores han confiado en la computación orientada a servicios para desarrollar dichos sistemas. Esta tesis continúa dicha tendencia proponiendo una innovadora plataforma, conocida como Microrestjs, para diseñar, desarrollar y desplegar sistemas construidos mediante microservicios. Concretamente, Microrestjs proporciona una metodología, un framework y algunos servicios básicos que trabajan conjuntamente con el objetivo de simplificar el desarrollo de sistemas software potentes. En definitiva, esta tesis pretende transformar la manera en que los desarrolladores crearán el software del futuro.

Table of contents

List of figures

1. Introduction

Science and technology are evolving at a rate never seen before in human history. Every decade, our knowledge in all the fields grows more than in the previous one, i.e. our rate of progress increases exponentially. This trend known as The Law of Accelerating Returns [1,2] was proposed and described in detail by Raymond Kurzweil in 1999.

This exponential growth would not be possible without the development of new suitable tools, methodologies, and approaches. In the last decades, the Information and Communications Technologies have played an essential role in the development of powerful and automatic tools. On the other hand, the Software Engineering discipline has also evolved with the purpose of developing higher quality tools as well as reducing the overall cost, the time-to-market, and the risks.

This thesis presents a platform for designing, developing and deploying new tools and applications. For that, it will introduce an approach to help developers build their own tools using microservices; and a framework to support the development and deployment of those microservices. In one word, this thesis aims to help develop the applications and tools of the future with the proposed platform.

This thesis is organized in five chapters that justify, design and evaluate the proposed microservices-oriented platform.

Chapter 2 reviews in detail important concepts to understand how the industry materialized microservices, the reasons for using them and the importance of applying microservices-oriented approaches in software development. First, service-oriented architectures, traditional Web services and RESTful Web services are presented in order to obtain a general knowledge about services. Then, microservices architectures will be introduced and connected thoroughly with other kinds of services.

Chapter 3 describes some of the Web development problems that developers face everyday and the causes behind them with the purpose of understanding which features should be included in new solutions to improve the development of Web applications. Then, following subsections introduce in detail “Microrestjs”, the platform for designing, developing and deploying microservices that has been developed in this thesis in order to solve the problems described previously.

Chapter 4 evaluates the whole platform with the purpose of obtaining a complete picture of the platform and verifying if all the objectives have been achieved successfully.

Finally, chapter 5 draws the obtained conclusions and analyzes which parts require more work and research to improve the proposed platform.

2. Background

In last decades, Software Engineering has matured and experienced more significant changes than other fields and sciences. The main researches in Software Engineering were focused on how to increase the quality of the software, how to reduce software complexity, costs, time-to-market and, ultimately, how to minimize the risk of project failure. All these aspects cannot be improved without making new processes, methodologies, paradigms, architectures, and technologies that promote reusability as a key factor. In 1960s, reusability was improved using subroutines. In 1970s, using modules. In 1980s, using objects. In 1990s, using components. In 2000s, using services. And nowadays, microservices are used for increasing the reusability among different projects. The purpose of the following sections is to obtain a complete picture of the last two breakthroughs and how services have evolved into microservices.

2.1. Service-oriented architecture

Service-oriented architecture (SOA) is introduced by Gartner in 1996 as "a client/server design approach in which an application consists of software services and software service consumers (also known as clients or service requesters). SOA differs from the more general client/server model in its definitive emphasis on loose coupling between software components, and in its use of separately standing interfaces". [3]

Although this first definition is quite wide and ambiguous, it outlines some of the key concepts of SOA: design approach, services, consumers, loose coupling, and interfaces. Other key concepts of SOA are presented in some other definitions:

> "Service-oriented architecture is a business-driven IT architecture approach that supports integrating your business as linked, repeatable business tasks, or services. SOA helps today's business innovate by ensuring that IT systems can adapt quickly, easily, and economically to support rapidly changing business needs. SOA helps customers increase the flexibility of their business processes, strengthen their underlying IT infrastructure and reuse their existing IT investments by creating connections among disparate applications and information sources". [4]

> "Service-oriented architecture (SOA) is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations". [5]

From the IBM and OASIS definitions, it can be deduced that SOA is closely related to business, flexibility, reusability, distributed capabilities, ownership, uniformity, discoverability, and interoperability, among others.

As can be observed, SOA is a complex paradigm with a large number of properties and concepts that require to be explained step by step.

The basic aim of SOA is to provide capabilities via services to consumers with the purpose of solving their business needs. For that, three important aspects may be considered:

1. **Visibility.** Capabilities providers and consumers require some methods to know and understand each other. These methods are descriptions and interfaces that specify functional and technical requirements, constraints, policies, and communication mechanisms.
2. **Interaction.** Consumers require using capabilities providers with the purpose of solving their business needs. This interaction can be achieved through the exchange of messages.
3. **Real world effects.** The purpose of the interaction is to accomplish effects that solve the business needs of the consumers. These effects may be the return of some information, a change of state or the combination of the two previous possibilities.

From the above aspects, the concept of a service can be derived. According to [5], "a service is a mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description".

At this point, it is important to be able to distinguish between capabilities, services, service providers and service consumers. In general, a service can be seen as a group of operations that provides one or more capabilities through an interface. On the one hand, a service provider offers a service, which, in turn, provides some capabilities. And, on the other hand, a service consumer uses the service offered by the service provider in order to solve some business need.

One more component, the service directory or service registry, may be described to complete the general picture. The service directory is in charge of increasing the visibility of the services. It acts as an intermediary between providers and consumers. Providers can publish their services and location, and consumers can search these services and know their locations.

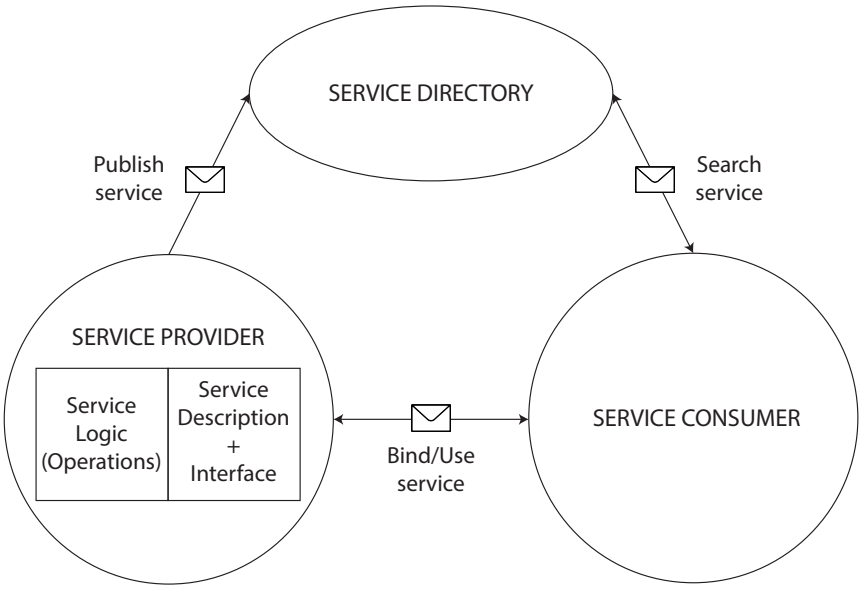


Figure 1. SOA Triangle

Apart from the main components and concepts, Thomas Erl also proposes eight principles to design high-quality services [6]:

1. **Reusability.** Every service should be able to be used again in a different context requiring minimum adaptations. Therefore, each service should have a clear objective and avoid implementing really specific operations to achieve high reusability.
2. **Formal contract.** Every service should be described formally through a service description that specifies the messages, operations, location, policies, and constraints, among others. Service consumers need all this information to interact properly with the service.
3. **Loose coupling.** Every service should avoid depending on other services as much as possible because a change in these other services can require also modifying the service itself.
4. **Abstraction.** Every service should provide a well-designed interface and service description with the purpose of hiding the concrete service implementation to the outside world, i.e. each service should be a black box. This property enhances reusability and loose coupling.
5. **Composability.** A service can act as a façade representing other services. In this way, the façade service abstracts and reuses previous services to offer new functionalities to service consumers. But, composability should be used carefully because it may increase coupling between the façade and the previous services, and reduce it between the previous services and service consumers.
6. **Autonomy.** Each service should have complete control over its encapsulated logic to reduce coupling and enable an evolution independently to other services.
7. **Statelessness.** Services should be designed to execute their operations without managing any specific state of the service consumers in order to improve reusability and scalability. However, this approach requires implementing more sophisticated messages that contain all the necessary information to execute each operation fully independently at any time.
8. **Discoverability.** Services should be able to be discovered and used by service consumers. For that, every service may use service directories to publish its metadata, which is usually defined in the service description, with the purpose of being found by service consumers.

To sum up, SOA is an architectural style focused on developing independent pieces of software, services, which can be executed by service consumers. As any other architectural style, SOA does not specify the technologies to implement the functionality; it only defines and specifies the necessary concepts and components to implement satisfactorily a service-oriented system and provide a common vocabulary and principles to developers in order to speak the same language.

2.2. Web services

W3C proposes, in [7], a standardized architecture that respects all the SOA aspects. This standardized architecture identifies the functional components, describes their relationships, and specifies the technology to build Web services.

W3C defines the term Web service as “a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.”

As can be observed in the given definition, the main goal of Web services is to enable the interoperability between different pieces of software through the usage of several technologies such as WSDL, SOAP and XML.

W3C suggests the development of “agents” as the first step to achieve the desired interoperability. An agent is basically a concrete implementation in a specific programming language either of a Web service or of a service consumer. Thereby, Web services are considered abstract entities (i.e. programming language- and implementation-independent) that do not require to be changed if the service functionality remains the same.

In addition, W3C uses the terms provider entity and requester entity to refer to the organizations and persons that provide or request a Web service, respectively. Thus, provider entities offer services using provider agents. And those services are consumed using requester agents by requester entities.

With the purpose of being able to interact provider entities and requester entities through their respective agents, a formal contract of the Web service is required. Otherwise, provider and requester agents would not be able to understand each other. This formal contract, also known as Web Service Description (WSD), is a machine-understandable specification written in WSDL, a language for describing syntactically Web services based on XML. Using WSDL, it can be defined the service operations, the structure of the messages, the transport protocols, and the physical location of the services, among others.

Apart from the syntactic Web service description, a semantic description of the service might be necessary to detail the purpose of the service, the expected behavior, the non-functional properties, or other additional information as for example legal information, quality of service or service pricing. Besides, these semantic descriptions can be used to improve the discoverability of the Web services.

Usually, a functional description of the service is also defined from the semantic description. Basically, this functional description is a machine-understandable description of the service functionality in order to be able to publish some meta-information into a service directory to improve the discoverability of the service. Normally, the service directory is implemented as a Universal Description, Discovery and Integration (UDDI) registry – a standardized registry for Web services sponsored by OASIS.

All the above concepts and the common steps to discover and interact with services are illustrated in Figure 2. First, the provider entity usually defines the Web service description and Web service semantics; and registers the Web service description and the associated functional description into a service directory using the discovery service (step 1.a). Once the service has been registered, requester entities can look it up (step 1.b) and retrieve its Web service description (step 1.c). Then, both entities may need to agree somehow on the semantics that will manage the interaction (step 2). After the agreement between the concerned parties, the agents are implemented taking into consideration the Web service description and semantics (step 3). Finally, the requester agent can interact with the provider agent and use the service (step 4).

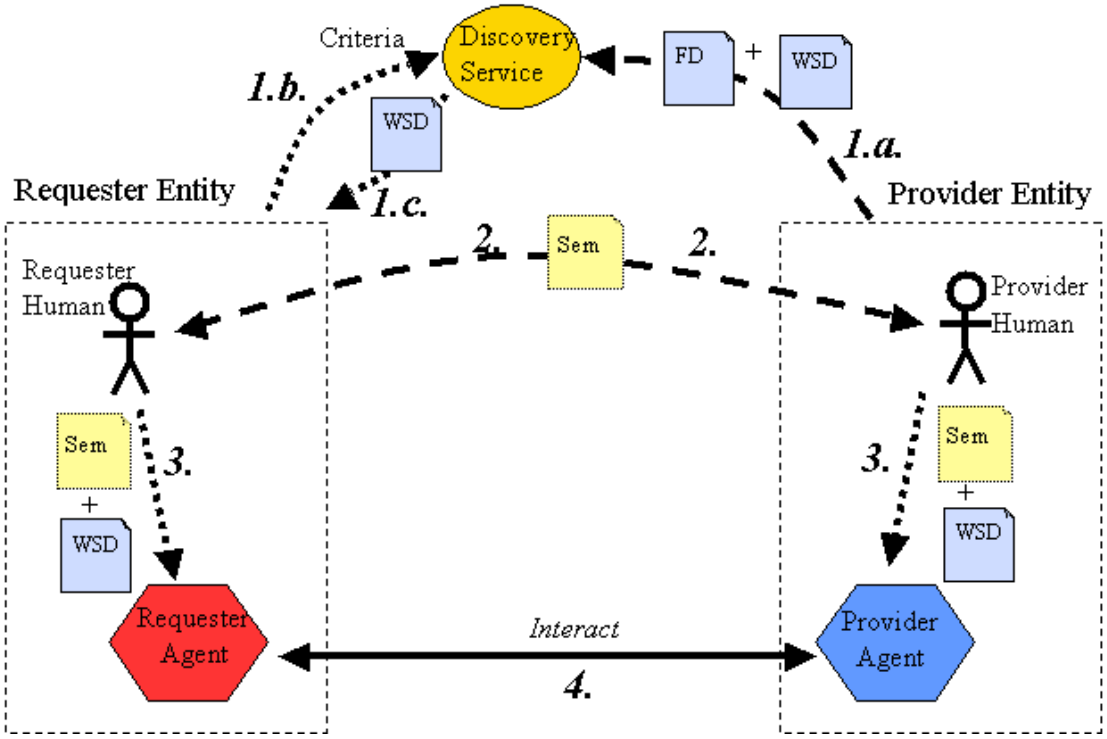


Figure 2. The general process of using a Web service (Source: [7])

The above picture illustrates all the aspects involved into Web services, but it does not present how the agents should be design or how the interaction between agents is achieved. Most of these design aspects are shown in Figure 3, which introduces the architecture stack proposed by W3C for developing Web services.

The proposed architecture is divided into four layers. From bottom to up, the first layer makes reference to well-known network protocols that agents might use to communicate each other over the Internet. These network protocol are used to exchange the XML messages defined by the second layer. These messages are created using the messaging framework SOAP, which defines a set of rules to serialize and encode messages in a standardized manner. The third layer denotes the importance of using service descriptions. In our case, service descriptions act as a bridge between the service logic and the messages, and also specify other aspects such as the service location or the network protocols. Using the service descriptions, it is possible to generate automatically the code of stubs and

skeletons with all the functionality of the behind layers. Thereby, agent developers may only need to implement the service logic and use the auto-generated code to interact with other Web services. Finally, last layer offers some general processes, like discovery or choreography, that could be used by the services to enable discoverability, composability, and other properties.

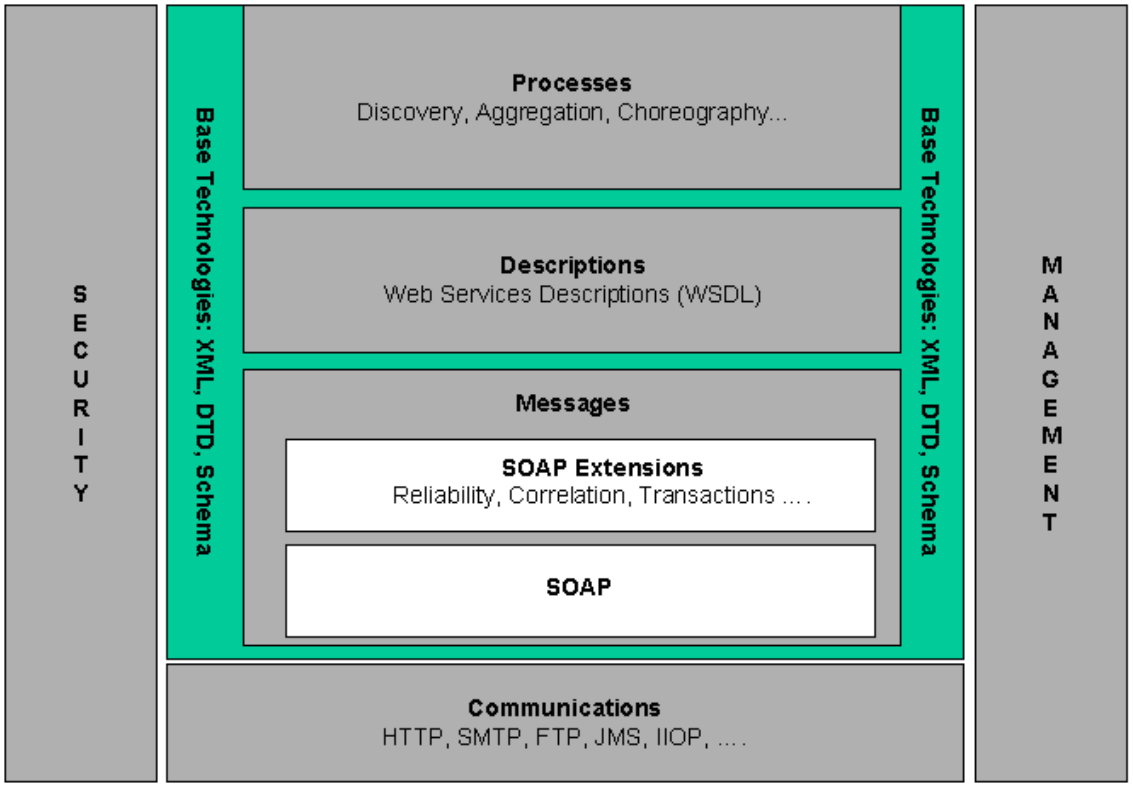


Figure 3. Web services architecture stack (Source: [7])

Additionally to the four layers, W3C also defines two transversal layers –security and management– in order to be able to implement different mechanisms at different levels. For instance, it would be possible to use SSL in the communications level and authentication in the processes level.

Recapitulating, W3C proposes a complete architecture and several technologies to develop standardized and platform-independent services that can interact between them without problems. Thus, the main aim of W3C is certainly to offer a possible implementation technology for SOA. Nevertheless, the usage of Web services does not guarantee that the implemented services would be SOA-compliant because of the freedom provided by the proposed architecture and technologies.

2.3. RESTful Web services

Roy Fielding and Richard Taylor present the Representational State Transfer (REST) [8] architectural style with the purpose of determining the principles, constraints and elements for the development of modern Web-based applications. REST appears in response to the need for a well-defined model on how the Web should work to achieve their initial goals.

According to Berners-Lee, the major goal of the Web [9] was to "be a shared information space through which people and machines could communicate". The problem was how to enable the access to that information space taking into consideration, among others, that the information could have different structure and format, the machines may be heterogeneous, and the operating systems might be different. Therefore, the real challenge was to create a system that could provide information through a universally consistent interface in order to minimize the dependence of concrete technologies and systems in such a way that the Web could scale over the time without experiencing a negative impact in previous functionality. In short, the non-functional properties of the Web should be low latency, high scalability, high adaptability, and high extensibility.

This vision of the Web and the desired non-functional properties are accurately modeled and contemplated by REST architectural style. REST distinguishes three classes of architectural elements: data elements, connectors, and components.

The data elements define how the data used by the application is represented, processed and distributed. In particular, REST defines six data elements to obtain a clear separation of data types:

1. **Resources.** A resource represents any conceptual information that can be named, accessed and manipulated.
2. **Resource identifiers** A resource identifier determinates unequivocally a specific resource in order to be able to access it and manipulate it.
3. **Representations.** A representation reflects the current or intended state of the resource identified by the resource identifier. In other words, a representation is the actual sequence of bytes that is transferring between components.
4. **Representation metadata.** The representation metadata describes important aspects of the representation such as the media type or the last-modified time. This metadata might be necessary to process correctly the representation.
5. **Resource metadata.** A resource can have additional meta-information about itself such as alternative representations or the original source of the information.
6. **Control data.** The control data describes the purpose of the current message. For instance, the request action or the response meaning.

The connectors provide a uniform interface to enable the communication between components and the encapsulation of requests and responses. Basically, this interface specifies the in-parameters (request control data, resource identifier, and an optional representation) and the out-parameters (response control data, optional resource metadata, and an optional representation) for the

communication. At this point, it is important to note that REST uses stateless interactions and this implies that every request has to contain all the necessary information to be completely understood and processed, independently to previous requests, by connectors and components. In particular, REST defines five connector types to manage the communication between components:

1. **Client.** The client connector initiates the communication with the server connector sending a request with the proper in-parameters. And, it waits until the server connector responds with the out-parameters.
2. **Server.** The server connector listens for clients' requests to execute the functionality of the components using the received in-parameters. After each execution, the server responds to the client connector with the out-parameters of the execution.
3. **Cache.** A cache connector can be located on a client or server connector to store temporarily responses with the purpose of reusing them again in future requests. Thereby, if a cache connector is located on a client connector, the client connector may avoid the repetition of further network communications. And, if a cache connector is located on a server connector, the server connector might avoid the repetition of computing new responses for the same requests. Therefore, the cache connector can be used mainly to reduce the interaction latency and improve performance.
4. **Resolver.** A resolver connector determines the network address information of a resource identifier in order to be able to establish a communication with the component that has such resource.
5. **Tunnel.** A tunnel connector allows establishing a direct communication between clients and servers using some network protocol. The typical scenarios are when a secure communication is required or when the client cannot reach the server directly because it is part of a private network.

The components are the elements in charge of processing the information and executing the functionality. REST defines only four components to handle and process the information:

1. **User agent.** A user agent uses a client connector to interact with the origin server, through requests and responses, with the purpose of executing some operations on the origin server.
2. **Origin server.** An origin server implements operations to access, use and modify its resources. In order to execute these operations, an origin server uses a server connector to receive requests from user agents and responds to them with the result of the execution.
3. **Proxy.** A proxy is an intermediary component selected by the client to perform some modifications or validations to clients' requests and/or servers' responses.
4. **Gateway.** A gateway, also known as reverse proxy, is an intermediary imposed by the network or origin server to perform some modifications or validations to clients' requests and/or servers' responses.

All the above REST architectural elements are the necessary concepts to know how to design RESTful Web services. In the paper [8], Roy Fielding and Richard Taylor also define how the above elements can work together. These aspects are not

discussed in this document, but the reader can obtain a precise picture of the REST architectural views in such paper.

Once the main concepts of REST have been presented, RESTful Web services can be introduced in detail. A RESTful Web service is a Web-accessible service that respects the rules defined by REST architectural style. Although these RESTful Web services can be implemented using the same technologies as W3C Web services, many RESTful Web services usually use other different technologies in order to obtain more lightweight services. Actually, the current trend is to develop the so-called RESTful HTTP Web services, hereinafter referred to as RESTful Web services.

Nowadays, RESTful Web services are basically characterized by the usage of HTTP protocol to perform the communications, URIs to name and identify the resources, and lightweight formats to transfer the representations of the resources. These three important aspects for developing RESTful Web services are described below:

Hypertext Transfer Protocol (HTTP) [10] is a request/response protocol for client-server communications. Thus, HTTP basically allows clients to send requests to servers and servers to respond to clients' requests. With the purpose of performing these interactions, the HTTP protocol proposes the use of messages with the following format:

1. **Start line.** The first line of the message defines either the purpose of the communication or its result if the message is a request or a response, respectively. When the message is a request, this start line specifies an HTTP Method, the request URI, and the HTTP Version. On the other hand, when the message is a response, this start line specifies the HTTP Version, an HTTP Status Code, and a brief description associated to the HTTP Status Code.
2. **Message header.** The next part of the message is the message header, which is composed of key-value pairs that describe the message and the content of the message. In other words, the message header is metadata necessary for the communication between clients and servers. HTTP defines which metadata can be included in the message header and, in general, it may depend on whether the message is a request or a response. For instance, requests may include the media types that the client can understand and responses might include information about how the information should be cached.
3. **Message body.** The last part of the message is the message body, which contains the information itself. In the case of HTTP, all type of information (plain text, JSON data, XML data, HTML data, images, etc.) can be included because HTTP has only the responsibility to transmit information and not to understand that information. The responsibility to understand the information lies in the components that will interpret the information according to the Content-type defined as metadata in the message header.

Although all above aspects make HTTP protocol a good option to use it to develop REST-compliant Web services, this protocol does not respect completely the REST architectural style because control data and representation metadata are mixed in

the message header. Moreover, HTTP allows the usage of cookies to define state information, which violates the REST principle that states that all the interactions are stateless.

On the other hand, a Uniform Resource Identifier (URI) [11] is "a compact sequence of characters that identifies an abstract or physical resource" that was designed for naming and identifying global resources over the Web. These universal identifiers satisfy all the requirements of REST resource identifiers, thus RESTful Web services usually use URIs to identify the resources. In particular, the most of the times, RESTful Web services uses a subset of URIs known as Uniform Resource Locator (URL). A URL does not only allow identifying resources, but also locating those resources on the Web. For this reason, URL identifiers are considered a perfect mechanism to access and use resources.

Finally, the correct choice of the representation format might make the difference in the sense of latency and performance. Nowadays, RESTful Web services usually use JSON format to send structured representations because JSON is more lightweight than other formats like XML or HTML. For unstructured information, RESTful Web services may use plain text or JSON format. And for binary information, RESTful Web services might use a large number of media types [12] defined by IANA.

Despite the fact that most of the RESTful Web service concepts are now related to the REST architectural style concepts, one more detail will be explained in order to have a complete overview of how to design RESTful Web services.

REST architectural style specifies the use of a uniform interface for the connectors in order to access and use the resources easily. In general, this uniform interface is provided by the HTTP methods. HTTP has several methods; however, the most relevant ones are GET, POST, PUT, and DELETE that are used for retrieving, creating, updating, and deleting resources, respectively. The proper usage of these methods is really important in order to respect the REST architectural style. Besides, developers are who decide which method should be used in each case because HTTP by itself does not define any kind of restriction. In addition, developers should also consider the properties of each HTTP method to take the final decision of which one should be used because some aspects, like being able to enable the cache, depends on these properties. On the one hand, an HTTP method is considered safe if it does not modify the resource. On the other hand, an HTTP method is considered idempotent if the same result is obtained in repeatedly executions. Considering these properties, the method GET should be safe and idempotent; POST should be neither safe nor idempotent; PUT should not be safe but idempotent; and DELETE should not be safe but may be idempotent.

Apart from satisfying REST, RESTful Web services should also respect SOA in order to be considered as SOA-compliant services. One of the most important aspects of SOA is the formal contract between providers and consumers. This aspect can be achieved in RESTful Web services using Web Application Description Language (WADL) [13] – a machine-processable language for describing RESTful Web services. Other SOA aspects like composability or discoverability are not formally

covered by RESTful Web services and, therefore, additional technologies should be used to obtain a complete SOA-compliant service.

Summarizing, REST is an architectural style that defines the general concepts to develop Web-based applications with low latency, high scalability, high adaptability, and high extensibility. And RESTful Web services are those services that respect REST architectural style and use properly the HTTP protocol, URL identifiers, and lightweight representations. Finally, the usage of RESTful Web services does not guarantee, as same as W3C Web services, that the implemented services would be SOA-compliant.

2.4. Microservice architecture

James Lewis and Martin Fowler, in [14], state that "the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms."

In contrast to SOA, microservices do not appear as a new business architecture solution. Microservices appear to introduce an architectural solution to solve the typical problems of monolithic applications. Therefore, the main scope of microservices is to develop individual applications and not the entire business.

Nowadays, monolithic architectures are used successfully in a large number of applications; however, the use of cloud computing, together with the need of developing highly scalable Web applications, has demonstrated that monolithic architectures have difficulties to scale without affecting the general performance. Hence, other alternatives architectures, such as microservices, have been used to develop these highly scalable and cloud applications.

Instead of building the functionality together, microservices architectures separate each piece of functionality into isolated services. This separation is illustrated in Figure 4 where monolithic and microservices architectures are compared.

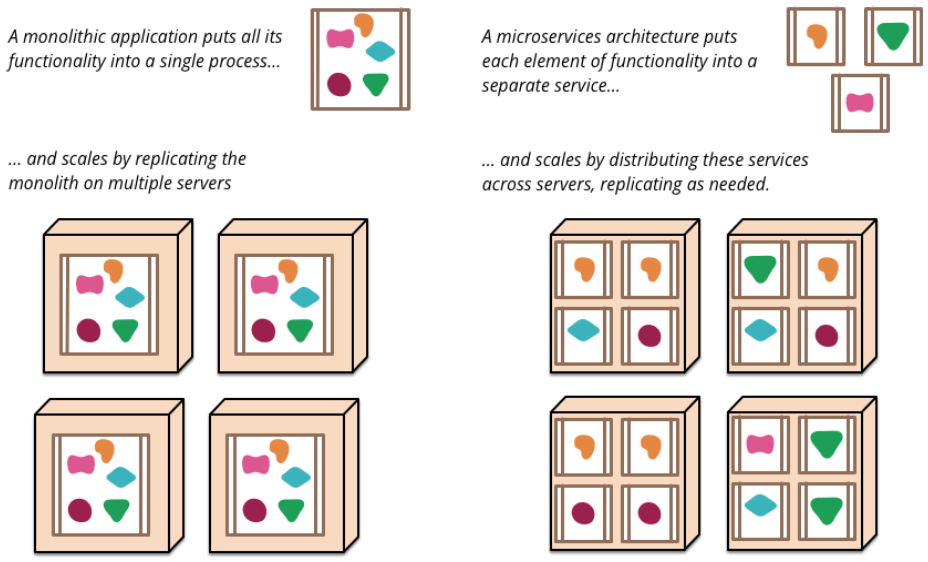


Figure 4. Comparison of monolithic and microservices applications (Source: [14])

After analyzing several "microservices" architectures, James Lewis and Martin Fowler realized that most of them exhibits nine common characteristics:

1. **Componentization via services.** For a long time of period, the software industry has sought new methods to develop systems by plugging components together. Essentially, a component can be seen as a unit of software that can be developed, deployed and maintained independently to other units. Microservices architectures use mainly services for componentization, rather than other mechanisms as libraries, because services are out-of-process components and, therefore, each of them can be deployed independently from the other ones. Nevertheless, services may also have dependencies between them, resulting in the impossibility of a complete independent deployment in some cases. Moreover, componentization via services may cause losses in efficiency and performance because remote calls through communication mechanisms are more much expensive than in-process calls.

2. **Organized around business capabilities.** Like SOA, microservices architectures also tend to divide its services considering the business capabilities. But, the main difference is that microservices architectures do not attempt to model and divide all the business operations, only single applications. As a result of this division by business capabilities, clear boundaries between services become visible and this allows forming cross-functional teams to take charge of each business capability and its services. These cross-functional teams include all the necessary skills to design, develop, maintain, and deploy microservices without the necessity of relying on other teams.

3. **Products not projects.** In general, microservices development are product-oriented and not project-oriented. A project-oriented development aims to deliver pieces of software that will be assembled afterwards. In contrast, a product-oriented development is completely created and maintained by the same team over its full lifecycle. Furthermore, this last approach fits perfectly with the division of services and teams by business capabilities.

4. **Smart endpoints and dumb pipes.** Whereas the communications used by traditional SOA services and W3C Web services use sophisticated smart mechanisms for message routing, choreography, and composition, among others; microservices architectures prefer the use of smart services (smart endpoints), but simple protocols and communication mechanisms (dumb pipes). For this reason, microservices architectures commonly use HTTP protocol or lightweight message buses with only routing capabilities such as RabbitMQ or ZeroMQ. Thus, microservices are very similar in this aspect to RESTful Web Services.

5. **Decentralized governance.** Microservices architectures have the tendency to seek the right tools and technologies for solving each individual task with the maximum possible efficiency. For this reason, this approach leads to find and apply non-standard solutions if they fit better than the standard ones. In any case, standards are not prohibited and, in fact, some of them are really used in microservices architectures. For instance, microservices tends to use some standards like HTTP, but tries to avoid other standards like provider contracts. As a consequence, some other patterns like Tolerant Reader or Consumer-Driven Contracts have to be applied in order to compensate somehow the functionality that these standard elements provide.

6. **Decentralized data management.** Monolithic architectures normally store the data in only one database with several tables that represent the complete conceptual model of the business. In contrast, microservices architectures would rather divide the conceptual model into several ones according to the business capabilities. As a result of this division, each single service may manage its own conceptual model avoiding unnecessary information, may decide which data storage (files, relational databases, non relational databases, etc.) should be used in each case, and may improve scalability because of the independence of services. But, on the other hand, this approach also presents one very important drawback: distributed transactions to obtain data consistency are really complex to implement. In order to avoid transactions, microservices architectures prefer to handle eventual data consistency and apply compensating operations when any inconsistency occurs.

7. **Infrastructure automation.** The automation of the development, deployment and management has evolved in last years due to the use of cloud computing and the increasing number of services. Typically, microservices architectures take advantage of these infrastructures in order to (1) reduce the complexity and time of developing and deploying high-quality services into production, (2) ensure that the new versions do not present regressions, and (3) avoid human errors caused by doing repeatedly and manually these tasks which may involve certain complexity.

8. **Design for failure.** The separation of functionality into several services causes the necessity of each service to be fault tolerant. This aspect is more relevant in microservices architectures than monolithic architectures because services run in different processes and use network communication mechanisms to interact with others, resulting in a larger number of failure points and a greater complexity to handle such failures. As a consequence, microservices architectures often use monitors and loggers to detect any abnormal behavior in real time with the purpose of restoring automatically the proper operation of the services that are failing.

9. **Evolutionary design.** Whenever the functionality is properly divided into several services and each service may be replaced or upgraded independently, the system can evolve regularly and progressively without the need of deploying the whole system every time. Thus, the planning of new features can be more granular avoiding large releases of the complete system that may cause problems in different points.

Although microservice architectural style and SOA style were created to solve different problems (scale monolithic applications vs. organize business processes, respectively), they propose two solutions with many commonalities like the use of services, the division by business capabilities, or the independence of specific technologies and languages, among others. Thus, SOA style may be applied to microservices architectures to obtain SOA-compliant microservices.

With respect to W3C Web services, microservices architectures usually differ from them because W3C Web services tend to use heavyweight standard technologies and communication mechanisms such as WSDL, SOAP, UDDI, and choreography, among others. In contrast, microservices architectures prefer simpler technologies and communication mechanisms to adapt and extend services faster.

On the other hand, microservices architectures share certain similarities with RESTful Web services like the use of more lightweight communication mechanisms. Therefore, microservices architectures can be implemented using RESTful Web services. This means that microservices architectures might respect REST style and take advantage of its properties: low latency, high scalability, high adaptability, and high extensibility.

To sum up, the term microservice does not refer to the size of the services, but it refer to an architectural style that divides the software in smaller pieces with the purpose of developing, deploying, managing and scaling up each piece independently to others. Finally, the usage of microservices architectures does not guarantee, as same as W3C Web services and RESTful Web services, that the implemented microservices would be SOA- and REST-compliant.

3. Microrestjs

The development of Web applications has changed substantially in recent years, according to Figure 5. The initial approach was to develop SOA-compliant services using standard W3C technologies with the purpose of achieving a real interoperability; however, despite its large potential and ambition, this initial approach have not had the expected success. Maybe, this large potential and ambition have been the main reasons of its failure because the resulting technologies were complex to learn and to use properly. When the interest in W3C Web services started to decrease, RESTful Web services began to be used increasingly as a valid alternative for building Web applications. This change of approach led to lose some interesting features, such as formal contracts or discoverability, due to the absence of standard technologies for those purposes. Thus, RESTful Web services began to use own interfaces or APIs for providing their functionality, resulting in the necessity of creating specific intermediaries to interact with each of them. In brief, the development of Web applications changed from one extreme to the opposite one. In general, RESTful Web services are easier to learn and to use property, but they do not provide other features such as automation or uniformity. Finally, microservices appeared to propose a new development approach for Web applications, but they continued with the trend of RESTful Web services of offering their own interfaces or APIs to interact and communicate between them.

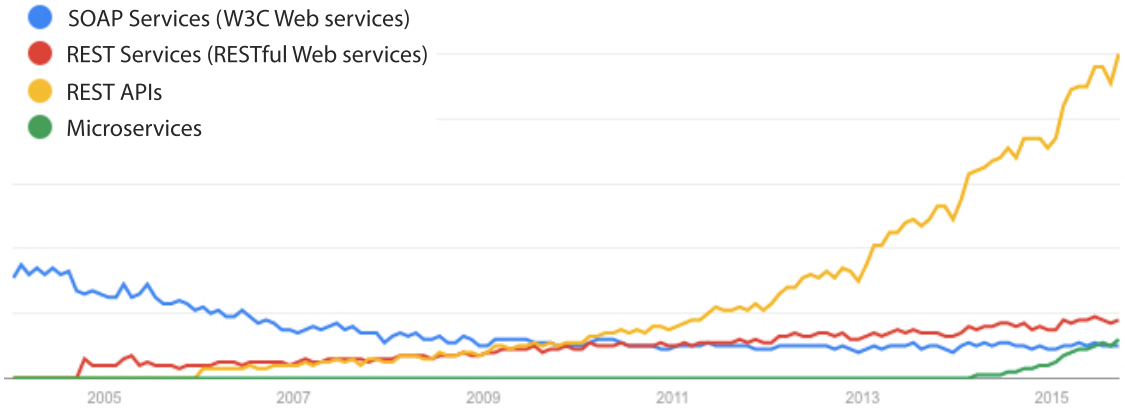


Figure 5. Interest in Web technologies (Source: Google Trends)

Based on this brief analysis of Web technologies, some conclusions may be drawn:

- Web developers seem to seek lightweight technologies to achieve low latency, high scalability, high adaptability, and high extensibility. Also, they may prefer technologies that are easy to learn and to use.
- Also, Web developers seem to prefer standard technologies, but only if they satisfy completely the needs without affecting the overall performance of the system.
- Finally, considering the rise of microservices and task automation, Web developers probably wish to use technologies for automating the development and deployment of software.

Considering the above premises, this thesis researches how to improve the current technologies in order to provide the following features:

- A method to design, develop and deploy microservices.
- A lightweight tool to support the design of microservices using simple contracts or descriptions.
- A lightweight tool to implement easily the functional logic of microservices without spending time in other secondary tasks such as network communication or discoverability.
- A uniform interface to interact with other microservices of the platform.
- An automated tool to deploy microservices into the platform.
- An automated tool to discover other microservices of the platform.
- A solution to offer complete freedom to Web developers with respect to the logic implementation and type of data storage.
- A solution to obtain compatibility with some previous technologies such as RESTful Web services and RESTful APIs.
- And some other out-of-the-box tools for logging, authentication, and authorization, among others.

As a result of this research, a new microservice-oriented platform, known as Microrestjs, has been created with the purpose of providing the above features. Microrestjs can be defined as a platform for designing, developing and deploying microservices. Basically, Microrestjs can be considered a platform because provides a set of software tools and technologies that facilitate the development and execution of microservices without depending on hardware. Concretely, Microrestjs provides a methodology, some tools for designing, developing and deploying, and some core services.

The Microrestjs methodology proposes a general idea about how to apply systematically the Microrestjs tools in the different phases of the development lifecycle. More details about the methodology are given in section 3.1. Methodology.

For designing, Microrestjs provides a Service Description Specification that defines how microservices should be described in order to be compatible with Microrestjs and take advantage of its automation features. More details about service descriptions are given in section 3.2. Service description.

For developing, Microrestjs provides a framework to help developers implement the logic of their microservices without worrying about secondary tasks such as network communication or service discoverability. More detail about the framework are given in section 3.3. Framework.

For deploying, Microrestjs uses a launcher included in the framework to run the implemented microservices and make them accessible from the network. More details about the launcher and the deployment process are given in section 3.3. Framework.

Finally, Microrestjs also provides some core services to offer developers automated features like service discoverability, authentication and authorization. More details about the core services are given in section 3.4. Core services.

3.1. Methodology

Microrestjs proposes a methodology to build microservices that can be easily integrated in the most common software development lifecycles. Essentially, Microrestjs suggests performing some tasks in the design phase, in the implementation phase, and in the deployment phase. For other phases like requirements or testing, Microrestjs does not recommend any specific task. Nevertheless, these other phases are also really important to guarantee the development of high-quality microservices.

On the one hand, the design phase aims to define how the system requirements may be divided and implemented using several microservices. In particular, Microrestjs does not specify how to conduct this division, but it is highly recommendable to separate them by business capabilities, as SOA and microservice architectural style suggest. In addition, developers should also consider the principles of SOA and the characteristics of microservice architectural style with the purpose of obtaining a real SOA-compliant and microservice-oriented design. Some of the main properties, which should be considered in this phase by developers, are: reusability, loose coupling, abstraction, composability, autonomy, statelessness, decentralization, and design for failure. For these properties, Microrestjs only provides general mechanisms that may help obtain them indirectly because these properties essentially depend on doing a good design and not on using a concrete technology. In contrast, Microrestjs does fully support formal contracts for describing the information and interfaces of the microservices through the Microrestjs Service Description Specification. Thus, the main specific task that should be added in the design phase is to write a service description for every single microservice in order to make them compatible with Microrestjs. Otherwise, Microrestjs will not be able to deploy automatically the microservices. Therefore, this decision of making the service descriptions mandatory has been taken to guarantee certain automated features (such as discoverability, network communication, and uniform invocations to microservices' operation, among others) and also to increase the overall quality of the microservices' documentation. Anyway, Microrestjs seeks to offer lightweight service descriptions (most of the aspects that can be described are optional) in order not to increase considerably the designing time. More details about the Microrestjs Service Description Specification can be seen in section 3.2. Service description.

On the other hand, the implementation phase is focused on writing the functionality of each individual microservice. For that, every microservice has to implement at least its public operations, which were defined previously in the service description. Otherwise, other microservices will not be able to execute those operations remotely. For this phase, Microrestjs provides a development framework that facilitates the implementation of microservices reducing the complexity of certain tasks such as the network communication or the invocation of remote operations. At the end, the purpose of Microrestjs is to allow developers

to focus exclusively on the real logic in order to reduce the implementation size and increase the overall performance and quality in terms of scalability, adaptability, extensibility, readability, maintainability, and testability, among others.

Finally, the deployment phase aims to run the implemented microservices and make them accessible from the network. For that, Microrestjs provides an automatic launcher that loads the service descriptions and the implementations into memory, registers the loaded microservices in a service directory, and routes the incoming HTTP requests to the proper microservices' operations. In short, Microrestjs offers specific tools to automatize this phase completely. At this point, it may be important to remark that once the microservices are deployed, any client and not only Microrestjs clients can execute the remote operations of the deployed microservices using HTTP protocol. This is possible because the service description defines a RESTful API that is deployed during this phase. In this way, Microrestjs assures certain compatibility with previous technologies and offers the possibility of automating the deployment of RESTful APIs as microservices.

Recapitulating, Microrestjs proposes a methodology that suggests some changes for some specific phases of the software development lifecycle. In addition, Microrestjs also provides some technologies in order to facilitate the development of microservices during the different phases. Finally, as a result of applying this methodology, Microrestjs runs the implemented microservices and makes them accessible from the network.

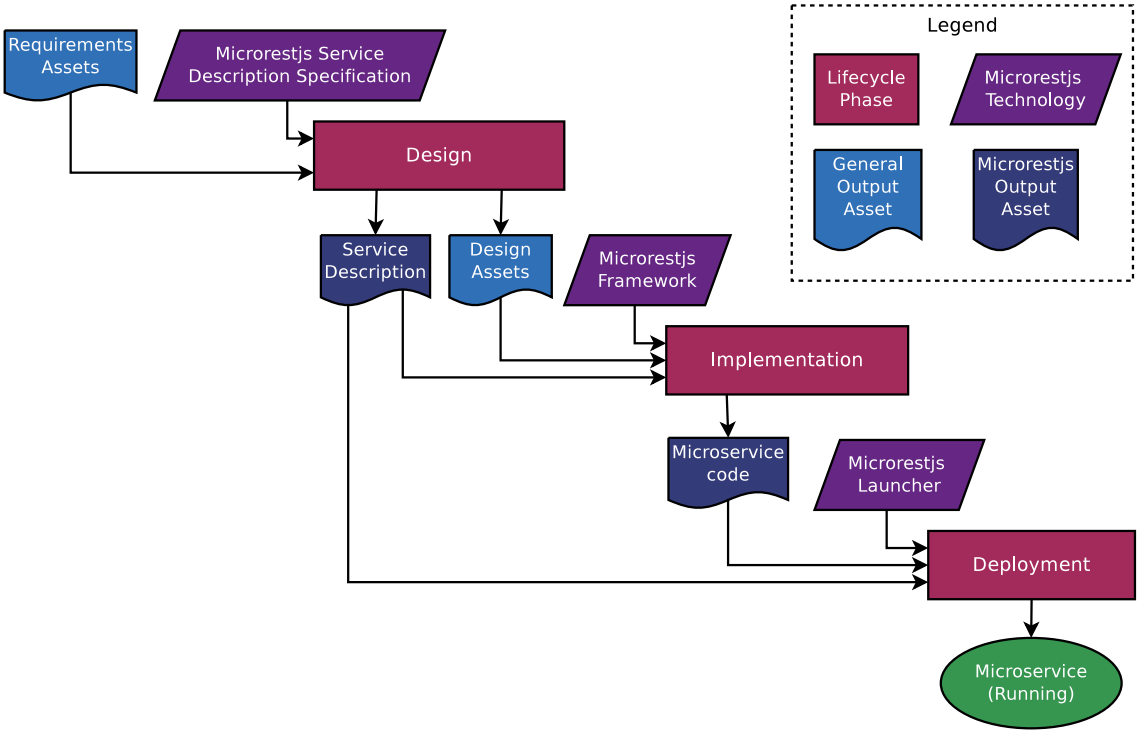


Figure 6. Microrestjs methodology

3.2. Service description

A service description is a formal contract that specifies the characteristics and behavior of a microservice. Like any other formal contract, service descriptions should present a uniform format and structure in order to be able to use them systematically. In our case, the format and structure are defined in the Microrestjs Service Description Specification, which can be consulted in the appendix *B. Microrestjs Service Description Specification.*

Microrestjs proposes to use JavaScript Object Notation (JSON) for writing service descriptions. JSON can be defined as a lightweight key-value format to store and exchange data. The decision of using JSON has been taken because it is maybe the most popular format for Web applications, it is easy for humans to read and write, it is easy for machines to parse and generate, and it is more lightweight than XML.

Basically, Microrestjs service descriptions are structured in three parts: information, configuration, and operations. On the one hand, the information part describes general information about the microservice, as for example, the name, the version, and the author, among others. Thus, this part can be seen as the microservice metadata. On the other hand, the configuration part describes some configuration aspects like the location and dependencies. Finally, the operations part defines all the public operations that can be executed remotely, i.e. the RESTful API of the microservice. For each operation, several aspects about the request and responses have to be specified in order to declare the intended behavior of the microservice. For more information about the format and structure of the service descriptions, please consult the Microrestjs Service Description Specification.

The use of formal service descriptions presents several advantages for providers and consumers. On the one hand, providers can verify whether the implementations of their microservices respect the intended behavior declared by the corresponding service descriptions. On the other hand, consumers can use these service descriptions to obtain information about the microservices and how to interact with them, in other words, the signatures and expected behavior of their public operations. Basically, all these advantages are really interesting for documentation, but they do not provide real competitive advantages over common RESTful APIs. The real competitive advantage of using service descriptions is the possibility of automating certain tasks. In particular, Microrestjs service descriptions are used for automating the deployment of microservices, the discoverability of other microservices, and the interaction and communication with other microservices.

Like any other tool, formal service descriptions also present some drawbacks. The main problem is that developers are forced to write these service descriptions to enable the automation capabilities. Summarizing, these service descriptions are not used only for documentation purpose, but they also provide certain functionality. For this reason, they should be written carefully in order to avoid specification mistakes that can lead unexpected behaviors. Other common problem that developers face every day is the necessity of using different specifications and tools to obtain a complete documentation and overview of the system. Microrestjs

Service Description Specification has been designed considering all these drawbacks with the only purpose of facilitating the design and development of microservices. For instance, Microrestjs Service Description Specification suggests a lightweight format for writing service descriptions that favors the creation of service descriptions progressively in several iterations. On the other hand, Microrestjs Service Description Specification has also attempted to minimize the dependency from other specifications and tools. For that, Microrestjs Service Description Specification has taken into account several existing specifications such as Swagger Specification, WSDL, JSON-WSP, Bower Manifest, Ruby Gem Specification, NPM Package Specification, IO Docs, and RAML.

Definitely, Microrestjs Service Description Specification has been designed carefully in order to provide useful service descriptions that can be used for documentation and automation purposes without requiring a large effort from developers. In this way, Microrestjs pursues to provide the first tool that helps developers design microservices.

3.3. Framework

Microrestjs Framework is a developer-oriented tool to implement and deploy microservices that respect the Microrestjs Service Description Specification. Essentially, Microrestjs Framework can be considered a black-box framework that can be adapted and extended easily to include the microservices implemented by developers. On the other hand, Microrestjs Framework also provides a launcher that executes the framework deploying the implemented microservices.

Next subsections describe in detail what features are provided by this framework, why those features are supported by the framework and how those features have been implemented by the framework.

3.3.1. Analysis

The main purpose of Microrestjs Framework is to implement and deploy microservices. For that, the framework implements a set of features that are analyzed in this subsection.

Firstly, the Microrestjs Framework provides to developers the freedom of choosing how their microservices are deployed. Basically, the Microrestjs Framework has a configuration file that allows deploying several instances of the framework on the same network host. In addition, the framework also supports the deployment and management of one or more microservices in every instance as if they had been deployed in independent instances. For more information about the configuration file, please consult C. Microrestjs Framework Configuration.

Secondly, the framework supports the complete Microrestjs Service Description Specification with the purpose of processing the service descriptions written by developers and automating the deployment of microservices. As a result of supporting the Microrestjs Service Description Specification, the following features are necessary:

1. **Load and parse JSON documents.** Both the configuration files of the framework and the service descriptions are written and stored using JSON format; consequently, Microrestjs Framework loads, parses and handles JSON documents.
2. **Verify service descriptions.** The service descriptions must entirely respect the Microrestjs Service Description Specification in order to obtain the automation capabilities; consequently, Microrestjs Framework checks whether a service description respects the Microrestjs Service Description Specification.
3. **Manage the dependencies of the microservices.** Developers can specify the dependencies of the microservices in the service descriptions with the purpose of obtaining uniform interfaces to interact transparently with them; consequently, Microrestjs Framework manages those dependencies and provides the corresponding uniform interfaces for them.
4. **Handle authentication and authorization protocols.** The Microrestjs Service Description Specification allows specifying how the requests for executing operations have to be authenticated and authorized; consequently, Microrestjs Framework handles transparently the authentication and authorization protocols supported by the Microrestjs Service Description Specification, and interacts transparently with remote authentication and authorization service if it is required.
5. **Deploy the microservices as RESTful APIs.** The service descriptions contain all the necessary information to deploy the microservices operations as RESTful APIs; consequently, the Microrestjs Framework deploys automatically the microservices and makes them accessible through the network.
6. **Register the microservices in the service directory.** The service descriptions contain also the necessary information to register the microservices in a service directory with the purpose of being found and used by other microservices; consequently, the Microrestjs Framework interacts transparently with the service directory.

On the other hand, the framework provides an easy mechanism that allows developers to implement microservices compatible with the platform. This mechanism does not require knowing the entire framework to integrate the microservices, but it provides a unique place or hot spot with a basic logic that can be extended by developers to include the microservice functionality. Traditionally, this integration of services with the rest of the framework or system has been achieved generating skeletons and stubs at compile time. Although this traditional solution can offer some advantages like type checking at compile time, it is not an optimal solution from developers' point of view because it requires more effort and time from them. Therefore, the Microrestjs Framework proposes to load and deploy dynamically the functionality without requiring previous compilation tasks. In short, the framework pursues that developers have to focus exclusively on writing the service description and implementing the service logic, and not on other secondary tasks like compilation or deployment.

Moreover, once the microservices have been deployed, the Microrestjs Framework handles all the incoming HTTP requests, delegates the execution of the operations to the microservices, and sends the HTTP responses with the results of the operations back. Also, the framework manages the network errors, serializes and deserializes data, and checks the correctness of the HTTP headers, among others.

Likewise, the Microrestjs Framework provides transparent mechanisms to make requests to remote microservices through the uniform interfaces created during deployment phase for the dependencies.

In order to provide a safe and secure platform, the Microrestjs Framework complements the previous core features with several security measures. The most important implemented security features are:

1. **Enable secure communications over TLS.** All the communications from and to a microservice must be encrypted to guarantee the confidentiality and integrity of the transmitted data. In recent years, the need of secure communications has increased due to the boom of e-commerce and online banking, and the disclosure of global surveillance programs. For these reasons, the trend is to offer these days secure communications even for non-critical services. Thus, the Microrestjs Framework always provides secure network communications over TLS transparently for users and developers by default.
2. **Protect against information leakage.** The information leakage is a type of system weakness that can be attacked to obtain private information without authorization. Usually, information leakage is caused by defects in design and implementation. Therefore, in order to minimize this kind of defects, the Microrestjs Framework has been designed and implemented carefully considering this weakness. For instance, the framework avoids that microservices can access and take the control of the platform by-passing the defined interfaces.
3. **Ensure the information privacy.** The Microrestjs Framework has the capability of deploying several microservices at the same time, as was mentioned before. For this reason, the framework ensures that the information of each microservice remains private and can be exclusively accessed by the own microservice through its operations. In consequence, microservices are the only ones responsible for sharing their own information with third parties.
4. **Execute the microservices inside "sandboxes".** Because of the same reason pointed in 3, the Microrestjs Framework executes the deployed microservices separately like in "sandboxes" in order to execute exclusively the public operations described in the service descriptions through HTTP requests and deny the execution of private operations as well as the invocation of public operations using any other protocol or invocation form. Consequently, these "sandboxes" help protect against information leakage and ensure the information privacy of the microservices.
5. **Authentication and authorization services.** As was stated before, the microservices may require authenticating and authorizing the incoming requests. For this reason, Microrestjs Framework handles the authentication and authorization protocols supported by the Microrestjs

Service Description Specification, and interacts with the authentication and authorization service if it is required.

Apart from all these main features, there are other secondary features that complement the framework with small but significant enhancements. For instance, the Microrestjs Framework respects the REST principles with the purpose of being completely compatible with all the existing Web technologies such as proxies, gateways, load balancers, and caches, among others. Besides, the framework does not restrict the type of microservices that can be developed with it because it is a general-purpose framework. Furthermore, the framework does not limit the external libraries that the microservices can use for implementing their business logic. Similarly, the framework does not impose a specific data management technology to persist the resources of the microservices. On the other hand, the framework does not hinder the integration between microservices and previous systems. Finally, the framework also pursues to avoid technical lock-ins in order to foster IT innovation and development.

Summarizing, the Microrestjs Framework needs both the main features and the secondary ones in order to obtain a complete framework that can be used for implementing and deploying microservices with a high degree of reusability, adaptability, extensibility, safety, security, scalability, availability, interoperability portability, reliability, fault tolerance, and maintainability, among others.

3.3.2. Architecture

At this point, the reader knows roughly what features are included in the Microrestjs Framework, but he might not have a precise picture of how all those features are assembled to build the platform. In order to complete this picture and understand how the platform works, this subsection presents the architecture of the platform using some diagrams to explain visually the structure of the platform.

Figure 7 illustrates how the main components of the platform can be distributed at runtime. Basically, the platform can be composed of servers, clients, gateways, and proxies. On the one hand, the servers run the Microrestjs Framework instances and, consequently, the microservices. On the other hand, the clients interact with the servers in order to execute the operations of the microservices. Finally, the gateways and proxies act as intermediaries between servers and clients to preprocess the HTTP requests and responses, if it is necessary.

In this case, Figure 7 shows two local servers running on the local area network and one external server running on the Internet. Besides, it shows two clients, one local and one external, that can interact with the servers, both local and external ones, to execute the operations of the microservices. On the one hand, each server can deploy one or more instances of the Microrestjs Framework using different network ports. On the other hand, each Microrestjs Framework instance can deploy and manage one or more microservices according to the settings of the configuration file. Finally, the diagram also specifies explicitly the use of the HTTPS protocol, i.e. HTTP over TLS, for all the communications between the different components of the platform.

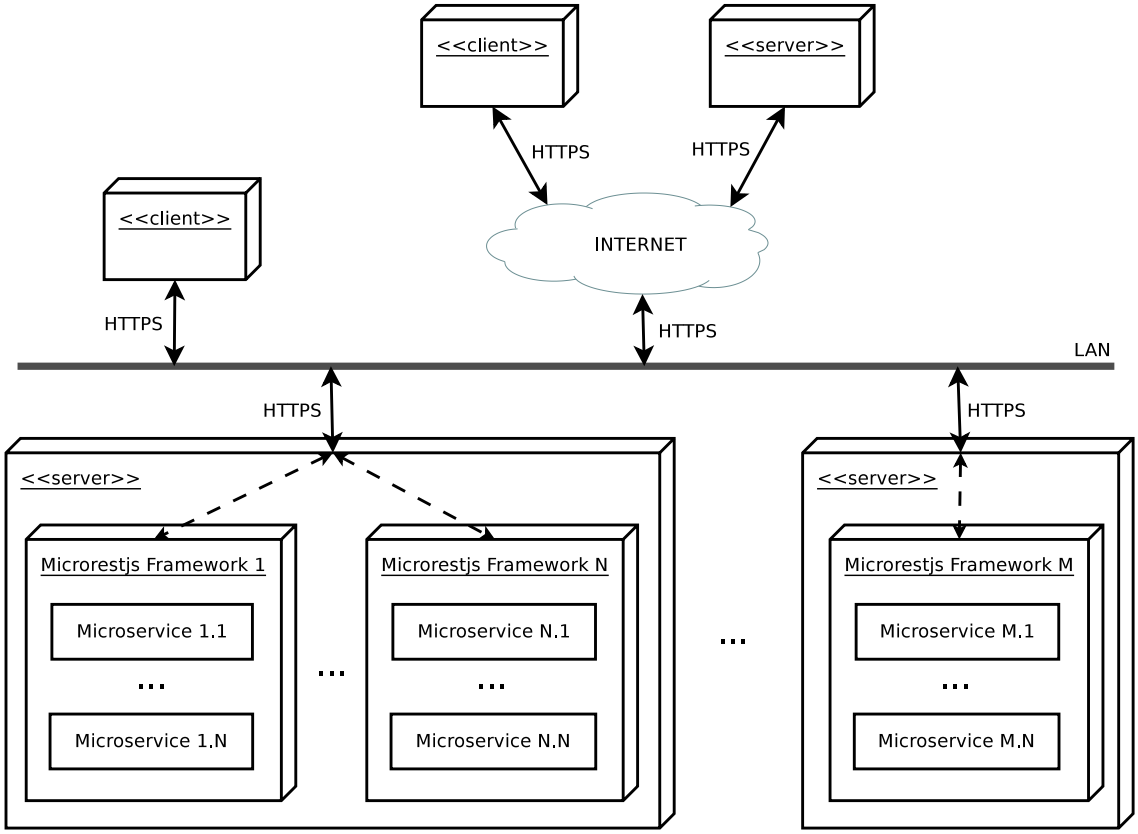


Figure 7. Microrestjs deployment diagram

After analyzing the above diagram, the reader may have noticed that the platform provides a high level of customization for distributing and using microservices. As a consequence, the platform is able to facilitate the development of highly scalable, available and reusable microservices-oriented systems. Moreover, the platform also facilitates the communication between microservices, resulting in a larger versatility in terms of abstraction and composability.

Once the physical view of the platform with all the involved components have been explained, a more logical and simpler view can be presented in order to show exclusively the deployed microservices without considering their physical location. Figure 8 shows some microservices and clients connected to a network. Specifically, the diagram details three particular microservices (the service directory, the authentication service, and the authorization service), which are required by most of the systems. These three microservices have been considered completely necessary for the basic operation of the platform and, for this reason, they are provided as core services. For more details about these core services, please consult the section 3.4. Core services.

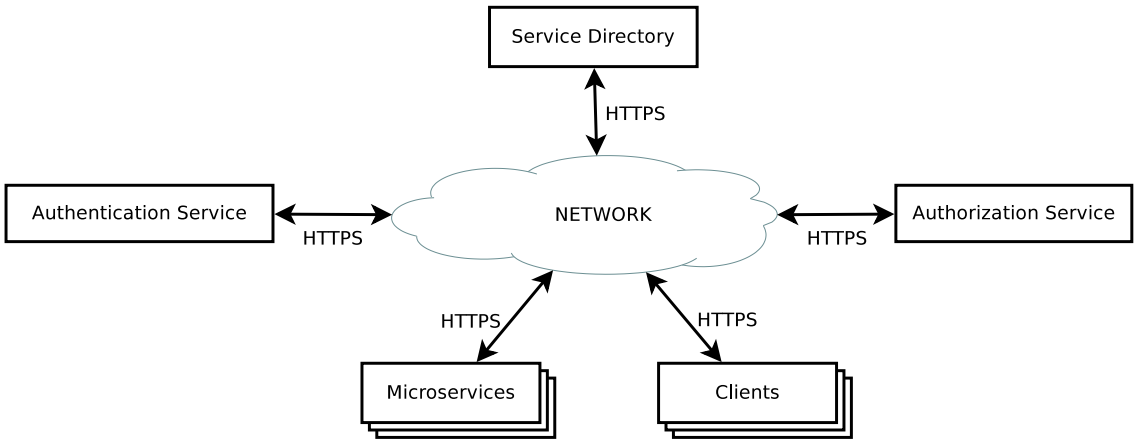


Figure 8. Microrestjs logical view

At this point, the reader knows which components may be part of the platform, how those components can be distributed, which network protocols are used for communication, and which core services are commonly deployed in most of the systems. Thus, the only aspect that remains to be defined is the architecture of the Microrestjs Framework. Figure 9 defines such architecture locating the different modules into three layers: the runtime layer (bottom layer), the framework layer (middle layer), and the microservices layer (upper layer). First, the runtime layer is composed of a JavaScript runtime environment, called Node.js, which is in charge of executing the upper layers. Second, the framework layer is composed of the Express Framework, the Node.js Packages, and the Microrestjs Framework itself. Finally, the microservices layer is composed of all the microservices that are deployed by the Microrestjs Framework instance.

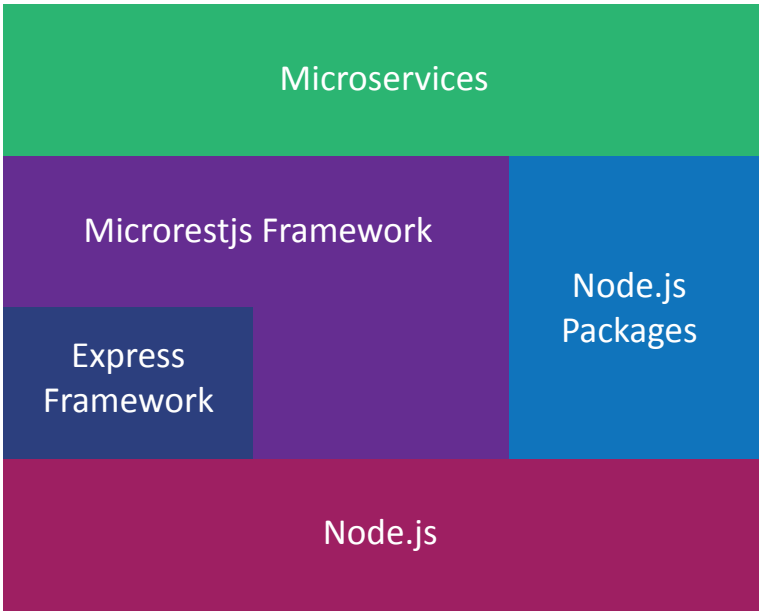


Figure 9. Microrestjs Framework architecture

For this thesis, the middle layer is the most important one because it is the responsible for providing the platform with all the characteristics described in previous sections. As can be observed, the distribution of this middle layer is not completely uniform. On the one hand, the Microrestjs Framework is in contact with the upper layer because it provides a uniform API for developing and deploying all the microservices. On the other hand, the Microrestjs Framework is in contact with

the bottom layer, the Express Framework and the Node.js Packages because all of them are used somehow for implementing all the features provided by the Microrestjs Framework. For instance, the bottom layer allows developing cross-platform Node.js applications using the Node.js API. The Express Framework facilitates the development of Web applications in Node.js. And, the Node.js Packages are third-party libraries for Node.js that provide different out-of-the-box functionality. Finally, the Node.js Packages have also contact with the upper layer because all the existing Node.js Packages can be used for implementing the own functionality of the microservices. Summarizing, the middle layer provides a uniform API to hide the behind complexity that exist for developing and deploying microservices.

From an architectural point of view, all the important aspects of the platform have been already discussed and, therefore, the next step is to detail the design and operation of the Microrestjs Framework at runtime.

3.3.3. Detailed design

This subsection explains in detail the design of the Microrestjs Framework, the most challenging part of this thesis. First, the packages of the framework are presented in a package diagram, together with the classes that compose the framework. Then, those classes are organized in a class diagram that shows the static structure of the framework. Finally, several sequence diagrams are introduced in order to show the interaction of the classes at runtime.

Figure 10 illustrates how the framework has been divided into several modules or packages. The figure shows five main packages: Platform, Services, Messages, Helpers, and Node.js Packages. The Platform package contains the main classes for managing the whole platform. The Services package contains the classes for representing the services and their information. The Messages package contains the classes for managing the HTTP requests and responses. The Helpers package contains additional classes for facilitating the development of the platform. Finally, the 'Node.js Packages' package is composed of third-party libraries for Node.js.

The Platform package contains four classes: Microrest, ServiceManager, Server, and Client. The Microrest class initiates and controls the rest of components of the platform. The ServiceManager class loads, deploys, registers and manages the microservices of the platform. The Server class creates and runs a Web server that listens for HTTP requests in order to route them to the appropriate microservice operation. Finally, the Client class allows sending HTTP requests to remote microservices and receiving the corresponding HTTP responses.

The Services package contains four classes: ServiceContext, Service, RunnableService, and CallableService. The ServiceContext class is a data structure that stores and manages the service descriptions in memory. The Service class is an abstract representation of the microservices that contains common properties like the service context. The RunnableService class represents the deployed microservices and contains all the public operations described in the service description. Finally, the CallableService class represents the remote microservices

that has been added as dependencies in the service descriptions, and provides a uniform interface to execute such microservices remotely.

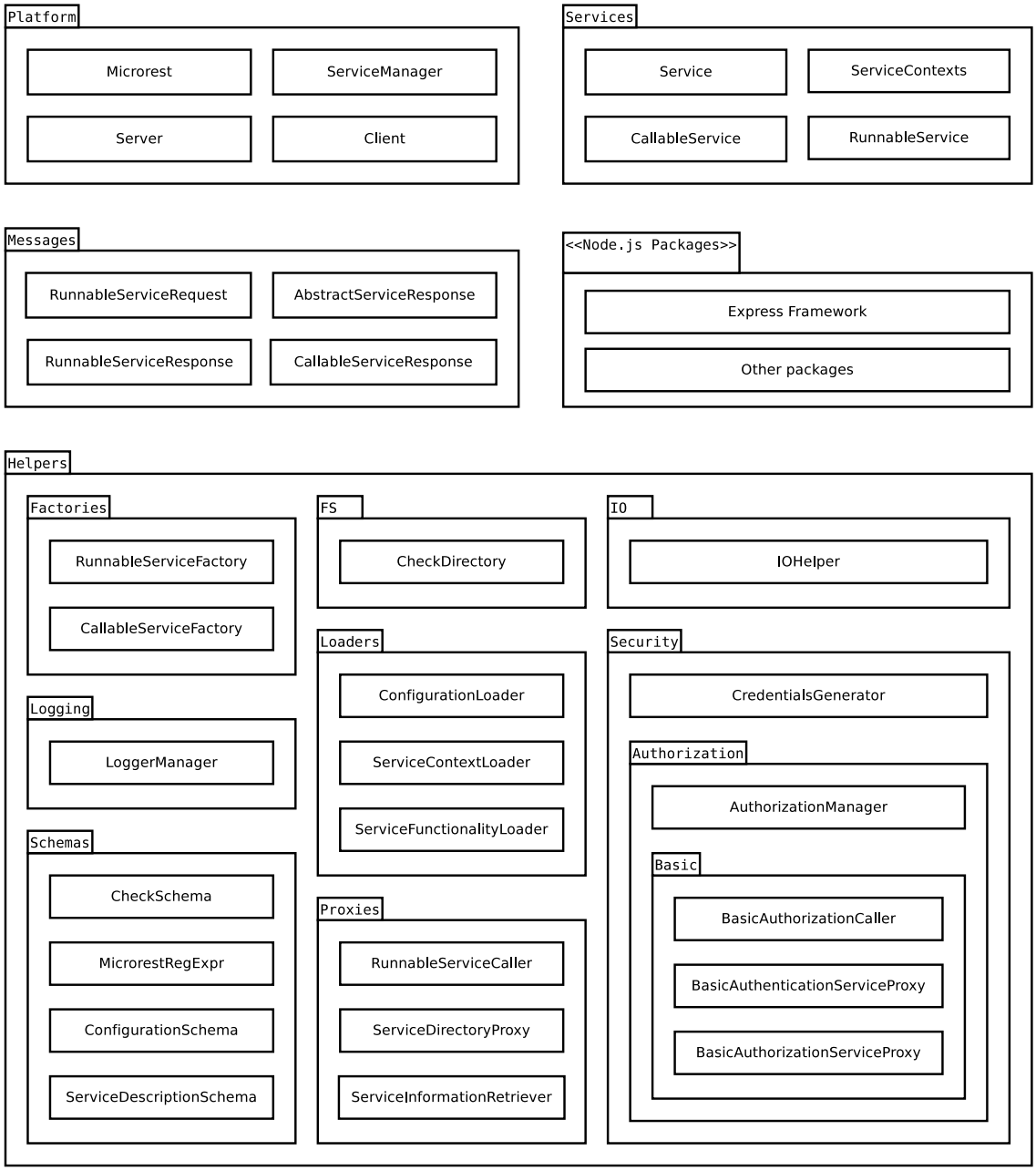


Figure 10. Package diagram

The Messages package contains four classes: RunnableServiceRequest, AbstractServiceResponse, RunnableServiceResponse, CallableServiceResponse. The RunnableServiceRequest class represents the received HTTP requests that any deployed microservice has to execute. The AbstractServiceResponse class is an abstract representation for HTTP responses. The RunnableServiceResponse class represents the HTTP responses that are sent by the deployed microservices. Finally, the CallableServiceResponse class represents the HTTP responses that are received from remote microservices.

The Helpers package is divided into eight subpackages: Factories, Loaders, Schemas, Proxies, Security, IO, FS, and Logging. The Factories package contains

classes for instantiating both the RunnableServices and the CallableServices. The Loaders package contains classes for loading files in memory from the file system. The Schemas package contains classes for checking whether every JSON object satisfies its schema. The Proxies package contains classes for facilitating the incoming and outgoing communications. The IO package contains classes for handling HTTP requests and responses. The Security package contains classes for increasing the security measures. The FS package contains classes for interacting with the file system. Finally, the Logging package contains classes for managing the logs of the platform and the microservices.

The Factories package contains two classes: RunnableServiceFactory and CallableServiceFactory. On the one hand, the RunnableServiceFactory class loads and instantiates the RunnableServices that will be deployed by the framework. On the other hand, the CallableServiceFactory creates the CallableServices to facilitate the execution of remote microservices.

The Loaders package contains three classes: ConfigurationLoader, ServiceContextLoader, and ServiceFunctionalityLoader. The ConfigurationLoader class loads the configuration file of the platform into memory from the file system. The ServiceContextLoader class loads service descriptions into memory from the file system. Finally, the ServiceFunctionalityLoader class loads the functionality of the microservices into memory from the file system.

The Schemas package contains two classes: CheckSchema and MicrorestRegExpr. The CheckSchema class checks whether a JSON object satisfies a specific JSON Schema. The MicrorestRegExpr class provides several regular expressions used by the framework and schemas to check the correctness of the data. This package also contains two JSON schemas: ConfigurationSchema and ServiceDescriptionSchema. The ConfigurationSchema is a JSON schema to verify the configuration file of the platform. Finally, the ServiceDescriptionSchema is a JSON schema to verify the service descriptions according to the Microrestjs Service Description Specification.

The Proxies package contains three classes: RunnableServiceCaller, ServiceDirectoryProxy, and ServiceInformationRetriever. The RunnableService-Caller class is a skeleton (server proxy) that facilitates the execution of deployed microservices when an HTTP request is received. The ServiceDirectoryProxy class is a stub (client proxy) that facilitates the interaction with the service directory with the purpose of registering and looking up microservices. Finally, the ServiceInformationRetriever class is a stub (client proxy) that facilitates the interaction with remote microservices in order to obtain information about them.

The Security package contains one class, CredentialsGenerator, which facilitates the creation of the platform credentials (private key and certificate) for secure communications. This package also contains one subpackage, Authorization, which manages transparently the authentication and authorization of the requests for executing the microservices.

Finally, the last three packages contain only one class. The IO package contains the IOHelper class to manage all the HTTP requests and responses uniformly. The FS

package contains the CheckDirectory class to check if a path is a directory or not. And the Logging package contains the LoggerManager class to manage the logs of the platform and the microservices.

At this point, based on the concise description of the packages and classes, the reader may be able to intuit how certain parts of the platform works together. Nevertheless, the reader might need a more precise diagram in order to understand the complete structure of the platform and the relationships between the different classes.

Figure 11 illustrates a class diagram with the structure of the platform and the relationships between the different classes. The start point is the Launcher class that initiates the platform using the Microrest class. Then, the Microrest class loads the configuration (using the ConfigurationLoader), generates the platform credentials for secure communications (using the CredentialsGenerator), configures the logs (using the LoggerManager), and initiates the ServiceManager and the Server. On the one hand, the ServiceManager class loads all the microservices using the RunnableServiceFactory, deploys such microservices into the Server, and registers them in the service directory (using the ServiceDirectoryProxy). On the other hand, the Server class initiates an HTTPS server to listen for requests through Express Framework, checks if the received requests have authorization for execution (using the AuthorizationManager), and finally delegates the request to the appropriate microservice operation (using the RunnableServiceCaller). For its part, the RunnableServiceFactory loads the service descriptions (using the ServiceDescriptionLoader) and functionality of the microservices (using the ServiceFunctionalityLoader), obtains the CallableServices (using the CallableServiceFactory), and finally instantiates the RunnableServices with the corresponding context information, functionality and CallableServices. Finally, the CallableServices interact with the remote microservices using the Client, but first they have to obtain the real location of the microservices from the service directory (using the ServiceDirectoryProxy) and retrieve their service information (using the ServiceInformationRetriever).

The above description only tries to show briefly the structure of the platform and the relationships between the different classes in order to understand superficially how the platform works. A complete description about the operation of the platform is given later, but first some important design decisions about the structure should be commented and analyzed.

At the beginning of the section, the Microrestjs Framework was described like a black-box framework that can be adapted and extended easily to include the microservices implemented by developers. Usually, the black-box frameworks provide specific parts, also known as hot spots, which can be adapted and extended with new functionality. The rest of the parts are frozen and cannot absolutely be changed. In our case, the Microrestjs Framework provides exclusively one hot spot to plug in the real functionality of the microservices. In the class diagram, the hot spot is indicated with the class ‘Service A’ that can be substitute by any other microservice.

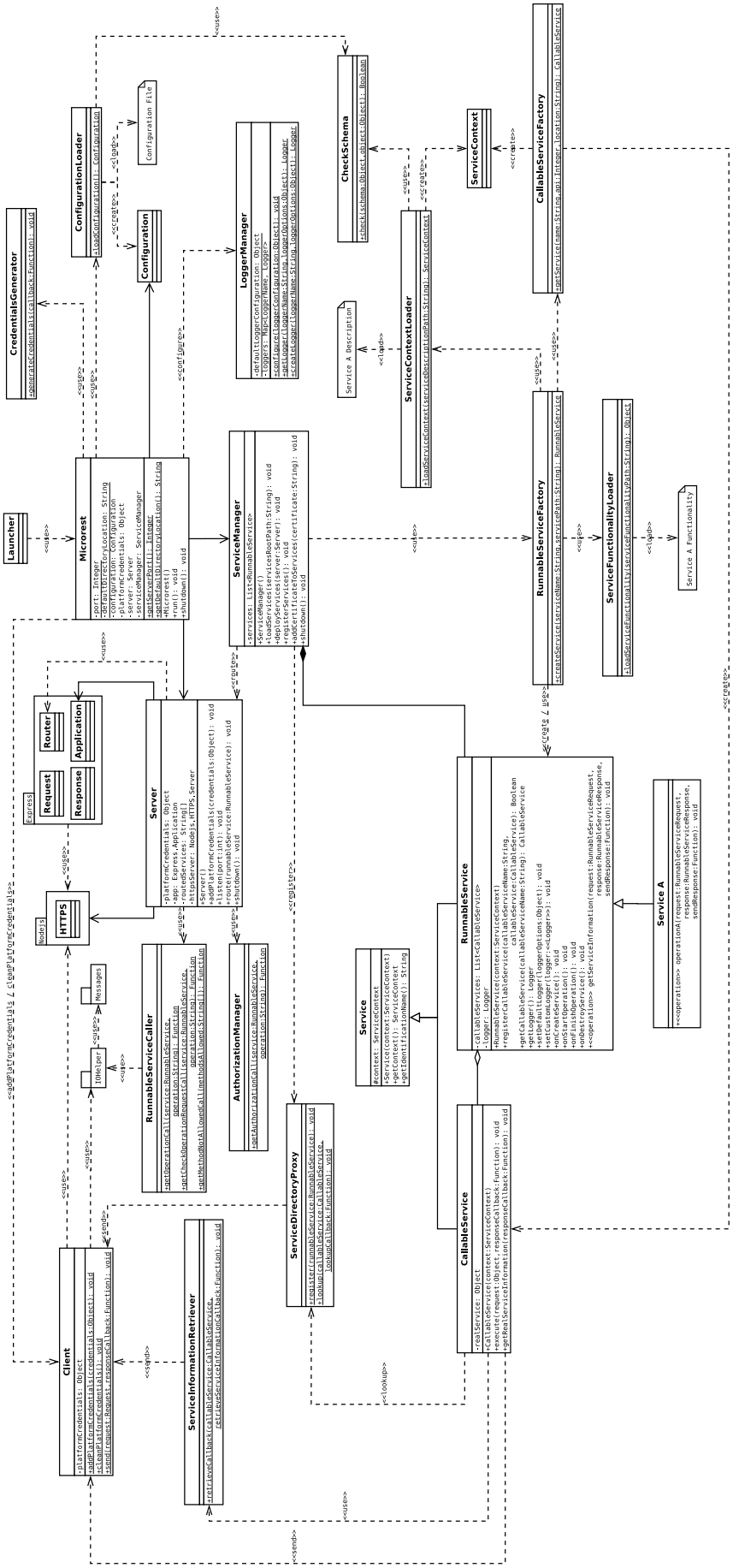


Figure 11. Class diagram

Moreover, the Microrestjs Framework also presents another typical framework characteristic known as Inversion of Control (IoC). Basically, the IoC is a design principle that changes the control flow from the user's application to the framework. Accordingly, the framework acts as the main program deciding how and when the user's application code is executed. In our case, the Microrestjs Framework applies the IoC principle in order to control the microservices (the user's application). The application of this principle brings several advantages for the platform. In first place, IoC facilitates the creation of hot spots, which results in a larger modularity and decoupling. In second place, IoC provides complete control to the framework on how the microservices are deployed and executed when the HTTP requests are received. In third place, IoC facilitates the automation of the different tasks such as the execution of remote microservices, the registration in the service directory, and the authorization of the requests, among others. And finally, in fourth place, IoC enables an effective separation of microservices in "sandboxes" allowing the management of several microservices at the same time without sacrificing security and privacy.

Furthermore, the Microrestjs Framework has been designed taking into consideration the problem of information leakage. As can be observed in the class diagram, the framework has the control of all the deployed microservices because of the composition relationship between the ServiceManager and RunnableService classes. By design, the framework must avoid absolutely the access of any malicious RunnableService to the Microrest, Server or ServiceManager class. Otherwise, any malicious RunnableService will be able to take the control of the platform and skip all the security mechanisms in order to access directly to the private information of other microservices. For preventing this scenario, this design separates completely the RunnableServices from the rest of the framework. As a consequence, the RunnableServices can be seen as independent units of code that can be executed exclusively by the framework in a controlled environment or "sandboxes". Maybe, the reader could be wondering about the relationship between the RunnableService and CallableService classes. Nevertheless, such relationship is not problematic because CallableServices cannot take the control of the platform, directly or indirectly. Summarizing, there is only one way of executing a microservice and obtaining information from it; it is through HTTP requests that are received by the framework, preprocessed for security reasons, and finally delegated to the appropriate microservice.

Apart from the previous security measures, the Microrestjs Framework also aims to provide secure network communications. For that, the framework uses three different classes: the CredentialsGenerator, the Server and the Client. First, the CredentialsGenerator class creates the necessary resources for establishing secure communications over TLS: a RSA private key and a X509 certificate. Then, the Server class uses the generated private key and certificate to instantiate an HTTPS server to accept and establish exclusively secure communications over TLS. Finally, the Client is also configured to encrypt all the communications with other microservices using the generated private key and certificate. At this point, the use of the TLS protocol can seem trivial; however, it requires configuring properly several elements to guarantee real secure communications. First, the platform must use 2048-bit RSA keys and SHA-256 X509 certificates in order to be safe the

next few years, according to the recommendations from NIST [15], ANSSI [16] and BSI [17]. Second, the platform should support TLSv1.1 and TLSv1.2 exclusively and avoid the use of SSLv2, SSLv3 and TLSv1.0 to prevent certain attacks, as for example POODLE. Third, the platform should only accept a safe list of cipher suites; for that, the platform can use the default list of Node.js because such list contains only cipher suites that are considered completely safe at this moment. Finally, with regard to the key-agreement protocol, the platform should avoid Diffie-Hellman (DH) protocol and use Elliptic Curve Diffie-Hellman (ECDH) in order to be protected from Logjam attack that was reported on May 2015; for that, the platform can use again the default configuration of Node.js without any change. Recapitulating, the platform is able to transparently provide secure communications over TLS for all the deployed microservices without any additional configuration.

As can be observed, the Microrestjs Framework has been designed carefully to include all the required features without sacrificing its quality in terms of simplicity, modularity, and security, among others. Basically, this has been possible because the Separation of Concerns principle has been applied in order to divide properly the functionality into the different classes that compose the framework; as a result, the design presents high modularity and low coupling.

Once the most important design decisions have been commented, the reader should be able to understand the overall structure of the Microrestjs Framework and how it works. In any case, the following sequence diagrams describe exhaustively how all the classes of the platform work together to deploy and run the microservices.

Figure 12 illustrates the startup process of the Microrestjs Framework. This process begins with the execution of the Node.js environment with the purpose of running the Launcher of the framework. The Launcher creates and runs a Microrest instance in order to manage the whole platform. During the creation of the Microrest instance, the Microrest constructor:

1. Loads the configuration of the platform using the ConfigurationLoader. During this step, the ConfigurationLoader uses the CheckSchema to verify against the Configuration Schema if the values of the configuration are correct for the execution of the platform.
2. Configures the LoggerManager to be able to provide default logs for the framework and the microservices.
3. Generates asynchronously in background the credentials of the platform, i.e. the RSA private key and the X509 certificate, using the CredentialsGenerator. The reason for designing this step asynchronous in background is because the generation of the credentials requires some time that could be used for executing the following steps.
4. Creates a Server instance to listen for the incoming HTTP requests. During the instantiation, the Server creates an Express Framework instance to facilitate the management and routing of the incoming HTTP requests towards the deployed microservices.
5. Creates a ServiceManager instance to load, deploy, register and centrally manage all the microservices of the platform.

6. Loads the microservices into memory using the ServiceManager. This step is meticulously described in Figure 13.
7. Deploys the loaded microservices into the Server using the ServiceManager. This step is meticulously described in Figure 14.
8. Finally, registers the deployed microservices in the service directory using the ServiceManager. This step is meticulously described in Figure 15.

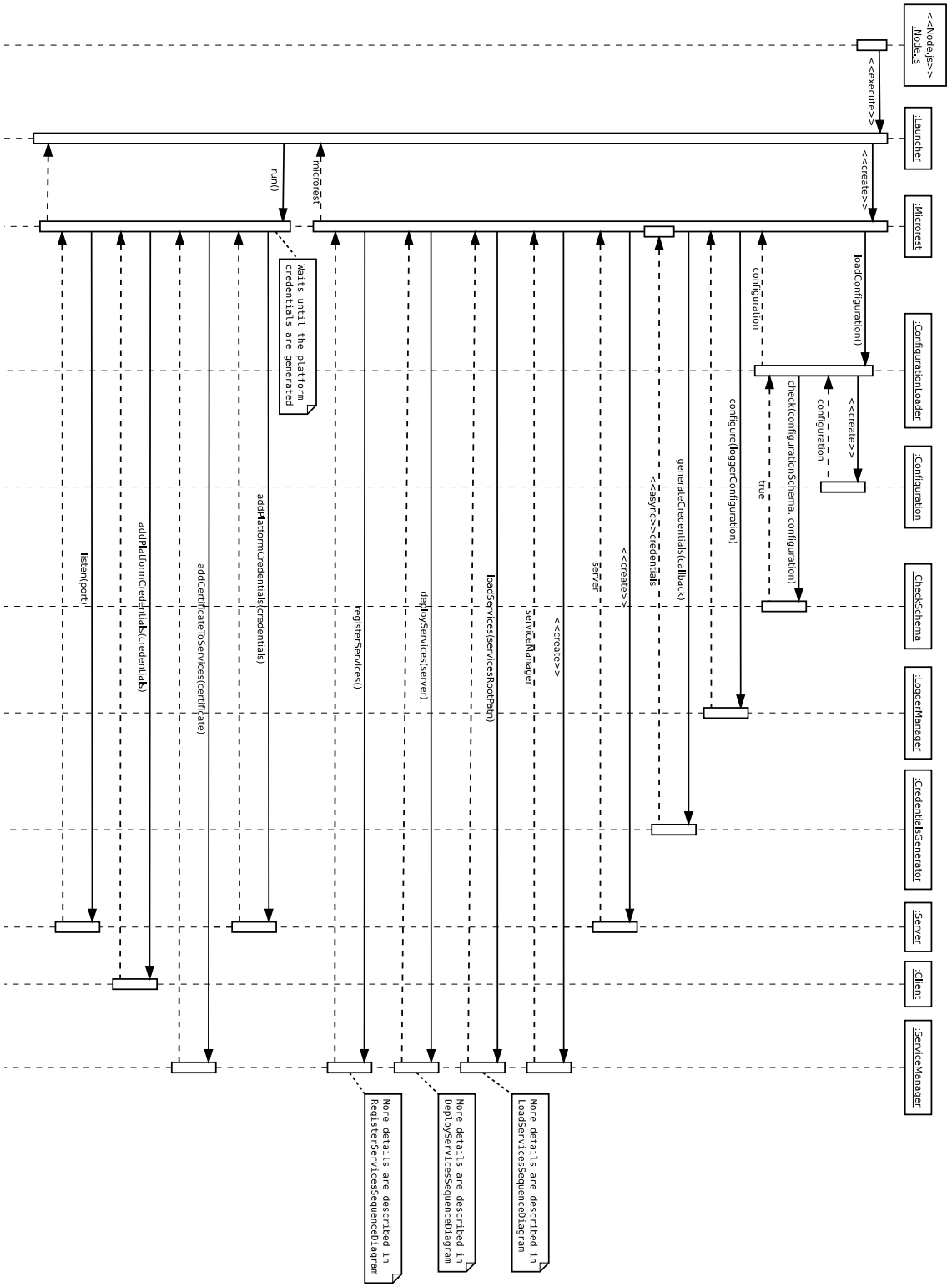


Figure 12. Startup of Microrestjs Framework

Later, during the running phase, the Microrest instance:

1. Waits until the platform credentials (RSA private key and X509 certificate) are generated, if they are not yet. Remember that the platform credentials are generated asynchronously by the CredentialsGenerator.
2. Distributes the platform credentials (RSA private key and X509 certificate) to the Server in order to exclusively accept and establish secure communications over TLS with remote clients.
3. Distributes the X509 certificate to all the deployed microservices using the ServiceManager in order to allow them to know and manage their own public certificate by themselves.
4. Distributes the platform credentials (RSA private key and X509 certificate) to the Client in order to transparently enable secure communication over TLS with other microservices.
5. Enables the Server to listen for incoming HTTP requests from remote clients or microservices.

From this moment, the platform becomes a passive system that only responds to either an incoming HTTP request or a signal from the operating system. Before describing these two scenarios in detail, three parts of the startup process are pending to be explained yet.

Figure 13 illustrates how the ServiceManager loads all the microservices into memory. The ServiceManager receives the root path with all the microservices separated in different subfolders. Then, the ServiceManager reads all the directories of the root path in order to know the microservices that have to be loaded. Afterwards, once the name and the path of the microservices are known, the ServiceManager uses the RunnableServiceFactory to create all the microservices one by one. During this creation phase, the RunnableServiceFactory:

1. Loads the service description into memory using the ServiceContextLoader. During this step, the ServiceContextLoader uses the CheckSchema to verify against the Microrestjs Service Description Specification Schema if the service context respects the specification.
2. Loads the service functionality into memory using the ServiceFunctionalityLoader.
3. Creates a new RunnableService instance with its service context. Then, adds dynamically the functionality of the service to the RunnableService.
4. For each dependency defined in the service context, creates a new CallableService instance using the CallableServiceFactory. Then, registers the CallableService instance in the RunnableService in order to facilitate the remote invocation of such microservice. At this moment, the CallableService has the minimum necessary information for being able to find the location of the remote microservice and obtain its service context.
5. Invokes the “onCreateService” function of the RunnableService in order to initialize its resources. Developers are responsible for overriding this method and initializing the resources. More details about this method are given in 3.3.4. Microservice lifecycle.

Finally, after the creation of each RunnableService, the corresponding instance is stored in memory by the ServiceManager.

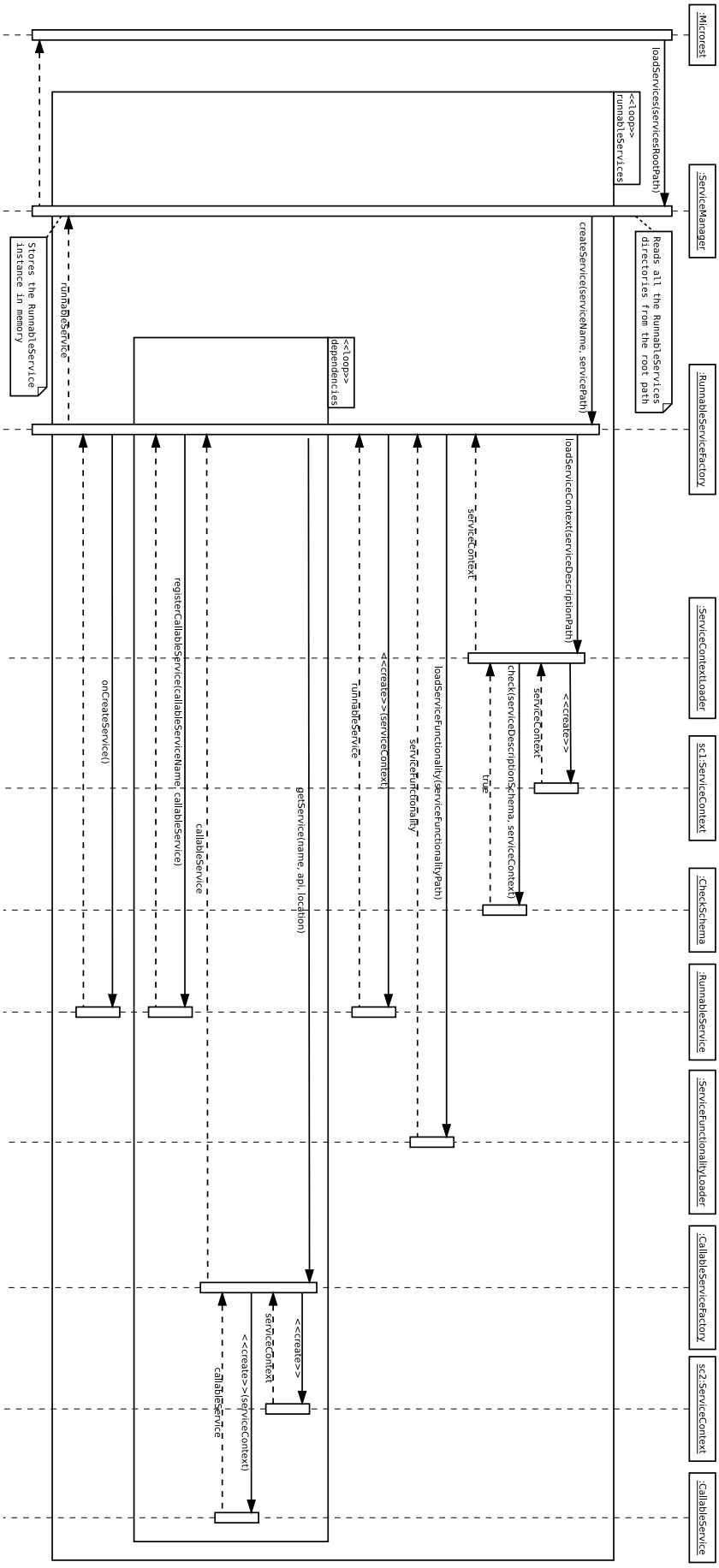


Figure 13. Load of microservices

Figure 14 illustrates how the ServiceManager deploys all the loaded microservices into the Server making them accessible from the network. During this process, a unique URL with the following format is assigned to every deployed microservice:

https://URL-SERVER/MICROSERVICE-NAME/vAPI-NUMBER/

This URL is used for identifying unequivocally every microservice and retrieving its information, i.e. its service context. Also, this URL can be extended with the path of the public operations in order to execute the different operations of the microservices. Remember that all the information about the public operations is defined in the service context that can be retrieved from the base URL.

During the deployment process, the ServiceManager routes the previous loaded RunnableServices one by one using the Server to provide them with a unique URL. During the routing phase, the Server:

1. Creates a new Express Router instance for the current RunnableService.
2. For each public operation defined in the current RunnableService:
    a. Gets a function that check if the received HTTP requests satisfy the conditions imposed by the service description for the execution of that operation.
    b. Gets a function that check if the received HTTP requests are authorized for the execution of that operation.
    c. Gets a function that executes under control the operation of the microservice.
    d. Adds the route of the operation to the Router. Basically, the routes can be seen as tuples composed of the HTTP method (defined in the service context), the path of the operation (defined in the service context), and the three previous functions to check, authorize and execute the operation.
3. For each path that has been routed, gets a function that responds with the error "405 METHOD NOT ALLOWED" if an HTTP request with a valid path but an invalid method is received. And, adds a new route for the rest of the methods that have not being used in such path to respond with "405 METHOD NOT ALLOWED" instead of "404 NOT FOUND".
4. Gets the function that executes the operation "getServiceInformation". This public operation is not defined in the service description because the framework is the responsible of including it automatically during the instantiation of the microservice. The main purpose of this general operation is to provide public information about the microservice (like the service context, the X509 certificate, etc.) to others that can be interested in.
5. Adds the route for the "getServiceInformation" operation to the Router. This operation is always accessible using the path "/" (i.e. the base URL) and the method GET and does not require any special authorization for its execution because it is open for everyone by default.
6. Gets the function to respond with the error "405 METHOD NOT ALLOWED" if an HTTP request with path "/" is received but the HTTP method is not GET. And, adds this new route to the Router.
7. Mounts the Router with all the previous routes at the base URL of the microservice using the Express Application. Thus, the Express Application can delegate all the incoming HTTP requests that contains the base URL to the Router, and the Router to the operation.

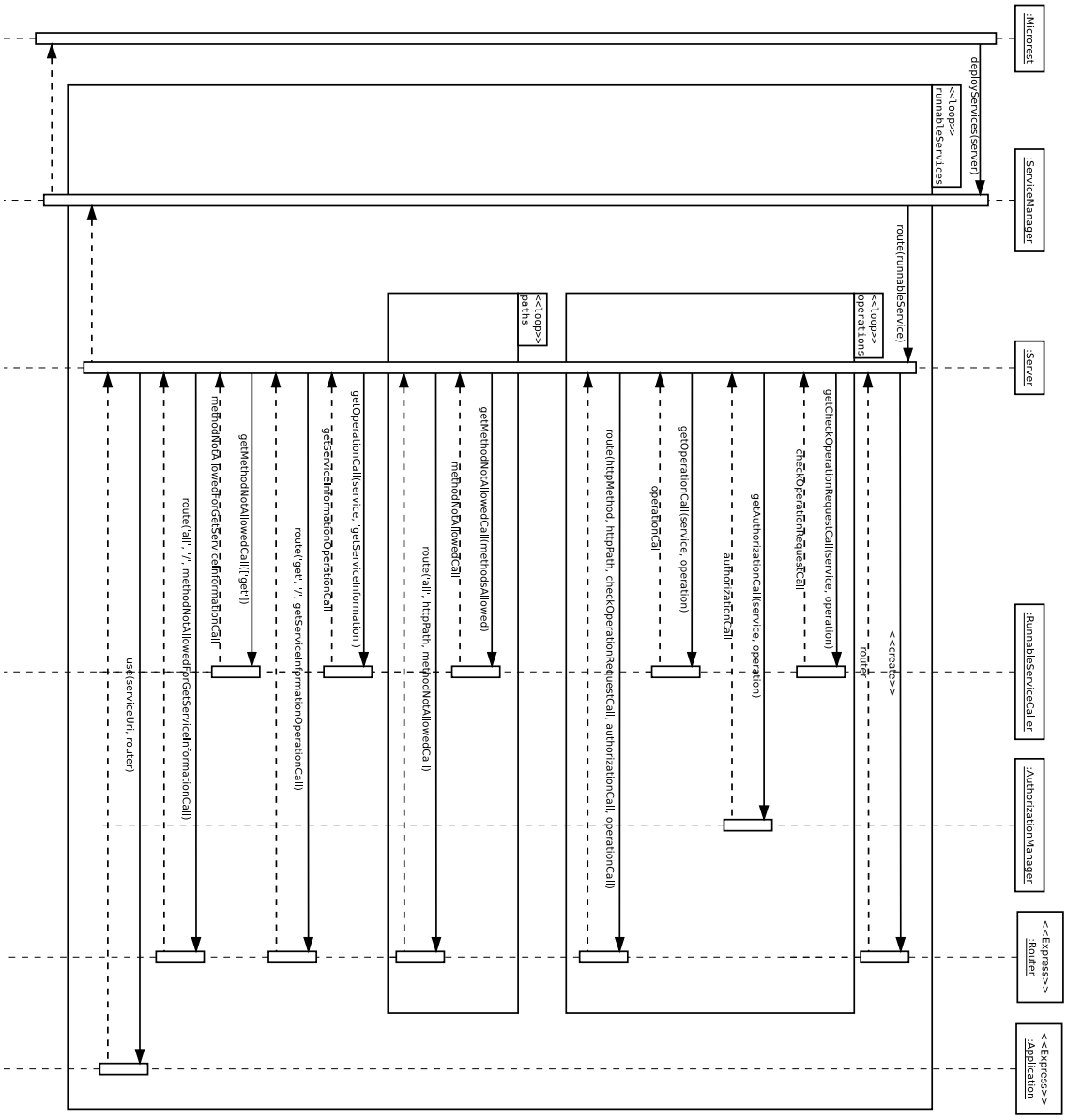


Figure 14. Deployment of microservices

Figure 15 illustrates how the ServiceManager registers all the deployed microservices in the service directory one by one. The ServiceManager uses the ServiceDirectoryProxy to interact with the service directory and register the microservice. For that, the ServiceDirectoryProxy uses the Client to send the registration request to the service directory through the network. The ServiceDirectoyProxy receives and manages the response transparently. Thus, if the registration is not successful, the ServiceDirectoryProxy warns in the log and retries again sending a new registration request. Otherwise, the registration process is considered complete. For more details about the service directory, please consult the section 3.4.1. Service directory.

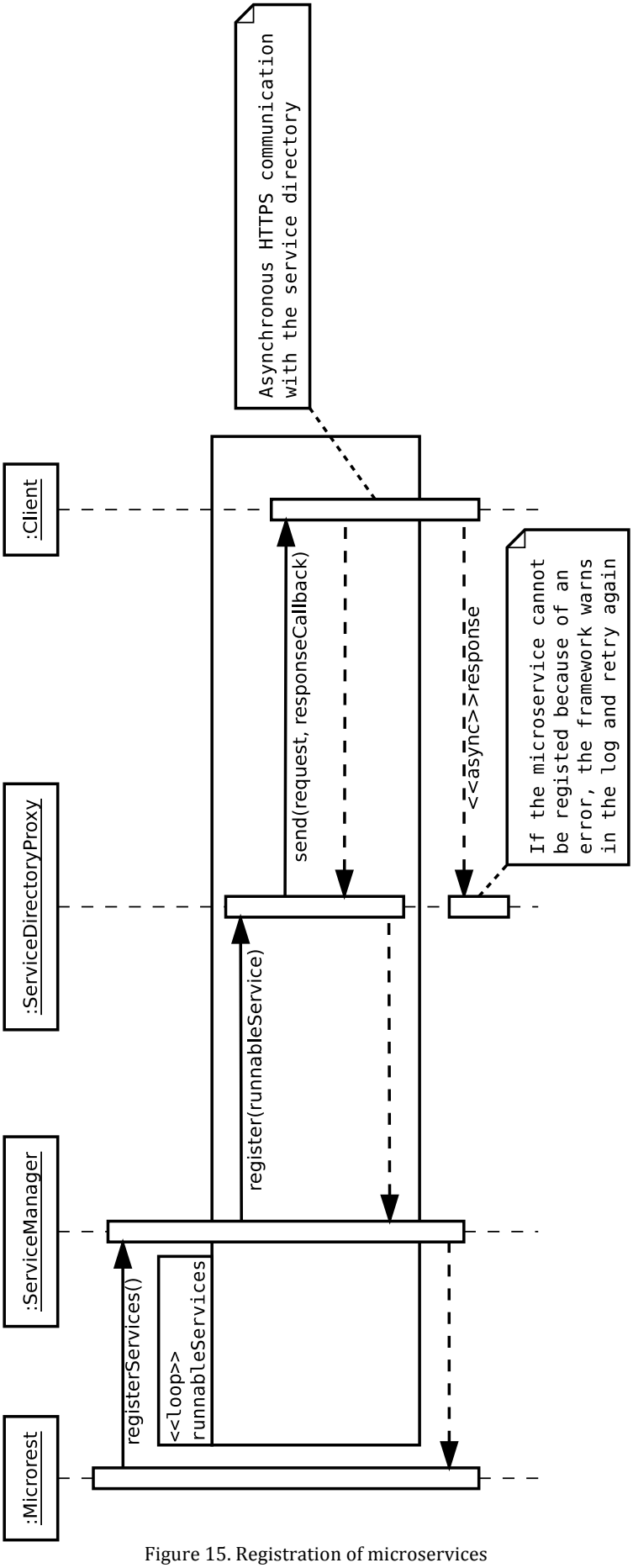


Figure 15. Registration of microservices

Once the startup process finishes, the platform becomes a passive system that only responds to certain events such as some termination signals from the operating system and the incoming HTTP requests from the network.

Regarding the termination signals, the platform reacts to two specific termination signals, SIGTERM and SIGINT. When one of those signals is received, the platform begins the shutdown process for closing and cleaning all the resources gracefully. Figure 16 illustrates how the entire platform is shut down. The Launcher receives the termination signal from the Node.js environment and asks the Microrest for shutting down. Then, the Microrest asks the Server for shutting down, closing and cleaning its resources. After, the Microrest asks the ServiceManager for shutting down, closing and cleaning its resources. Afterwards, the Microrest asks the Client for shutting down, closing and cleaning its resources. Then, the Microrest closes and cleans its own resources. Finally, the shutdown process ends successfully with the exit of the platform.

During the shutdown process, one of the most important aspects is the destruction of the platform credentials that are in memory. Thus, all the classes that use somehow the platform credentials are responsible for destroying the credentials that are using. More specifically, the Server, the Client and the ServiceManager must destroy the platform credentials during the shutdown process.

On the other hand, the ServiceManager is also responsible for managing the shutdown of the deployed microservices. In particular, the ServiceManager invokes the “onDestroyService” function of all the RunnableServices in order to allow them to close and clean all their resources. Developers have the responsibility for overriding this method with the appropriate functionality to shut down their microservices. More details about this method are given in 3.3.4. Microservice lifecycle.

As can be observed during the shutdown process, the platform does not notify the service directory that their microservices will not be available anymore. As a consequence, the service directory does not unregister those microservices immediately. They will be unregistered by the service directory itself later, during a periodical checking. Although this aspect can seem a drawback, it has been designed in this way with the purpose of avoiding that malicious microservices could unregister real microservices. More details about how the microservices are unregistered are presented in 3.4.1. Service directory.

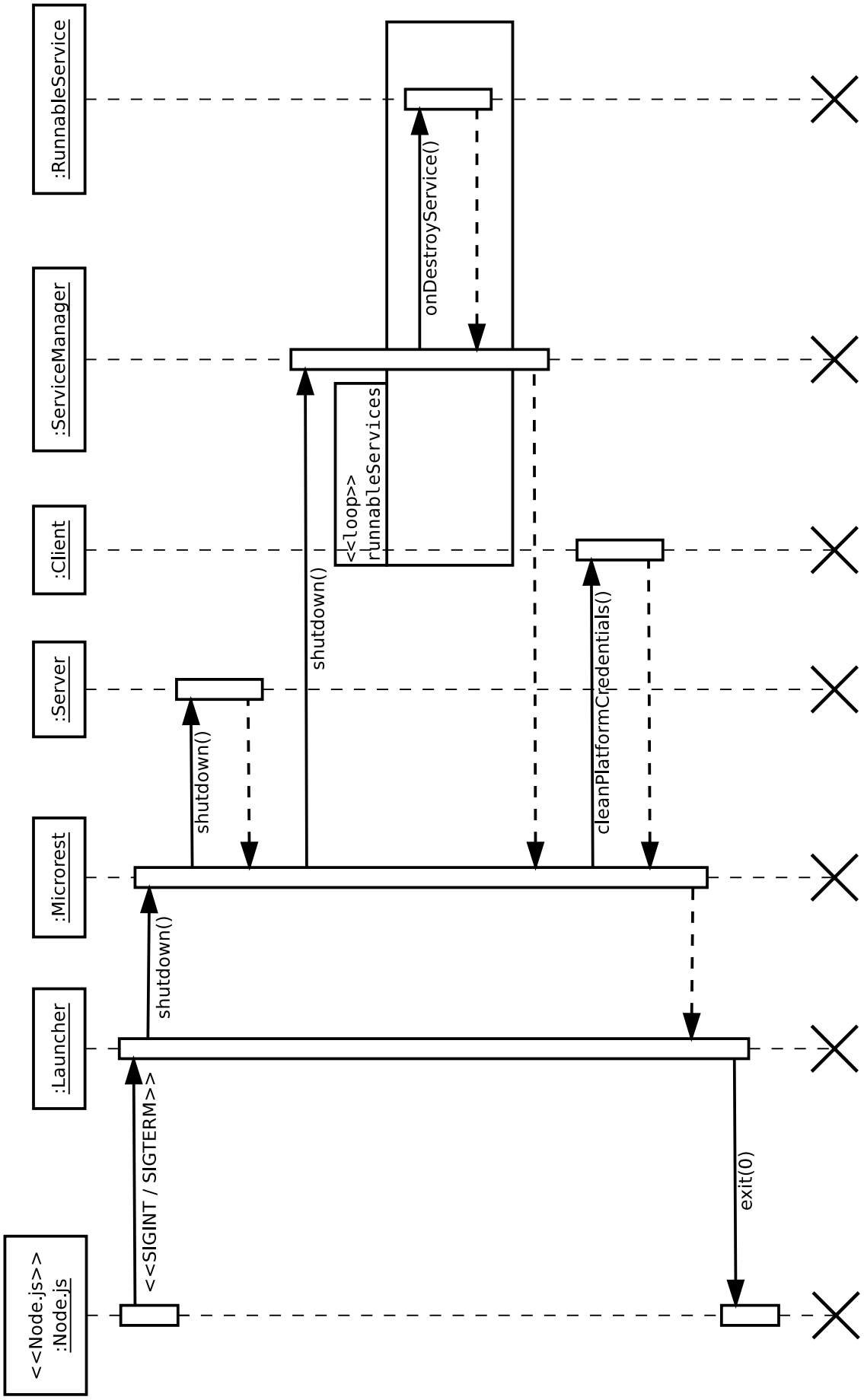


Figure 16. Shutdown of Microrestjs Framework

Regarding the incoming HTTP requests, the platform is able to receive and handle HTTP requests from remote clients or microservices through the HTTPS Server. Figure 17 illustrates how the platform handles all the incoming HTTP requests. When a new HTTP request is received, the HTTPS Server delegates that request to the Express Application in order to redirect the request to the appropriate microservice. During the deploying phase, each route was defined together with three functions to be executed. Thus, the Express Application executes the first function (checkOperationRequestCall) to check transparently if the request satisfies the conditions imposed by the service description for the execution of the operation. If the result of this first function is satisfactory, the Express Application executes the next function (authorizationCall) to authorize the execution of the operation. If the result of this second function is satisfactory, the Express Application finally executes the third function (operationCall). This third function is the responsible for executing under control the operation of the microservice. For that, the "onStartOperation" function is invoked to inform the microservice which operation will be executed. Then, the received HTTP request is converted to a RunnableServiceRequest and an empty RunnableServiceResponse is created with the purpose of controlling exhaustively what information can be accessed or modified by the microservices in order to increase the security of the platform. Afterwards, the operation of the microservice is called, executing the logic of the microservice. Once the microservice has filled the RunnableServiceResponse with the information for the client, the RunnableService can invoke the "sendResponse" function to send the corresponding HTTP response to the client that is waiting for. Finally, the "onFinishOperation" function is invoked to inform the microservice if the HTTP response has been sent successfully.

As mentioned previously, the Inversion of Control principle is a key aspect of the platform that allows controlling the execution of the microservices and guaranteeing certain properties such as security and privacy, among others. The application of this important principle can be seen explicitly in Figure 17 when the operation of the microservice is called from the framework.

With respect to the possible problems that could occur during the handling of any HTTP request, the platform is prepared for handling six different exceptional behaviors that are illustrated by Figure 18, Figure 19, Figure 20, Figure 21, Figure 22, and Figure 23. First, if the HTTP Server receives a request for a microservice that has not been deployed, the framework responds with error 503 SERVICE UNAVAILABLE. Second, if the HTTP Server receives a request for a path that has not been routed, the framework responds with error 404 NOT FOUND. Third, if the HTTP Server receives a request for a routed path but using an incorrect HTTP method, the framework responds with error 405 METHOD NOT ALLOWED. Fourth, if the HTTP Server receives a correct request that does not satisfy the service description, the framework responds with error 400 BAD REQUEST. Fifth, if the HTTP Server receives a correct request that does not have correct credentials or authorization for executing the operation, the framework responds with error 401 UNAUTHORIZED or 403 FORBIDDEN, respectively. Finally, if the framework encounters an unexpected condition that prevents from continue the execution, the framework responds with error 500 INTERNAL SERVER ERROR.

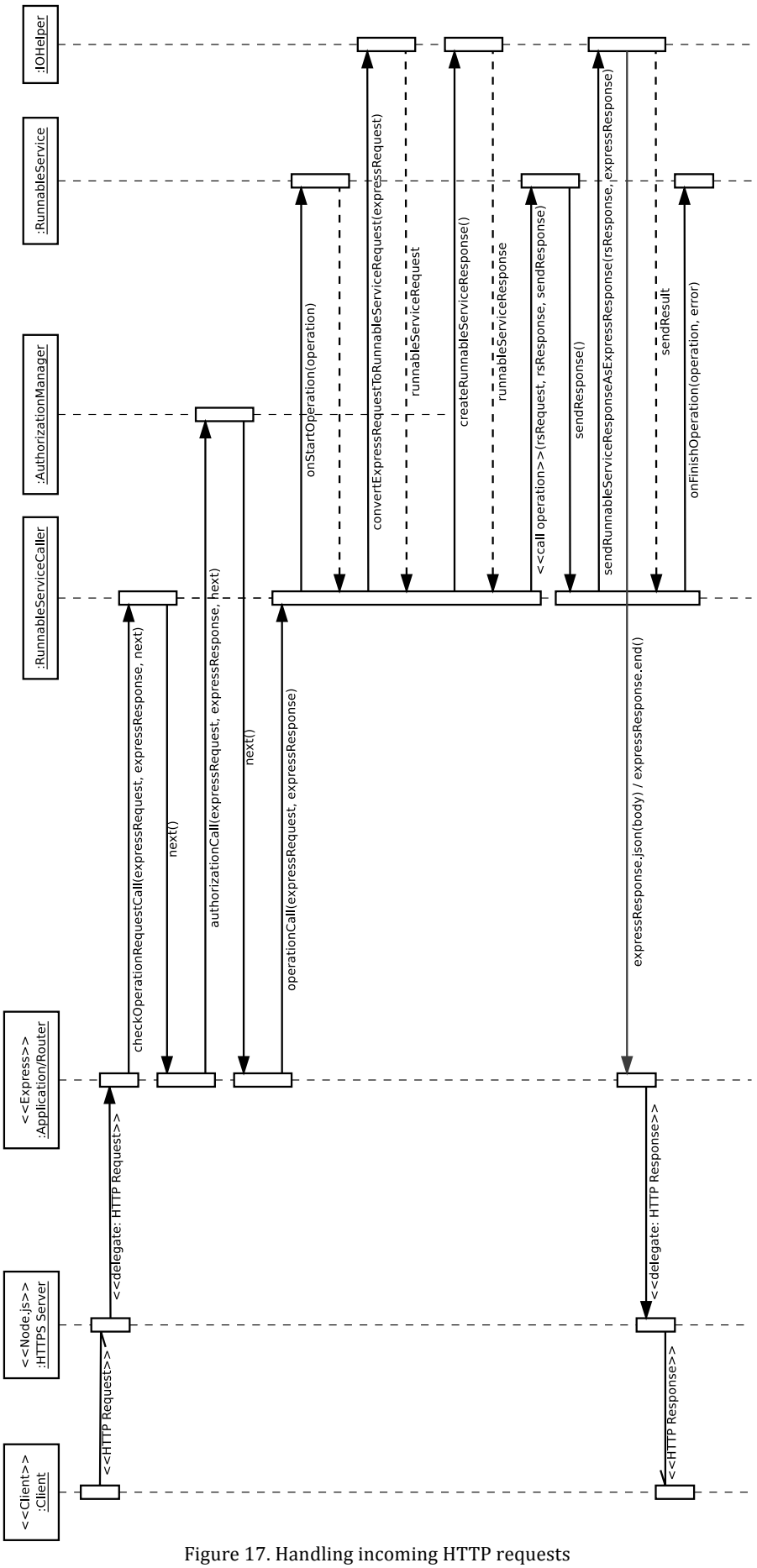


Figure 17. Handling incoming HTTP requests

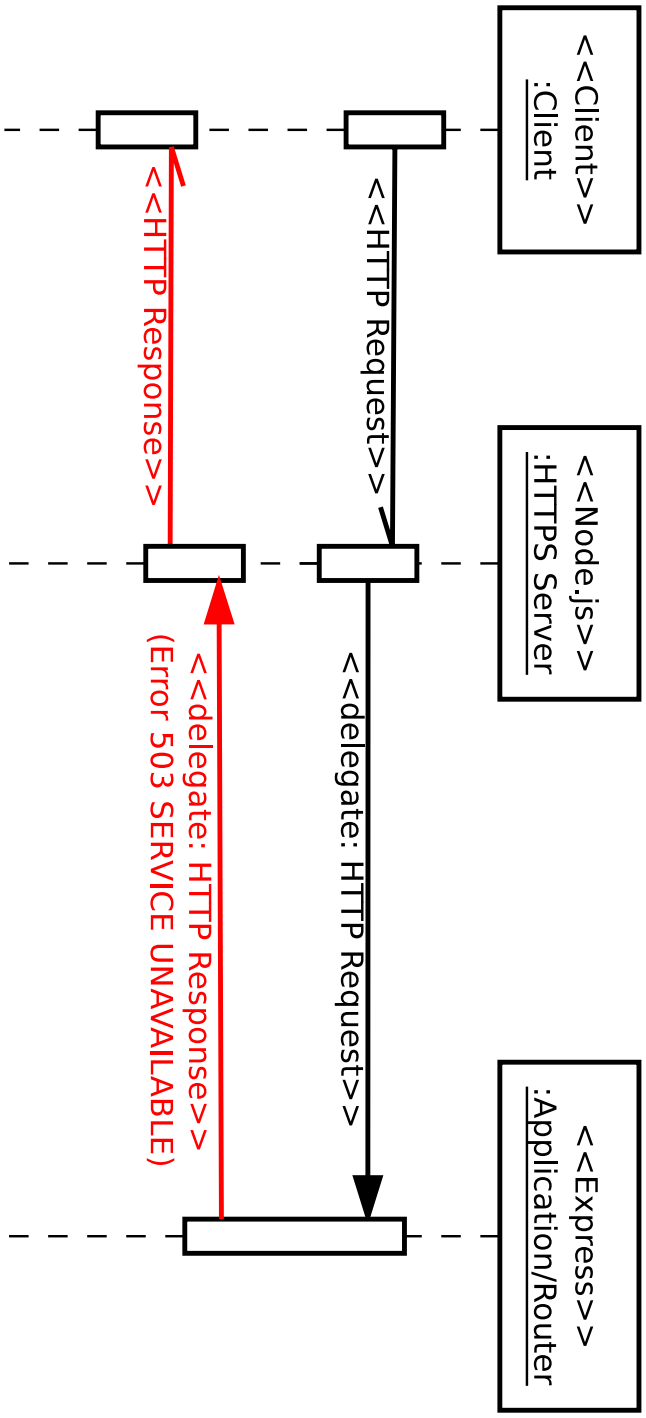


Figure 18. Error 503 SERVICE UNAVAILABLE

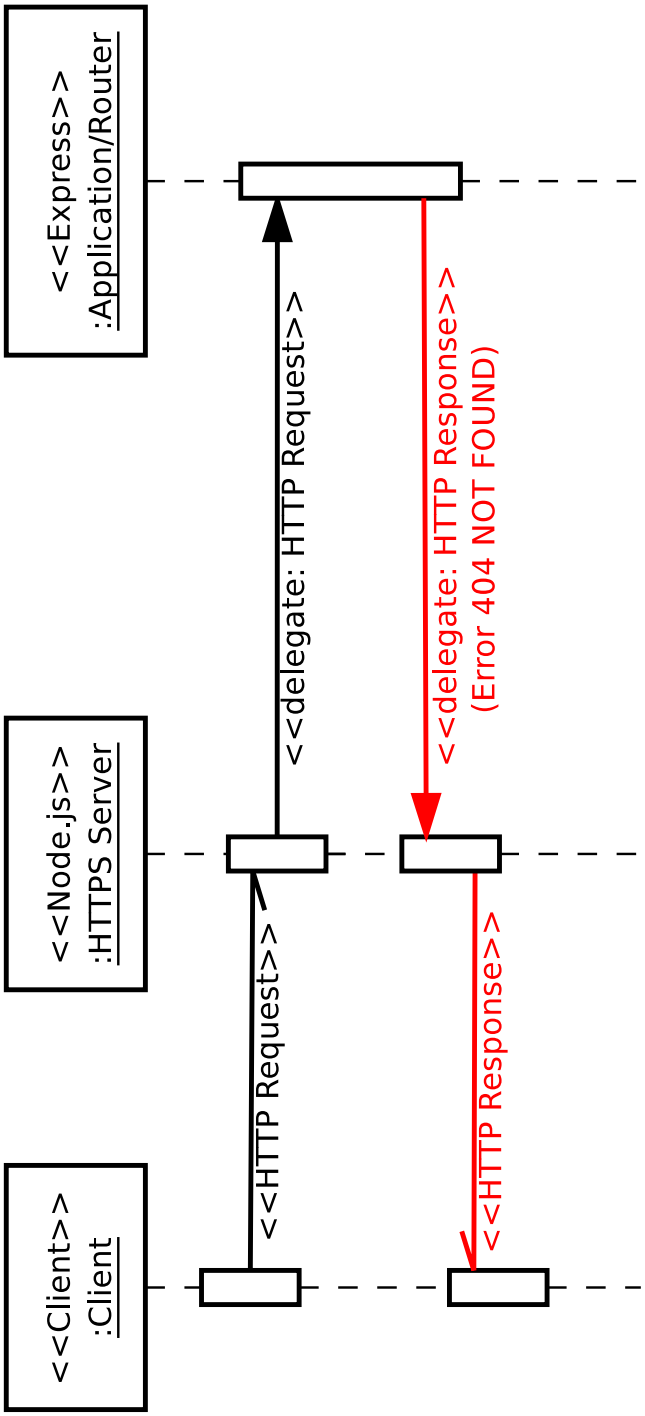


Figure 19. Error 404 NOT FOUND

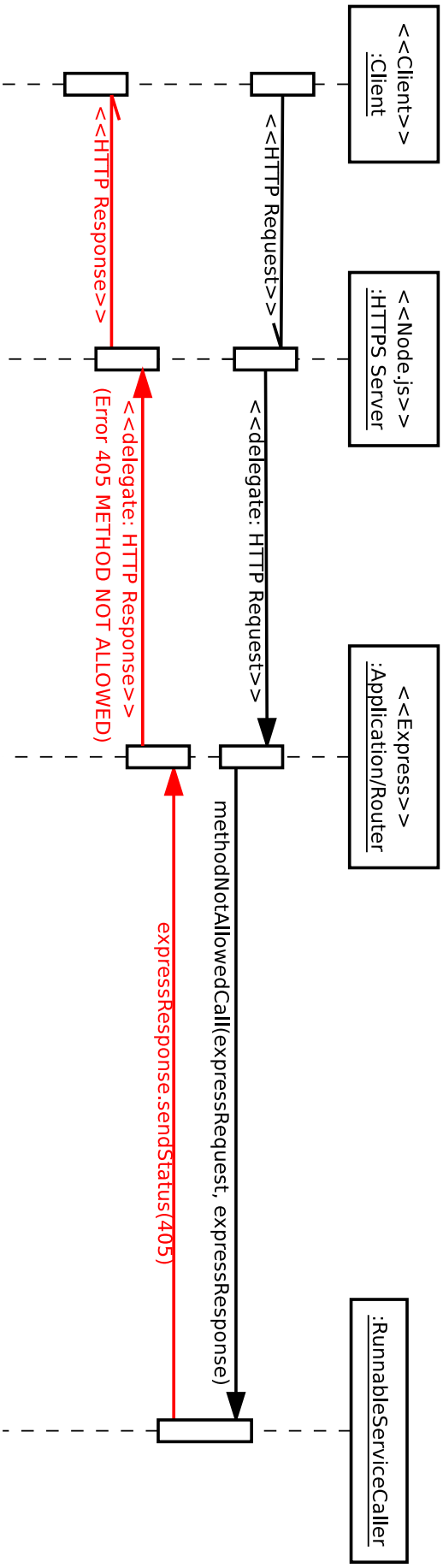


Figure 20. Error 405 METHOD NOT ALLOWED

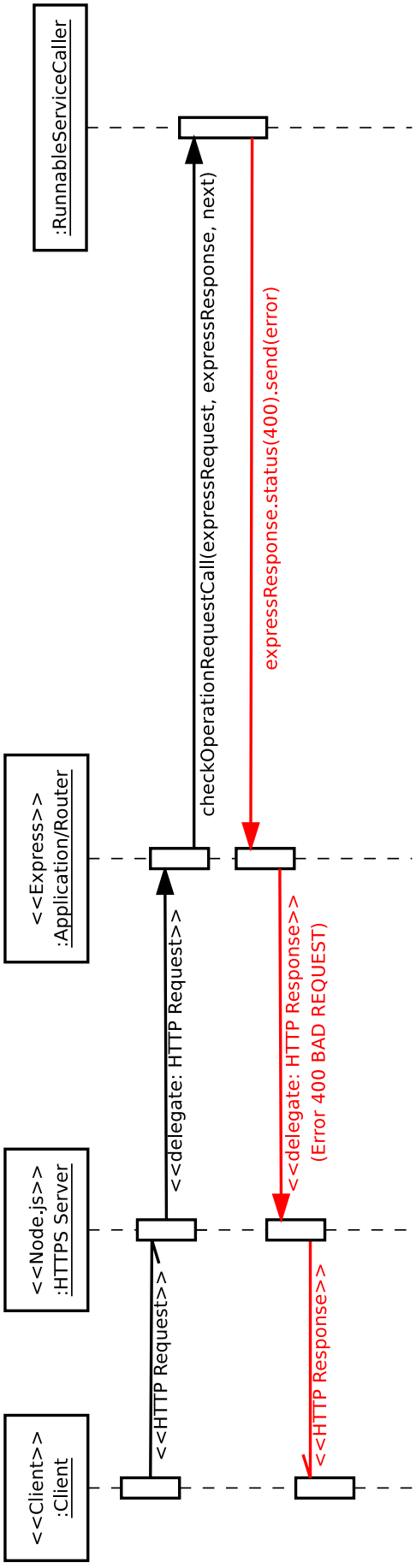


Figure 21. Error 400 BAD REQUEST

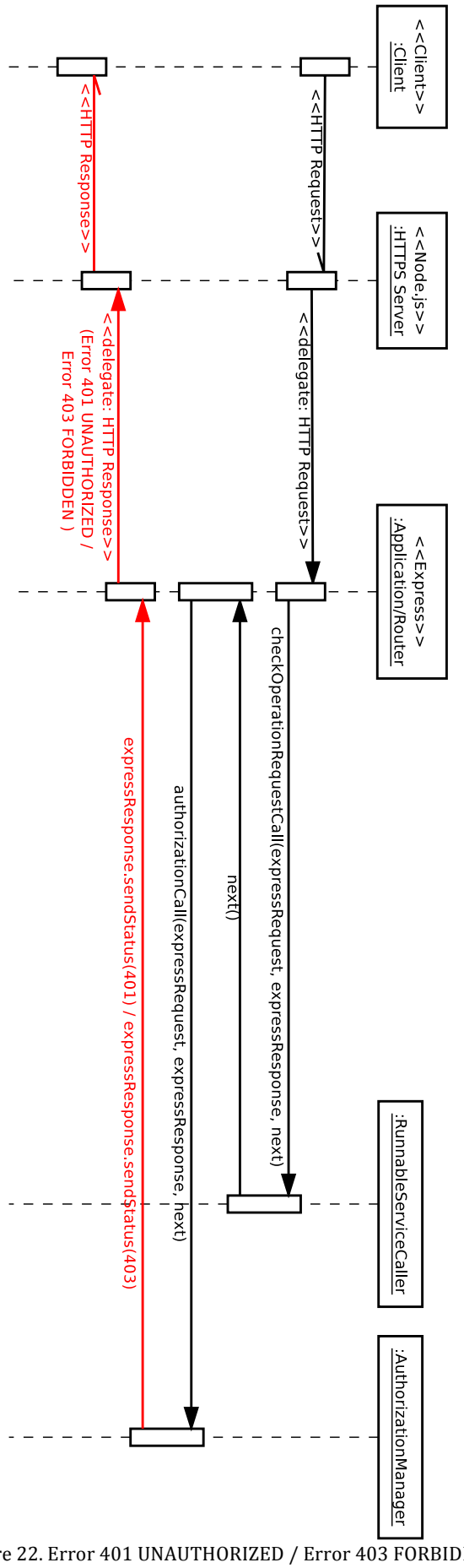


Figure 22. Error 401 UNAUTHORIZED / Error 403 FORBIDDEN

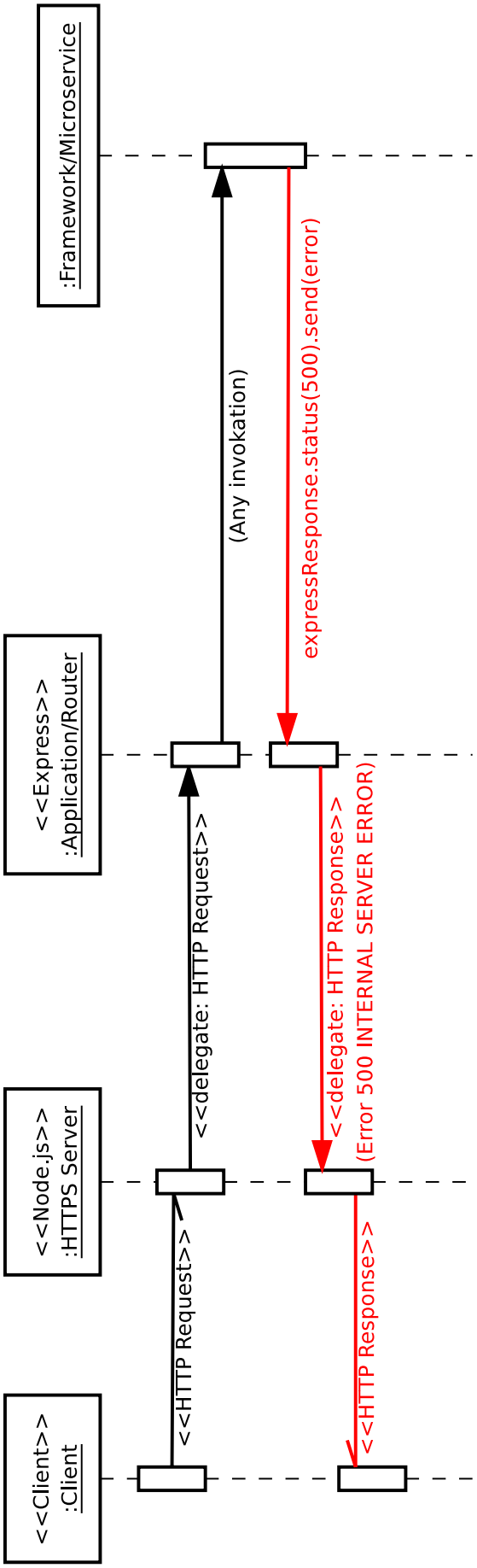


Figure 23. Error 500 INTERNAL SERVER ERROR

The previous diagrams describe how the microservices' operations are executed when any HTTP request is received; however, they do not specify how other remote microservices can be executed from these operations using the CallableServices defined during the startup process.

Figure 24 illustrates the process for executing a remote microservice from a RunnableService. First of all, it is necessary to remember that the CallableServices are created from the dependencies defined in the service description during the startup process with the purpose of providing stubs with a uniform interface for interacting with the real microservices remotely. During the startup process, these CallableServices are also registered within the RunnableService in order to be able to use them at runtime. Essentially, the diagram shows how the registered CallableServices can be executed to interact with the corresponding remote microservices. The process begins when the RunnableService invokes the "execute" method of any registered CallableService. At this moment, the CallableService starts to run the following steps:

1. The CallableService checks if the real location of the remote microservice is known. In such case, step 2 is skipped.
2. If the CallableService does not know the real location of the remote microservice, it uses the ServiceDirectoryProxy for looking up the real location of the remote microservice in the service directory. Once the real location is retrieved, the CallableService caches it and runs the "executed" method again from the beginning.
3. The CallableService checks if the service context of the remote microservice is known. In such case, step 4 is skipped.
4. If the CallableService does not know the service context of the remote microservice, it uses the ServiceInformationRetriever for retrieving its service context. Once the service context is retrieved, the CallableService caches it and runs the "executed" method again from the beginning.
5. At this point, the CallableService has all the necessary information to verify locally if the request parameter satisfies the conditions for executing any operation of the remote microservice. After the local verification, the CallableService uses the Client for sending the corresponding HTTP request to the remote microservice.
6. Later, when the HTTP response is received from the remote microservice, the HTTP response is converted to a CallableServiceResponse using the IOHelper. Then, the Client passes the CallableServiceResponse to the CallableService, and the CallableService to the RunnableService that is waiting for the result of the execution of the remote operation.

Once the RunnableService gets the response, it can continue the execution of its own operation with the new received information. At the end, the RunnableService has to respond to the Client, as was illustrated in Figure 17.

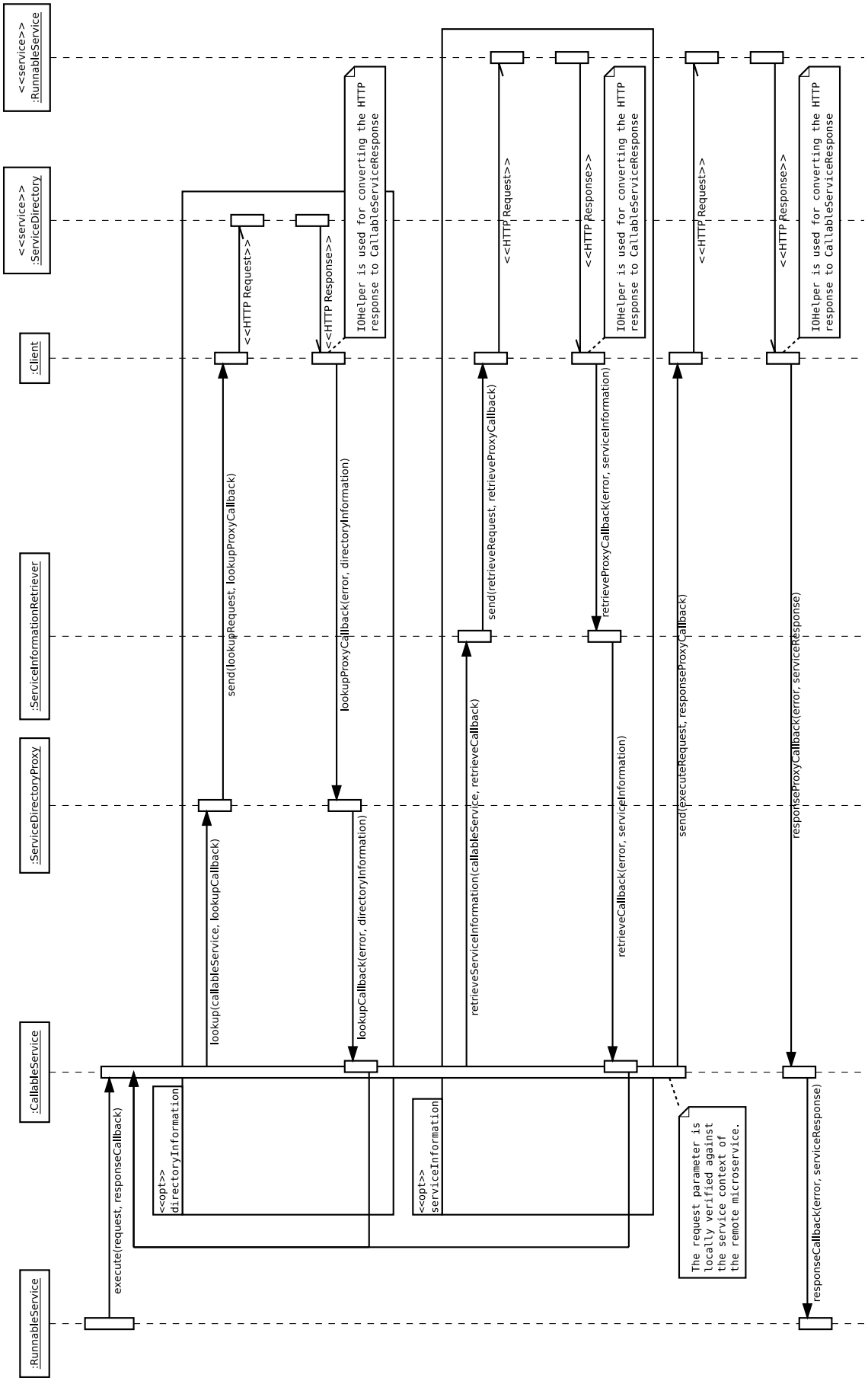


Figure 24. Execution of remote microservices

As can be observed in the above diagram, the CallableServices are stubs with a uniform interface that facilitates the interaction with the real remote microservices. In particular, the CallableServices provide an interface with only one method, called "execute", which requires two parameters: request and responseCallback. The first parameter specifies the request that is going to be executed by the remote microservice. The second parameter specifies the function that has to be invoked after the execution for notifying the result of the operation. In order to guarantee a high uniformity and eliminate any kind of uncertainty, the framework describes the request parameter formally:

```
request = {
     operation = <string>,
     credentials = {
          username = <string>,
          password = <string>
     },
     parameters = {
          <name: string> = <value: string, integer,
                                   number, boolean>
     },
     body = <object>
}
```

Field Name	Type	Description
operation	string	**Required.** Specifies the operation to be executed remotely.
credentials	object	Specifies the credentials for the authentication and authorization of the request, if it is required by the operation. The credentials are composed of a username and a password, both strings.
parameters	object	Specifies the set of parameters required for the execution of the operation. Each parameter is composed of a name and a value. The name is a string and the value can be a string, an integer, a number or a boolean.
body	object	Specifies the information to be sent for the execution of the operation.

With respect to the responseCallback parameter, it requires a function with two parameters: error and response. On the one hand, the error parameter will be used for notifying any error that could occur during the execution of the remote operation. On the other hand, the response parameter will be used for notifying the result of the operation using a CallableServiceResponse object.

In conclusion, the framework provides a real uniform interface for interacting with remote microservices using the CallableServices.

Returning to Figure 24, the diagram shows two optional parts, which are not always required for the execution of remote microservices. Indeed, these optional parts are executed the first time that every CallableService is executed because they were created with the minimum necessary information to be able to find the real microservices. Thus, the CallableServices cannot interact directly with the microservices. First, they have to find the real location of the microservices (first optional part). And then, they have to retrieve their service information (second optional part). Once the two optional parts have been executed, the CallableServices can cache the retrieved information in order to skip the optional parts next time. When the optional parts are skipped, the CallableServices undergo a considerable increase in performance. This increase in performance is due to the fact that network communications are much more expensive than local computations. Therefore, if the network communications are reduced to the minimum, the overall performance increases significantly. In our case, the two optional parts use network communications for retrieving information, but they can be omitted because the information retrieved by them remains "immutable" over the time and, therefore, it can be cached.

The main drawback of having to cache information is to detect when that information is not valid anymore in order to retrieve the new one. In our case, the CallableServices implement an easy tactic to clean the invalid information and force the execution of the optional parts again. Basically, the CallableServices consider that the information is valid as long as they can interact with the corresponding remote microservice without problems. Thus, if a CallableService has interaction problems because of an error, the CallableService cleans the cached information and forces the execution of the optional parts next time. Then, once the optional parts are executed again, the CallableService is able to interact again with the remote microservice. It is important to remark that the new remote microservice could be different from the previous one because the new information returned by the service directory could point to another registered instance that could be even in another physical location. With respect to the possible errors that could occur, the most common ones are caused for the shutdown or the restart of the current remote microservice. In case of shutdown, the microservice is not available anymore. In case of restart, the microservice is deployed as a new instance in the same location, but with some different pieces of information, as for example, a different X509 certificate. In addition, the interaction could fail because of a temporary failure in the network. In this last case, the microservice will be eventually accessible through the network in the future; thus, the service directory could return the same instance again.

Apart from the increase in performance, the current design with the optional parts provides also high flexibility, which facilitates the evolution of the topology of the microservices. Basically, the framework is able to adapt to the new changes in the topology transparently. Summarizing, the framework is prepared for being used in highly changeable environments.

Now, the reader may be wondering about how the CallableServices are able to interact with the remote microservices during the second optional part without having their service information. This is possible because all the RunnableServices

have an operation called “getServiceInformation” by default that is deployed on the base URL. Thus, the ServiceInformationRetriever knows how to execute this operation having exclusively the location of the microservice. As was mentioned before, the purpose of this operation is to provide the service information in order to be able to execute the rest of public operations of the remote microservice.

The last important aspect regarding the diagram is the local verification that is done before sending the HTTP request to the remote microservice. Initially, this local verification at the client side can seem unnecessary because the remote microservices (server side) check all the incoming HTTP requests meticulously for security reasons. Thus, the main obvious drawback of doing local verifications is the negative impact that they have in performance. On the contrary, the main advantage is the increase in the number of operations that are executed successfully. Moreover, it is important to remember that an error at the server side is much more expensive than at the client side because of the cost of the network communications. Therefore, the use of local verifications may be justified if the decrease of performance caused by the local verifications is less than the decrease of performance caused by useless network communications that could be detected by the local verifications. At the end, the design decision of including local verifications was taken considering the high impact that useless network communication could have in systems with a large number of microservices and interactions between them.

Although the reasons of some errors were introduced previously, the following diagrams show visually the possible errors that could occur during the execution of a remote microservice. Figure 25 illustrates the errors that can occur during the interaction with the service directory. In particular, the CallableService could have problems for establishing a network communication with the service directory; in this case, the Client is who raises the error. On the other hand, the network communication can work properly, but the service directory cannot find any microservice; in this case, the ServiceDirectoryProxy is who raises the error because the network communication was successful and the result is part of the normal operation of the service directory. In either case, the execution of the remote microservice is not possible, and the error is passed to the RunnableService through the responseCallback. Similar to the previous one, Figure 26 illustrates the errors that can occur during the interaction with the microservices for retrieving their service information. In this case, the errors can be because of either the network connection or an operational error during the execution of the “getServiceInformation” operation. In either case, the cached information is cleaned (the location of the microservice), and the error is passed to the RunnableService through the responseCallback. Figure 27 illustrates the errors that occur during a local verification. In this case, these errors are passed to the RunnableService through the responseCallback, but they do not imply the cleaning of the cache information. Finally, Figure 28 illustrates the errors that can occur during the interaction with the microservice for executing the operation. In this case, the only possible error is a failure in the network communication. As a consequence, the cached information is cleaned (the location of the microservice + the service information), and the error is passed to the RunnableService through the responseCallback.

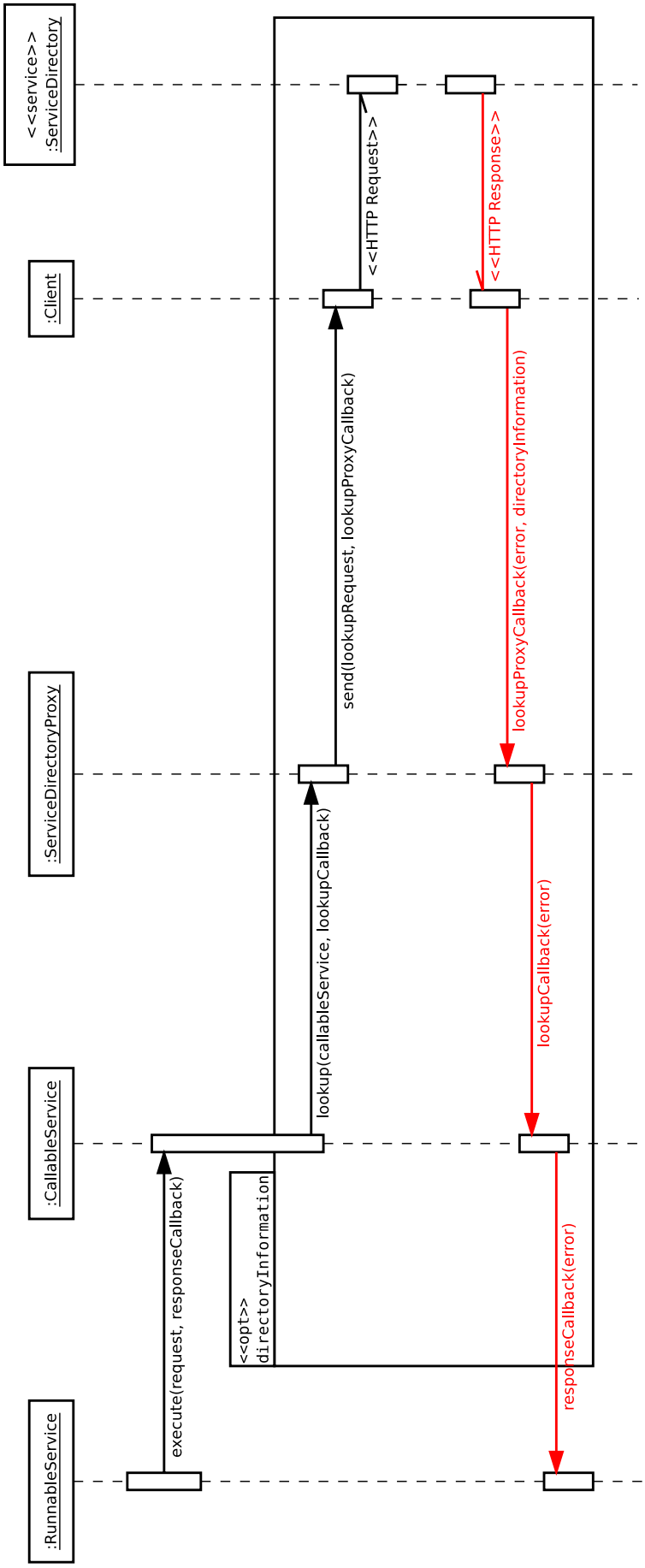


Figure 25. Service directory error

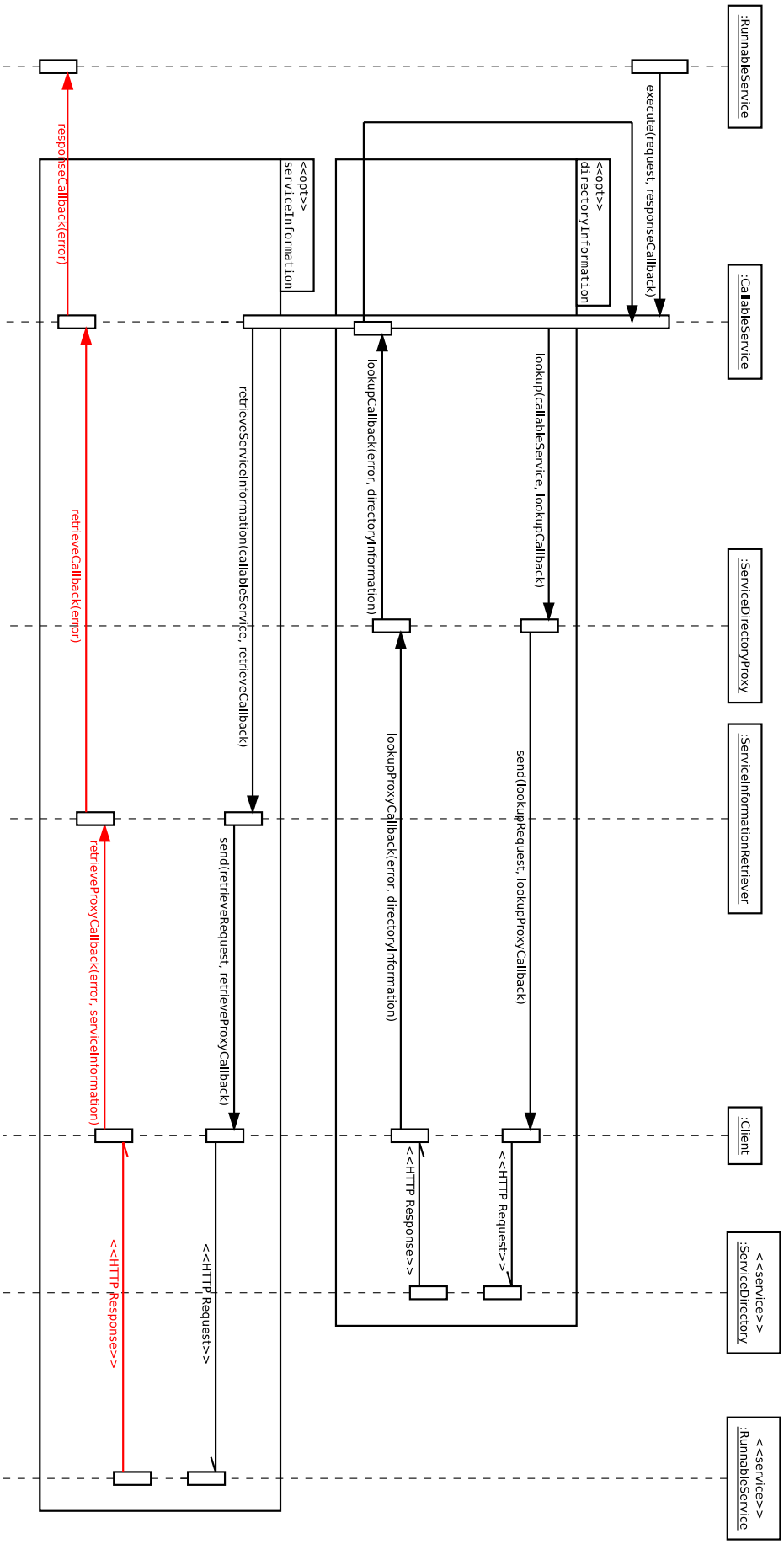


Figure 26. Service information error

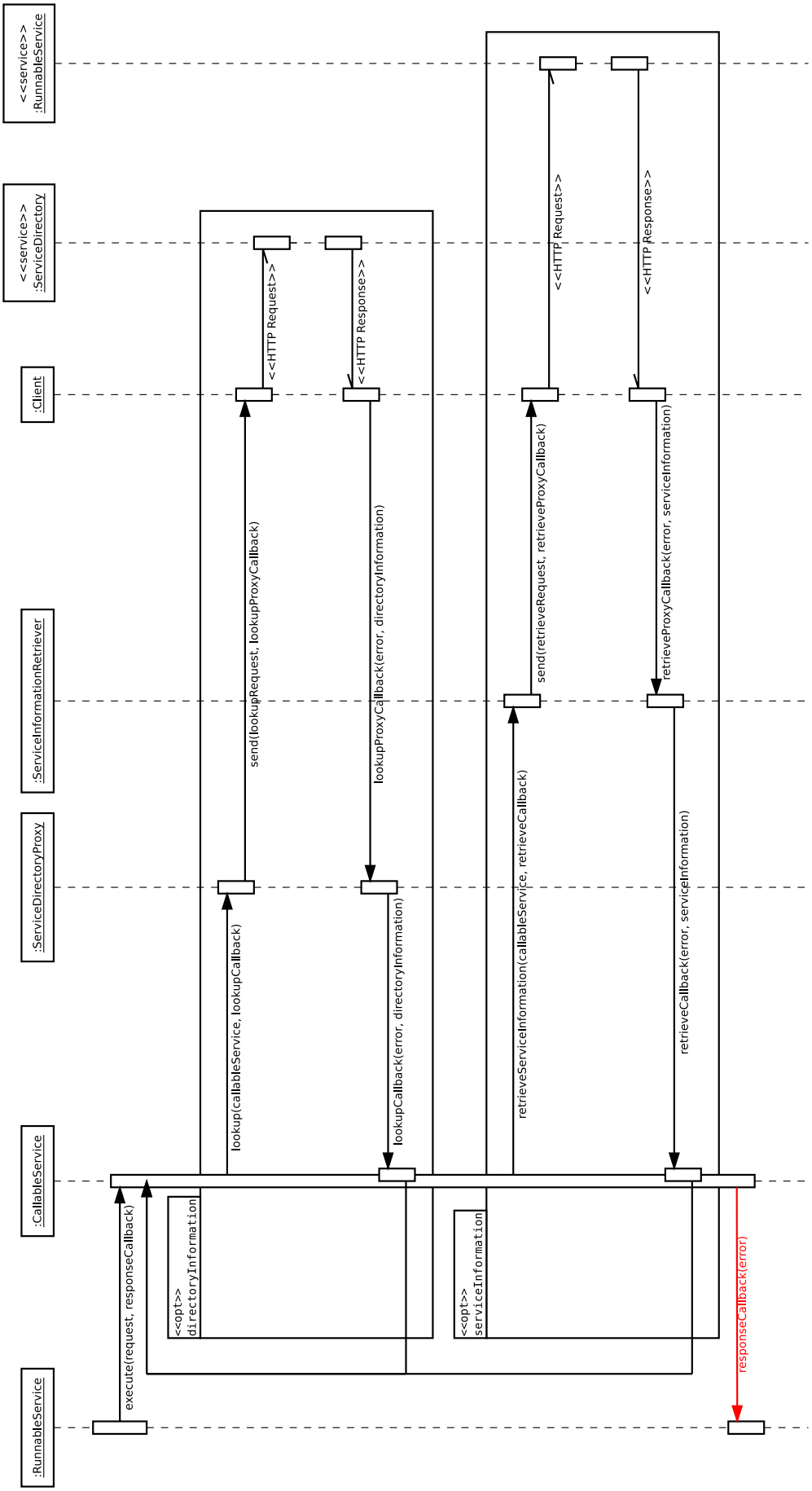


Figure 27. Request verification error

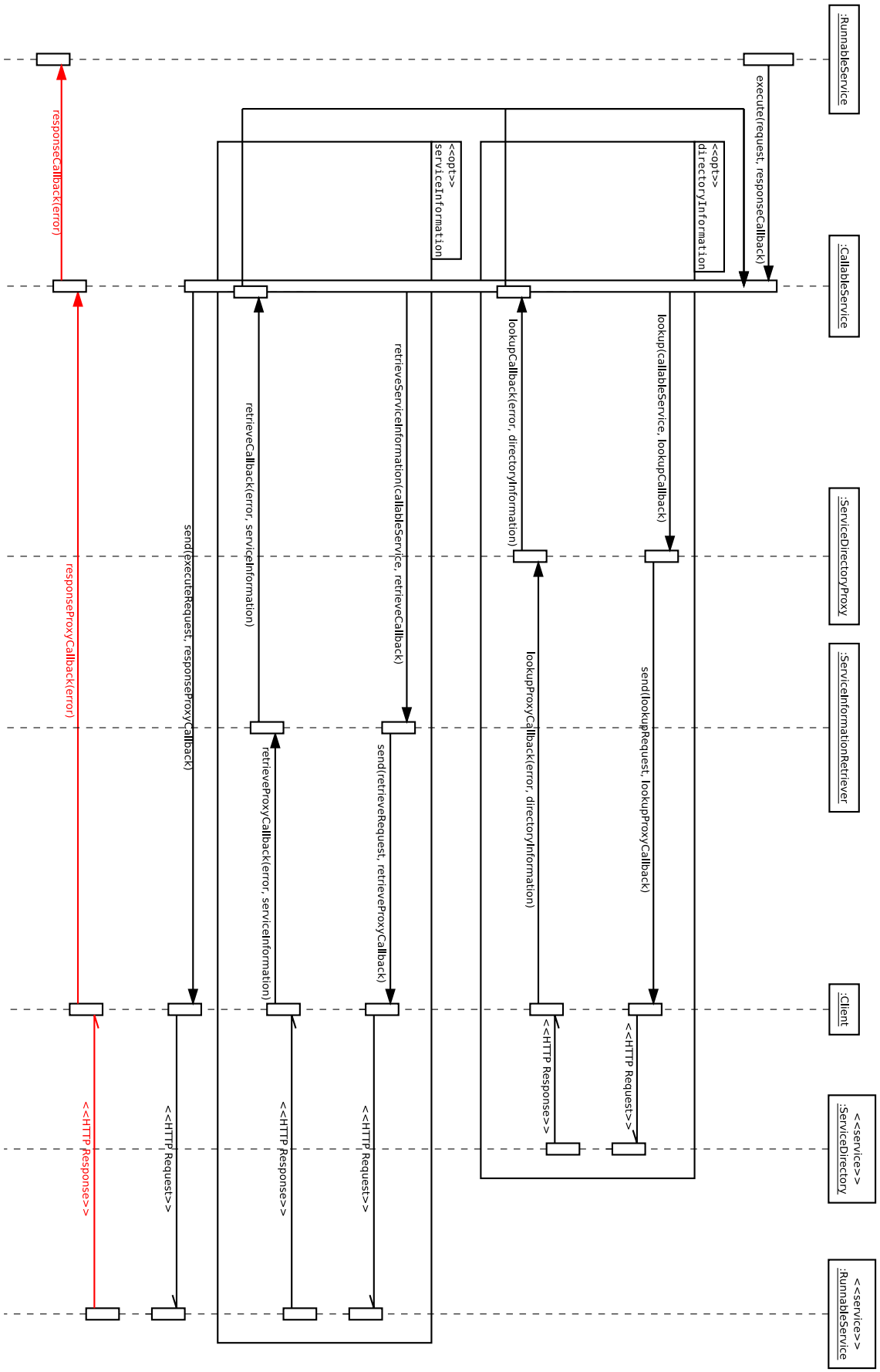


Figure 28. Remote execution error

Summarizing, the reader should by now have deep knowledge about the structure and operation of the platform. Specifically, the reader should know which are the reasons behind the proposed design, how the platform is started up, how the microservices are loaded into memory, how the microservices are deployed into the Server, how the microservices are registered in the service directory, how the platform is shut down gracefully, how the microservices operations are executed when any HTTP request is received, and how the microservices can execute other remote microservices.

3.3.4. Microservice lifecycle

The previous subsection described the design and operation of the Microrestjs Framework from a general perspective without focusing on the microservices. The objective of this subsection is to complete the picture of the Microrestjs Framework, focusing on the microservices exclusively.

In short, the Microrestjs Framework is in charge of creating, executing and destroying the microservices, but the reality is that the microservices have their own lifecycle, which is entirely controlled by the framework. Figure 29 shows this lifecycle, which is composed of seven states and four functions. On the one hand, the states represent the phases in which a microservice could be at any moment. On the other hand, the functions allow microservices to execute custom functionality before moving to another state.

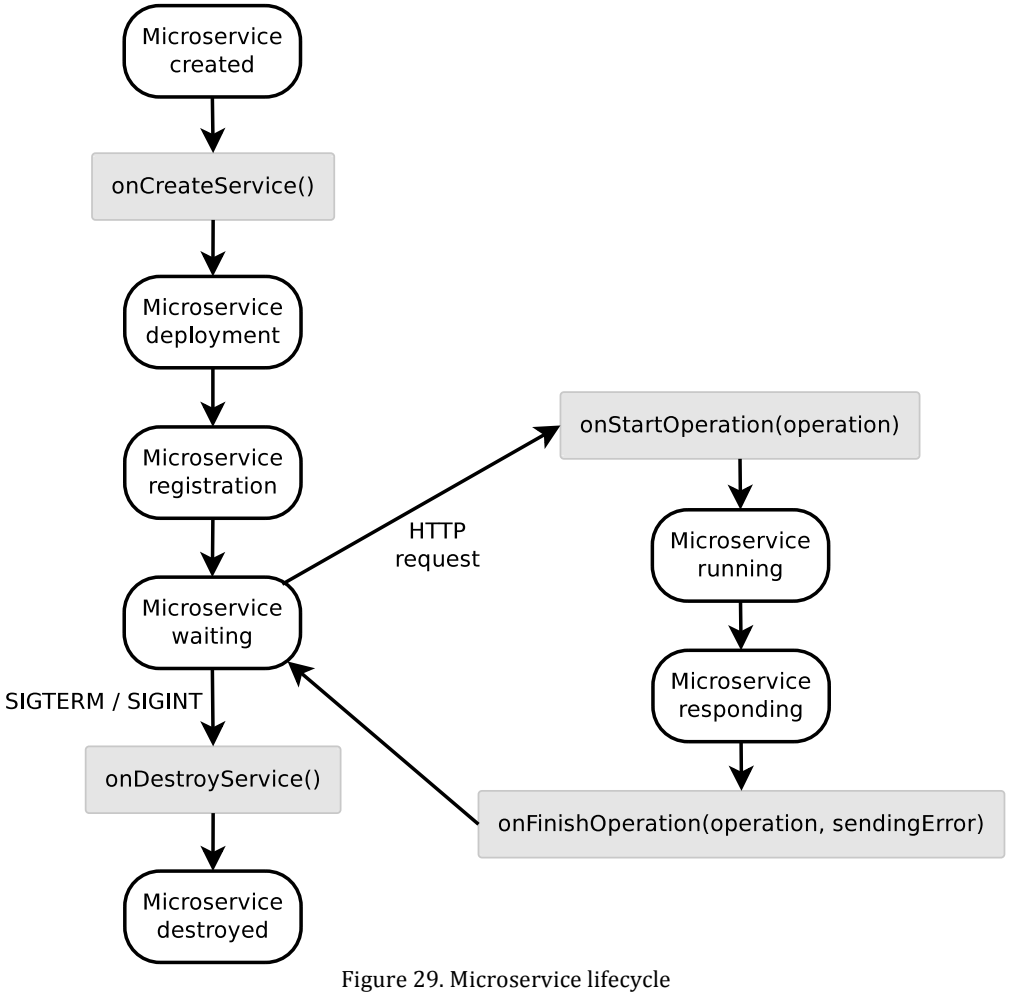


Figure 29. Microservice lifecycle

The lifecycle of a microservice begins with state “Microservice created” because the first moment of its existence is just after being instantiated by the framework. Then, the Microrestjs Framework invokes the function “onCreateService” in order to allow the microservice to initialize its own resources. After invoking this function, the microservice moves its state to “Microservice deployment” because it will be deployed into the Server by the framework. Afterwards, the microservice moves its state to “Microservice registration” because it will be registered in the service directory by the framework. Later, the microservice moves its state to “Microservice waiting” because it starts waiting for either an HTTP request or a termination signal. At this point, if the framework receives an HTTP request for the microservice, the framework invokes the function “onStartOperation” in order to notify the microservice that one operation will be executed. After invoking this function, the framework executes the public operation of the microservice, moving its state to “Microservice running”. Once the microservice has finished running its code, the microservice moves its state to “Microservice responding” in order to allow the framework to send the HTTP response to the client. Once the response has been sent, the framework invokes the function “onFinishOperation” in order to notify the microservice that the operation has finished and the corresponding HTTP response has been sent to the Client. After invoking this function, the microservice move again to “Microservice waiting”. At this point, if the framework receives another HTTP request, the same process is repeated. On the contrary, if the framework receives a termination signal, the framework invokes the function “onDestroyService” in order to close and clean all the resources of the microservice gracefully. Finally, the microservice is destroyed and the lifecycle ends with the state “Microservice destroyed”.

Regarding the four functions of the lifecycle, the framework does not implement any functionality for them, except for “onFinishOperation” that logs the sending error if occurred. The purpose of these functions is to allow developers to execute custom functionality at certain important moments, such as after the creation of the microservice, before and after the execution of the operations, and before the destruction of the microservice. For that, developers have to implement the functions together the public operations in the service functionality file that is loaded during the startup process. Thus, the framework is able to load the new behavior and override the default one.

Recapitulating, the lifecycle is a simplification of how the Microrestjs Framework manages the microservices during their existence. This simplification is focused exclusively on the microservices in order to complete the design picture of the whole platform.

3.3.5. Implementation

The Microrestjs Framework has been implemented using JavaScript programming language. JavaScript was developed in May 1995 by Brendan Eich in order to implement and run functionality of Web applications directly in the Web browser (client-side) instead of in the server. In these 20 years, JavaScript has been standardized as ECMAScript and has become one of the three most important technologies for Web development together with HTML and CSS. Although JavaScript has been used since the first moment, the development of real powerful

JavaScript applications has not been possible until the end of 2000s. At the end of 2000s, JavaScript experimented a big change due to the fifth version of the standard, the new generation of JavaScript engines (like V8 from Google) and the development of innovative technologies and tools (like jQuery, Node.js, Underscore.js, Angular.js, and Backbone.js, among others). All these innovations and technologies have contributed to the maturation of JavaScript, making the development of real powerful JavaScript applications possible. Nowadays, JavaScript is considered a high-level, untyped, dynamic, interpreted, and multi-paradigm (object-oriented, imperative and functional) programming language.

Regarding the decision of using JavaScript for implementing the Microrestjs Framework, the characteristics of JavaScript have influenced considerably in the final decision. First, the Microrestjs Framework has been designed to develop and deploy microservices without requiring compilation; thus, it is necessary to use an interpreted programming language. Second, the framework has been designed following an object-oriented approach; thus, it is necessary to use an object-oriented programming language. Third, the framework has to be able to add the loaded public operations to the corresponding RunnableService instances at runtime; thus, it is preferable to use a prototype-based programming language instead of a class-based programming language. Finally, the framework has to be able to route the incoming HTTP requests to the corresponding public operations at runtime; thus, it is preferable to use a functional programming language with first-class functions. Definitely, JavaScript satisfies all these programming requirements; however, the use of JavaScript also implies to address other undesired characteristics. For instance, JavaScript is an untyped language, which means that the types of the elements are not declared in the code. In other words, the types are assigned at runtime by JavaScript depending on the content of the elements. Moreover, JavaScript uses a dynamic type checking system to detect at runtime any incompatibility between the types that are being used at that moment. On the other hand, JavaScript also uses a type coercion system to convert implicitly the elements from one type to another "compatible" when the type checking detects a type incompatibility that could be solved with a type conversion, avoiding a runtime error. At this point, it is important to remark that most of these undesired characteristics are shared by most of the interpreted programming languages because they do not execute static verifications at compile time to check if the program is well typed, i.e. the program does not get stuck. In conclusion, JavaScript provides some desired characteristics that facilitate the implementation of the Microrestjs Framework, but it also provides other undesired properties that could cause failures or unexpected behaviors at runtime. Because the Microrestjs Framework does not live on JavaScript alone, it is necessary to know the rest of the technology stack in order to decide if JavaScript is the best option for implementing the framework.

As could be observed in the architecture diagram, the Microrestjs Framework has been implemented to run on Node.js. Node.js is a runtime environment for developing and running JavaScript applications at the server side. This environment was created in May 2009 by Ryan Dahl with the purpose of developing real-time Web applications that require a high throughput and scalability. For that, Node.js proposes a single-thread runtime environment.

Initially, this first characteristic may not be beneficial for developing high throughput applications because these applications would not be able to use the multicore capabilities of the current CPU architectures; however, Node.js proposes a completely asynchronous and non-blocking runtime environment. Basically, these two characteristics allow Node.js to schedule the execution of all the functions that are invoked, avoiding blocking the control flow of the current function. Thus, the invoked functions are executed at any time in the future (asynchronously) and their results are returned through the invocation of a callback function. Considering all these aspects, a single-thread runtime environment like Node.js could obtain a high throughput if and only if the applications are developed as a set of small, asynchronous and non-blocking functions in order to maximize the use of CPU and simulate a concurrent environment. On the one hand, this model presents several advantages. First, Node.js applications usually require less memory and CPU resources because it uses only one thread in memory. Second, Node.js applications can usually be scaled easily because there seems to be less concurrency problems. Third, Node.js applications are usually able to handle a larger number of requests than similar applications that use other Web platforms because each request requires fewer resources. On the other hand, this model also presents some drawbacks. First, Node.js applications have to be designed and implemented for running asynchronously without blocking the control flow because they run using only one thread. Second, Node.js applications should be composed of short and non-intensive CPU functions because the only thread that is used should not be blocked. Third, Node.js applications should only use asynchronous and non-blocking third-party libraries and frameworks because they are also executed in the same thread as the applications. In conclusion, Node.js can provide some important properties, such as scalability and performance, to the Microrestjs Framework if it is implemented according to the Node.js model.

Similar results to Node.js could be obtained using other platforms; however, most of them have not been designed asynchronous since the beginning, they have been adapted to apply the same model as Node.js. As a result, they usually mix synchronous and blocking APIs with asynchronous and non-blocking APIs, hindering the development of fully asynchronous applications. Likewise, the third-party libraries have the same problem. For Node.js, they have been designed asynchronous since the beginning because they were created after Node.js. However, for other platforms, many third-party libraries are synchronous because their platforms have a synchronous part. In conclusion, Node.js has a large community that develops all type of libraries following its model from the beginning.

According to the previous analyses, it can be concluded that the use of JavaScript and Node.js seems to be a good combination for implementing the Microrestjs Framework. Apart from these core technologies, it is important to highlight the use of JSON, NPM, Express Framework, and OpenSSL. JSON is used for describing the microservices and for exchanging information between them. NPM is used for managing the third-party dependencies of the framework. Express Framework is used for routing the HTTP requests to the corresponding microservice. Finally, OpenSSL is used for generating the platform credentials and establishing the network communications over TLS.

Once all the most important technologies for the implementation have been presented, it is important to detail how the problems of JavaScript type system have been addressed in order to minimize their impact in the framework. Basically, three approaches could be applied:

1. First, it can be used some programming languages (e.g. TypeScript) that compiles to plain JavaScript. In general, these programming languages apply static verifications for checking types during the compilation. As a result, the auto-generated JavaScript can guarantee similar behavior that compile programming languages. The only problem of this approach is the use of JavaScript libraries because they are compatible with these programming languages but they are not typed. As a consequence, it is necessary to type these libraries to perform a complete static verification.
2. Second, it can be used static verification checkers (e.g. Flow) to verify directly JavaScript. The main advantage compared to the previous approach is that there is no necessity of using another programming language different to JavaScript. With respect to the disadvantages, this approach presents a similar problem that the previous one. All the JavaScript code, including third-party libraries, has to be annotated with types in order to check the types statically.
3. Finally, it can be applied good programming practices in order to reduce the possibility of having runtime errors or unexpected behaviors. For instance, the public functions can check if all of the received parameters are from the type declared in the documentation in order to avoid unexpected behaviors when they are used. Similarly, the results of the functions can also be checked before using it. Additionally, automatic analysis tools (e.g. JSHint, JSCS, etc.) can be used for detecting common coding problems and enforcing a uniform coding style.

In our case, the chosen approach has been the third one because the others require much more effort to guarantee a minimum level of quality. On the contrary, the third approach obtains satisfactory results with only a little effort.

Summarizing, the Microrestjs Framework uses several technologies that has been chosen carefully in order to facilitate the development of the desired features. Moreover, some tactics have also been applied with the purpose of increasing the quality of the framework and reducing the possibility of having unexpected behaviors at runtime.

3.3.6. Testing

From the beginning, the quality of the platform has been one of the most important aspects of the project because this thesis has pursued the development of a production-ready prototype of the Microrestjs Framework. Thus, the framework has been tested exhaustively in order to assure the minimum quantity of defects.

As any other area of software engineering, testing has to be applied using systematic techniques in order to guarantee satisfactory results. In our case, the framework has been tested using black-box and white-box techniques. On the one hand, the black-box techniques use the specification of the software to generate the test cases without requiring the code. On the other hand, the white-box techniques require the code to generate the test cases according to the current implementation.

First, the framework has been tested systematically using two black-box techniques, Equivalence partitioning and Boundary values, in order to find defects with respect to the specification. The purpose of these techniques is to check that the output of an element (program, function, object, etc.) is the expected with respect to the input and according to the specification. Thus, a valid input should generate the expected output defined in the specification. And, an invalid input should generate the expected output defined in the specification. In brief, if the output is not the expected according to the specification, either the element has a defect or the test case has been implemented wrongly. Therefore, these black-box techniques can help minimize unexpected behaviors caused particularly by the JavaScript type system. Finally, it is important to remark that the test cases created by these black-box techniques can be easily automated using some testing tools.

Then, the framework has been tested using white-box techniques in order to complement the previous test cases covering the parts of the code that have not been tested yet. Thus, the purpose of these test cases is to check unusual scenarios that could cause the crash of the framework if they are not handled properly. For generating the white-box techniques, a coverage analysis tool is used for knowing which branches and decisions of the code have not been covered by previous test cases. Afterwards, specific test cases are created to cover each of these parts of the code. Finally, it is also important to remark that the test cases created by the white-box techniques can be easily automated using some testing tools.

In particular, 245 test cases have been created using black-box and white-box techniques to assure the correct operation of the framework. These test cases cover most of the static functionality that does not require sending and receiving HTTP requests. Furthermore, these test cases can be used for avoiding regressions in future versions. Moreover, all these test cases have been automated using the following tools: Mocha, Should.js, Mockery and Istanbul. Mocha is a testing framework for creating and running test cases on Node.js. Should.js is an assertion library for Node.js that facilitate the creation of the test cases following a Behavior Driven Development (BDD) style. Mockery is a library that simplifies the use of mocks for the test cases. Finally, Istanbul is a code coverage tool that analyzes which parts of the code have been covered by the run test cases.

Lastly, the framework has been tested manually to assure the correct operation of the dynamic features that require deploying and executing the microservices. Using manual testing, the following features and characteristics have been tested:

1. The microservices use TLSv1.1 or TLSv1.2 for network communications and encrypt all the transmitted data.
2. All the deployed microservices register correctly their information in the service directory.
3. The service directory provides the information of the registered microservices, if they are available at that moment.
4. The framework responds with error 503 SERVICE UNAVAILABLE if the desired microservice is not available for executing the request.
5. The framework responds with error 404 NOT FOUND if the microservice does not have the desired public operation.
6. The framework responds with error 405 METHOD NOT ALLOWED if the desired public operation does not accept the used HTTP method.
7. The framework responds with error 401 UNAUTHORIZED if the request cannot be authenticated for its execution.
8. The framework responds with error 403 FORBIDDEN if the request does not have authorization to be executed.
9. The framework responds with error 400 BAD REQUEST if the request does not satisfy the conditions defined in the service description for executing the desired operation. Specially, it has been placed emphasis on checking the parameters of the request and their types exhaustively.
10. The framework responds with error 500 INTERNAL SERVER ERROR if the request cannot be processed because of an unexpected situation.
11. The framework executes the correct operation if the request satisfies the conditions defined in the service description.
12. The framework executes the lifecycle functions (onCreateService, onStartOperation, onFinishOperation, and onDestroyService) when corresponds.
13. The framework is able to make the local verifications and execute correctly remote microservices using the CallableServices.
14. The framework shuts down correctly the entire platform when a termination signal is received.

The decision of testing these features manually is not because they cannot be automated. It is because the automation of these test cases requires much more effort than if they are tested manually. Thus, it has been decided to use that effort in other important tasks, such as implementation and documentation, and postpone the automation of these test cases for the future.

To sum up, the framework has been tested exhaustively using systematic techniques with the purpose of providing a platform ready for production. Nevertheless, the reader should be aware that the current maturity is beta, which means that future versions of the framework can change abruptly causing backwards incompatibilities.

3.4. Core services

Lastly, Microrestjs proposes three core services (the service directory, the authentication service, and the authorization service) to provide common features that are used by most of the projects. At the end, the purpose of these core services is to facilitate the development of microservices, without spending time in secondary tasks. For that, the core services have to be fully integrated within the service descriptions and the Microrestjs Framework in order to offer a platform ready for designing, developing and deploying microservices.

3.4.1. Service directory

The service directory is a core service that acts as an intermediary between the microservices with the purpose of increasing their visibility and interoperability. Basically, the service directory is like the yellow pages because it allows the publication and search of microservices' information. In particular, the service directory provides two functionalities: (1) it allows microservices to register their real location, and (2) it allows microservices to look up the location of other ones. Once the real location of a microservice is known, the rest of its information can be retrieved directly from the microservice itself.

The service directory has been designed to be completely transparent from the microservices point of view. Actually, the microservices only have to define some related aspects in their service description in order to allow the Microrestjs Framework to interact automatically with the service directory. Specifically, the microservices have to define the service directory and the other microservices (dependencies) that are going to be used.

Figure 30 illustrates the operation of the first functionality of the service directory, i.e. how microservices can register their location in the service directory. More precisely, the service directory provides a public operation, called "register", to receive the location from the microservices. Then, the public operation processes the information and responds with 204 NO CONTENT if the location has been published correctly.

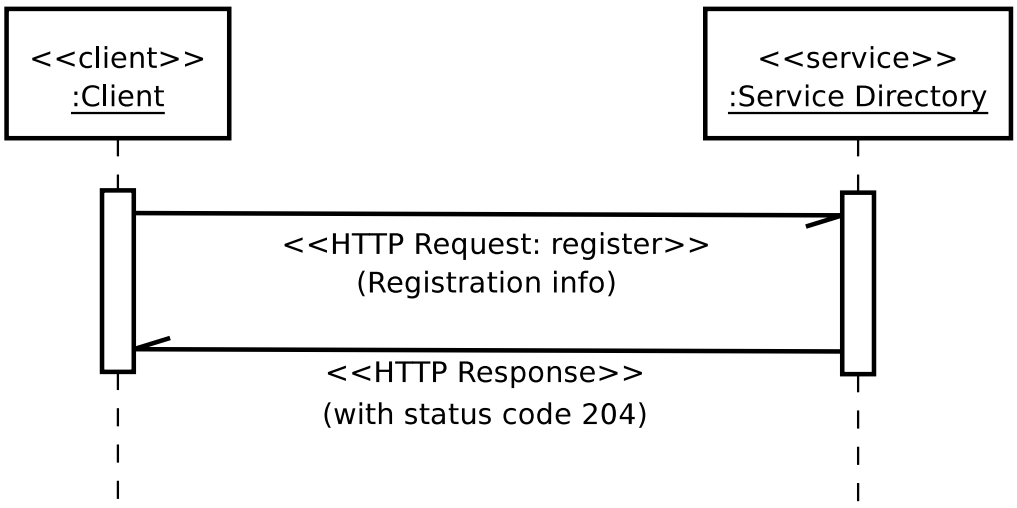


Figure 30. Service directory - Register operation

On the contrary, Figure 31 shows how the service directory responds if the register operation receives a request with some information that cannot be understood and processed. In this case, the service directory responds with error 400 BAD REQUEST to inform the client that its information could not be published in the service directory.

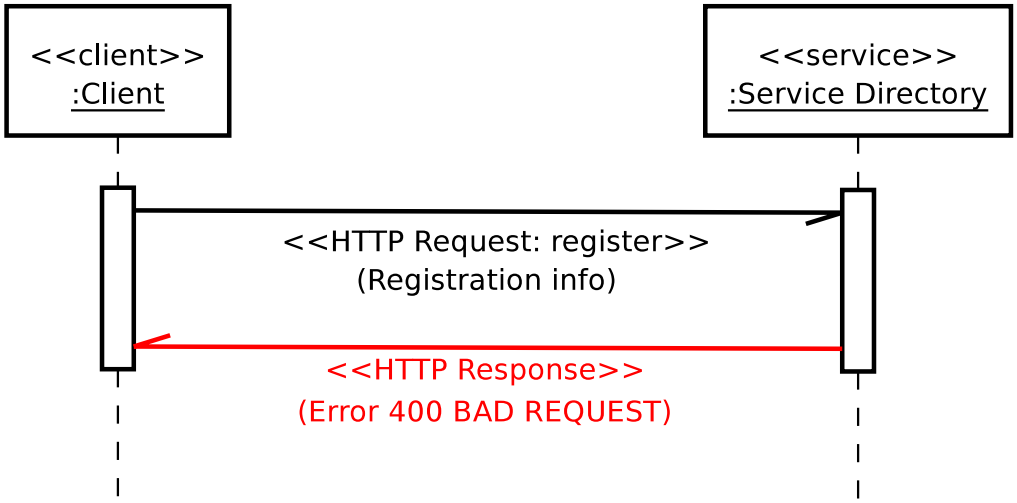


Figure 31. Service directory - Registration error

Figure 32 illustrates the operation of the second functionality of the service directory, i.e. how microservices can look up the location of other ones in the service directory. More precisely, the service directory provides a public operation, called "lookup", that retrieves the location of any microservice that has been registered previously. If the desired microservice has been registered in the service directory, the public operation responds with the location of the corresponding microservice.

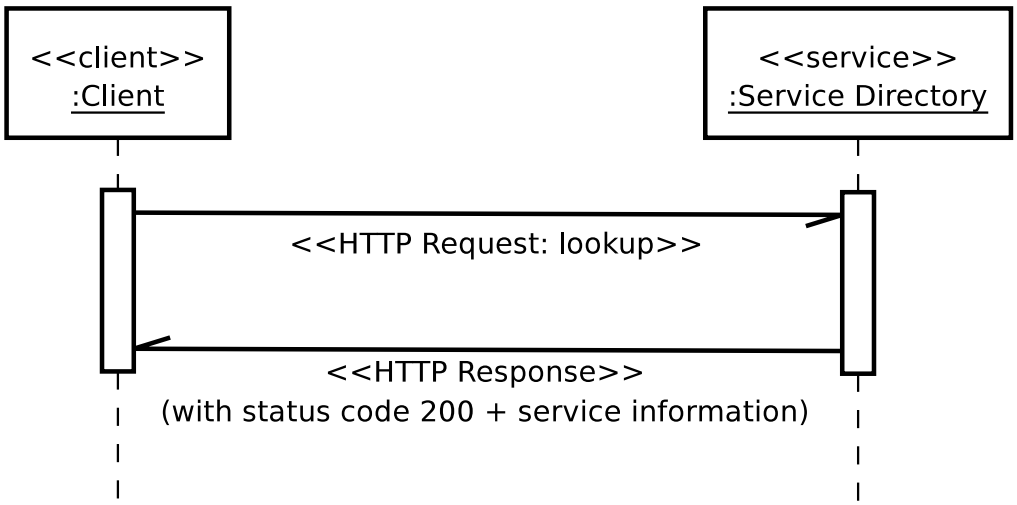


Figure 32. Service directory – Look up operation

On the contrary, Figure 33 shows how the service directory responds if the lookup operation cannot find the desired microservice. In this case, the service directory responds with error 404 NOT FOUND to inform the client that the desired microservice is not registered in the service directory.

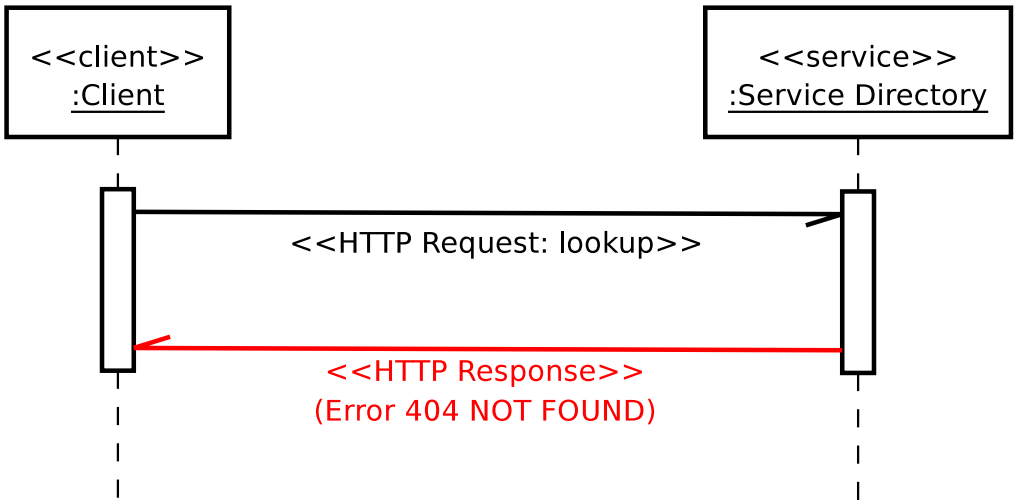


Figure 33. Service directory - Lookup error

Although this core service is used most of the time by the Microrestjs Framework, the previous diagrams demonstrate that the service directory is a completely independent microservice that can be executed manually by others. Thus, the integration of this core service with the Microrestjs Framework is only for facilitating the development of microservices within the Microrestjs environment.

Regarding the security, the service directory does not provide a public operation for unregistering microservices. The reason is because if this public operation exists, malicious microservices would be able to unregister other microservices indiscriminately. As a solution, the service directory is in charge of checking if the microservices are still available when they are looked up by another microservice. If the microservice is still available, the service directory responds with its locations. Otherwise, the service directory removes the microservice and tries to find another one. If there is no other microservice available, the service directory responds with error 404 NOT FOUND.

Another important security aspect that should be considered is how the service directory can control the registration of microservices. The problem is that malicious microservices can register themselves simulating to be another different microservice. The solution to this problem is not trivial. Some existing platforms propose that the registration of services should be done manually by human administrators in order to avoid similar problems. Nonetheless, our purpose is to minimize the interaction of human resources automating most of the tasks. Thus, Microrestjs proposes that the service directory can exclusively register microservices that are deployed in the same local network, denying any registration request from the Internet. Basically, this solution is based on trust. Supposedly, the microservices, which are deployed in your local network, are not malicious; whereas the microservices, which are deployed on the Internet, can be

malicious because there are not enough guarantees about their authenticity. Definitely, the solution proposed by Microrestjs is not perfect, but it can help minimize the risk considerably. Finally, it is important to highlight that this solution is not implemented by default and it is only a suggestion for increasing the security of this core service.

In the same fashion, the lookup public operation can be considered risky because the service directory could give the location of malicious microservices. It is important to remark that Microrestjs allows using several service directories to look up the desired microservices defined as dependencies. In brief, if developers trust in an external microservice, they should also trust in the service directory in which the microservice has been registered by its owner. In conclusion, Microrestjs cannot provide specific security measures because the decision of which service directory is trustworthy has to be taken by developers.

Recapitulating, Microrestjs provides a simple core service for registering and looking up microservices. Moreover, most of the required interactions with this core service are done transparently by the Microrestjs Framework because of the full integration of the core services with the rest of the elements of Microrestjs.

3.4.2. Authentication and authorization

Authentication and authorization are two security core services that work together in order to accept or reject the HTTP requests that the platform receives for executing any of the deployed microservices. Basically, the authentication service is in charge of checking the authenticity of the credentials that are sent within the HTTP requests. On the other hand, the authorization service is in charge of checking if the credentials of the HTTP request have enough rights to execute the desired public operation.

Similar to the service directory, the authentication and authorization services are fully integrated with the rest of the elements of Microrestjs. On the one hand, the service descriptions allow defining the authentication and authorization protocols that the microservices require for executing their public operations. On the other hand, the Microrestjs Framework is able to handle transparently all the protocols supported by the service descriptions without any extra effort for developers.

Regarding the supported authentication and authorization protocols, Microrestjs only supports officially two different protocols at this moment: none and basic. If the microservices do not require any authentication and authorization for executing their operations, developers have to specify “none” in the service description in order to declare that the Microrestjs Framework has to skip the authentication and authorization for all the HTTP requests. Otherwise, if the microservices require authentication and authorization for executing their operations, developers have to specify “basic” in the service description in order to declare that the Microrestjs Framework has to use the basic authentication and authorization protocol for all the HTTP requests. It is important to remark that the basic protocol is based on the Basic Access Authentication [18] scheme for HTTP.

Figure 34 illustrates how the basic protocol authenticates and authorizes the incoming HTTP requests. The protocol begins with a client sending an HTTP request for executing a public operation. If such public operation requires basic authentication and authorization, the client has to include a basic authorization header in the HTTP request in order that the Microrestjs Framework can validate the HTTP request. In short, the basic authorization header, which is completely specified in [18], includes the client's credentials (username and password) encoded in Base64. When the Microrestjs Framework receives the HTTP request with the basic authorization header, it decodes the credentials and checks if they are valid for executing the public operation. For that, the Microrestjs Framework sends the credentials to the Basic Authentication Service that (1) verifies if the credentials are genuine and (2) returns the client's identity. Once the Microrestjs Framework knows who is the client, it can use the Basic Authorization Service to verify if the client has enough rights for executing the desired public operation. If the Basic Authorization Service responds with 204 NO CONTENT, the Microrestjs Framework executes the public operation because the client has authorization for executing such public operation. Finally, the public operation responds to the client with the corresponding result.

The above description of the basic protocol assumes that all the steps are executed successfully; however, the reality is that the basic protocol can fail by different reasons. Firstly, the client could attempt to execute a public operation without sending the basic authorization header. In this case, the Microrestjs Framework cannot obtain the credentials to check them and, therefore, it responds with error 401 UNAUTHORIZED, as Figure 35 shows. Secondly, the client could attempt to execute a public operation using wrong credentials. In this case, the credentials are sent to the Basic Authentication Service in order to verify the client's identity. Nevertheless, the Basic Authentication Service cannot obtain the identity and responds with error 401 UNAUTHORIZED, as Figure 37 shows. Finally, the client could attempt to execute a public operation without enough rights. In this case, the Microrestjs Framework is able to obtain the client's identity, but the Basic Authorization Service responds with error 403 FORBIDDEN because the client does not have enough rights for executing the operation, as Figure 37 shows.

Essentially, Microrestjs proposes a simple protocol for all these microservices that need to control accurately who can execute their public operations. The problem is that the security of this protocol is considered weak because the client's passwords are sent in plaintext without any encryption. Thus, anyone with access to the HTTP requests can obtain and use the client's passwords without any problem. In the case of Microrestjs, the password remains safe during its transmission over the network because TLS encrypts all the network communication between the clients and the microservices. Therefore, the only ones that can access to the client's passwords are the Microrestjs Framework and the microservices that receive the HTTP requests. In consequence, malicious microservices could obtain and use the client's passwords fraudulently. In conclusion, the security of this protocol lies in the client's trust for using those microservices.

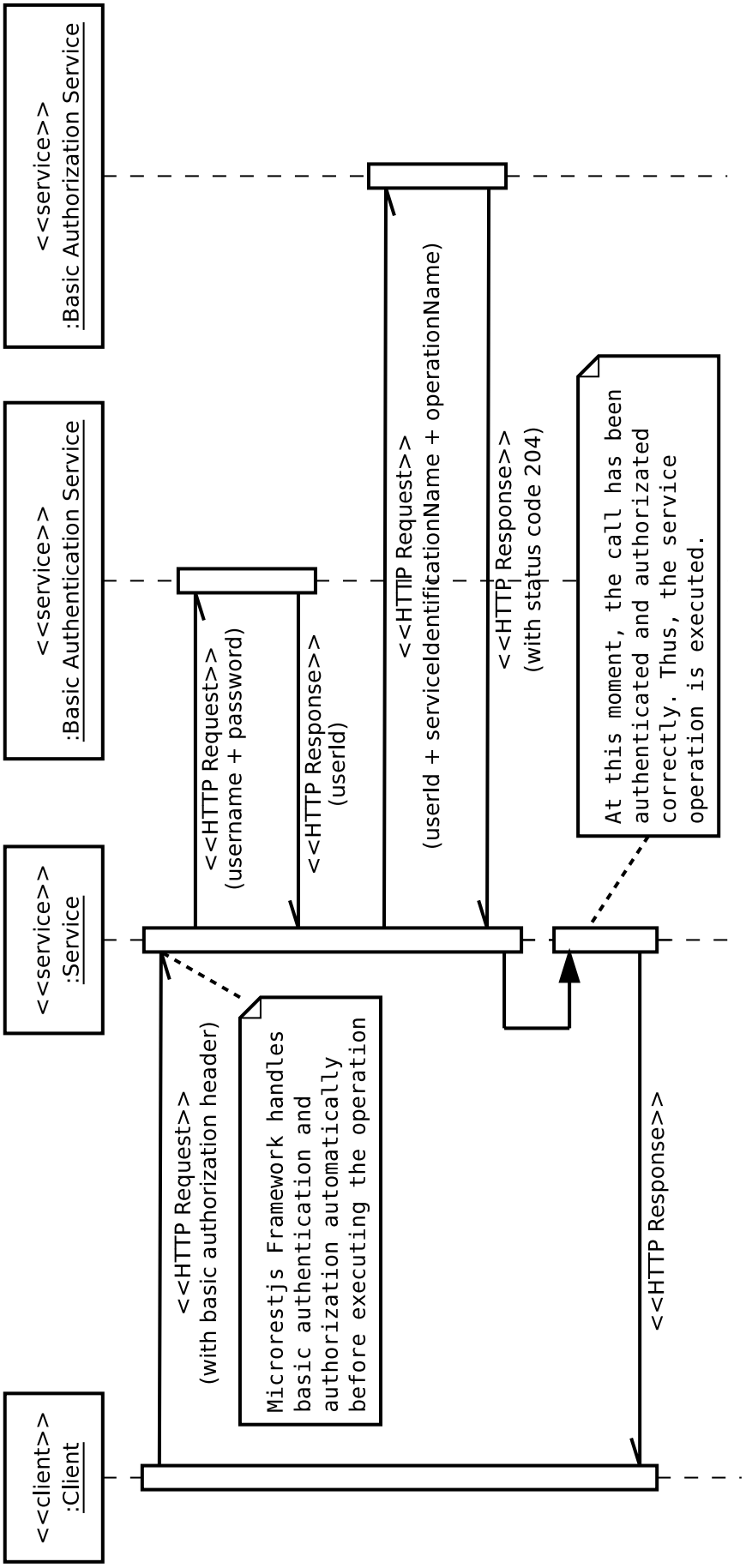


Figure 34. Basic authentication and authorization protocol

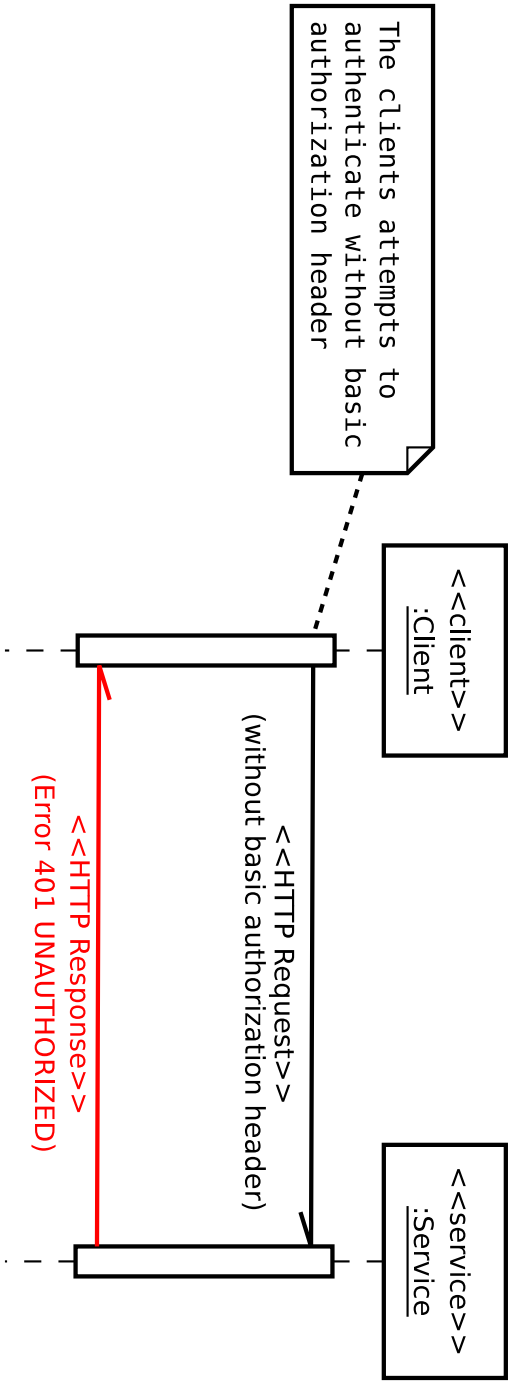


Figure 35. Basic authorization header error

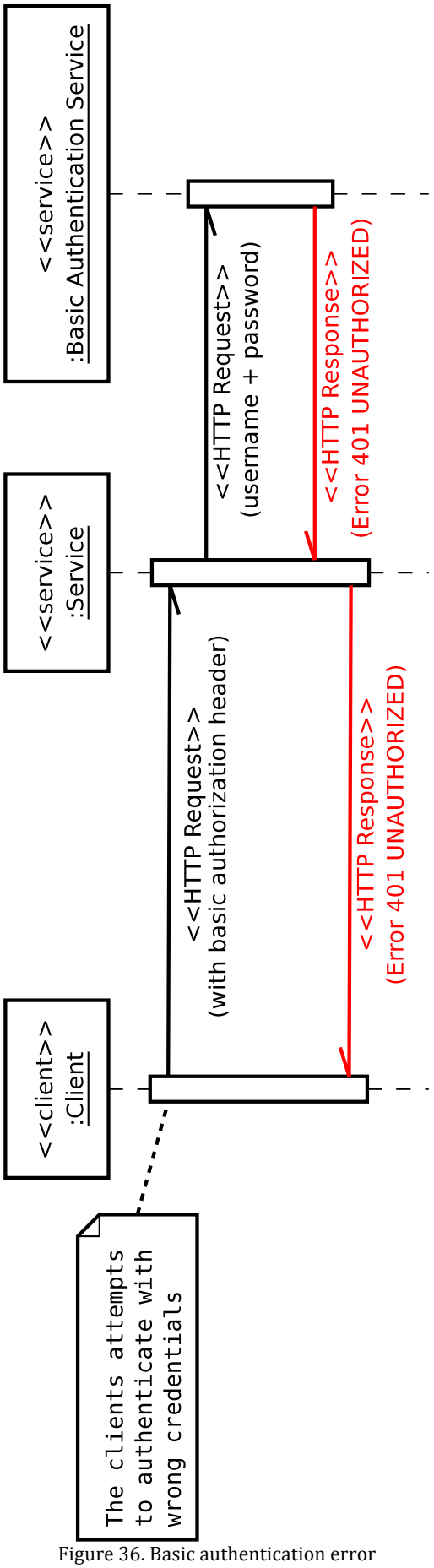


Figure 36. Basic authentication error

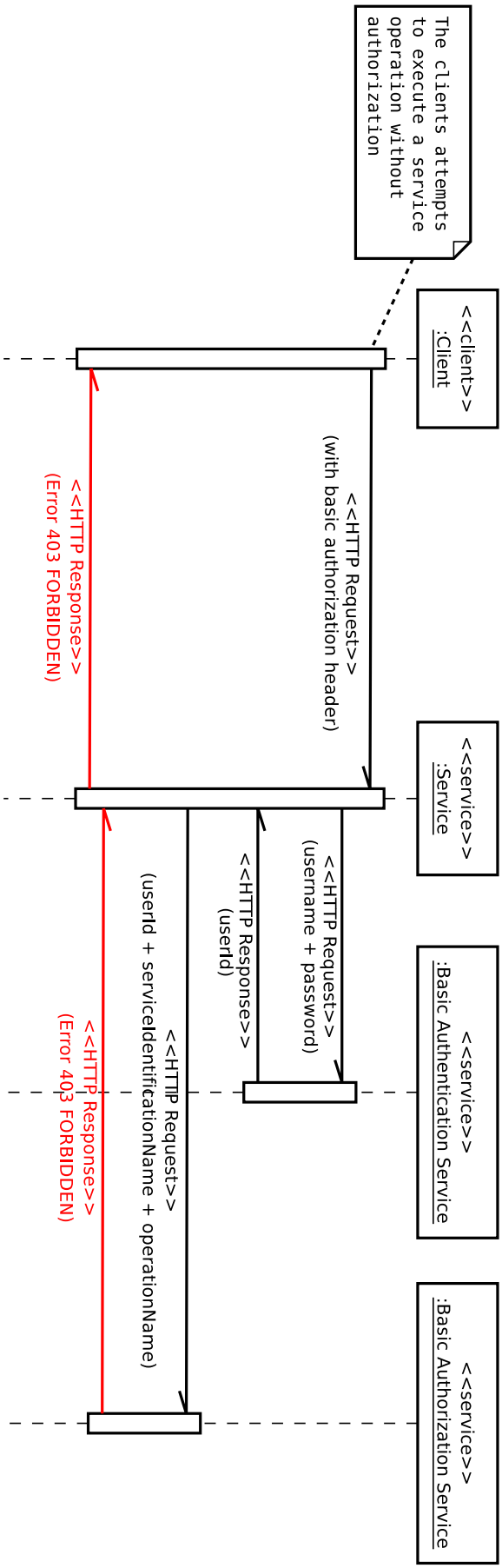


Figure 37. Basic authorization error

The literature proposes dozens of authentication and authorization protocols that are considered safer than the basic protocol supported by Microrestjs. As future work, Microrestjs shall include some of these protocols for those microservices that require higher levels of security. Thus, this thesis only presents some general aspects about these protocols and how they can be integrated within Microrestjs.

In general, most of these safer protocols replace the passwords with tokens. Thus, their security depends on how the tokens are created and transmitted. For instance, if the token is generated exclusively applying a hash function to the password, the corresponding protocol would be marginally safer than the basic protocol because malicious microservices would not have the password in plaintext, but they could impersonate the client using the tokens. On the contrary, if the token is generated and agreed between the client and the service, the corresponding protocol would be considerably safer than the previous ones because each token is unique and only works for one client and one service. In conclusion, the generation of tokens plays a fundamental role in the security of token-based protocols, as can be observed. For more details about token-based protocols, the reader is encouraged to read the specification of any popular authentication and authorization protocol, such as OAuth [19], OpenID [20], Kerberos [21], and Venice [22], among others.

Usually, the token-based protocols are composed of two phases. During the first phase, the client obtains somehow a valid token. Then, during the second phase, the client uses the token for executing a microservice remotely. Regarding the first phase, this thesis does not propose any specific suggestion because it depends strongly on how the protocol creates the token. With respect to the second phase, this thesis proposes the schema shown in Figure 38. As can be observed, this schema is similar to the basic authentication and authorization schema. For this reason, this schema can be integrated with the rest of the Microrestjs components easily. In the token-based protocol, the client sends an HTTP request with a token in order to execute a public operation that requires token authentication and authorization. When the Microrestjs Framework receives the HTTP request with the token, it uses the Token Authentication Service to verify if the token is genuine and obtain the client's identity. Once the Microrestjs Framework knows who is the client, it uses the Token Authorization Service to verify if the client has enough rights for executing the desired public operation. In such case, the Microrestjs Framework executes the corresponding public operation.

Concerning the possible errors that can occur during the authentication and authorization, they are also similar to the basic protocol errors. Firstly, the client could attempt to execute a public operation without sending the token. In this case, the Microrestjs Framework is not able to check the token and responds with error 401 UNAUTHORIZED, as Figure 39 shows. Secondly, the client could attempt to execute a public operation using a wrong token. In this case, the Token Authentication Service is not able to verify the client's identity and responds with error 401 UNAUTHORIZED, as Figure 40 shows. Finally, the client could attempt to execute a public operation without enough rights. In this case, the Token Authorization Service responds with error 403 FORBIDDEN because the client does not have enough rights for executing the operation, as Figure 41 shows.

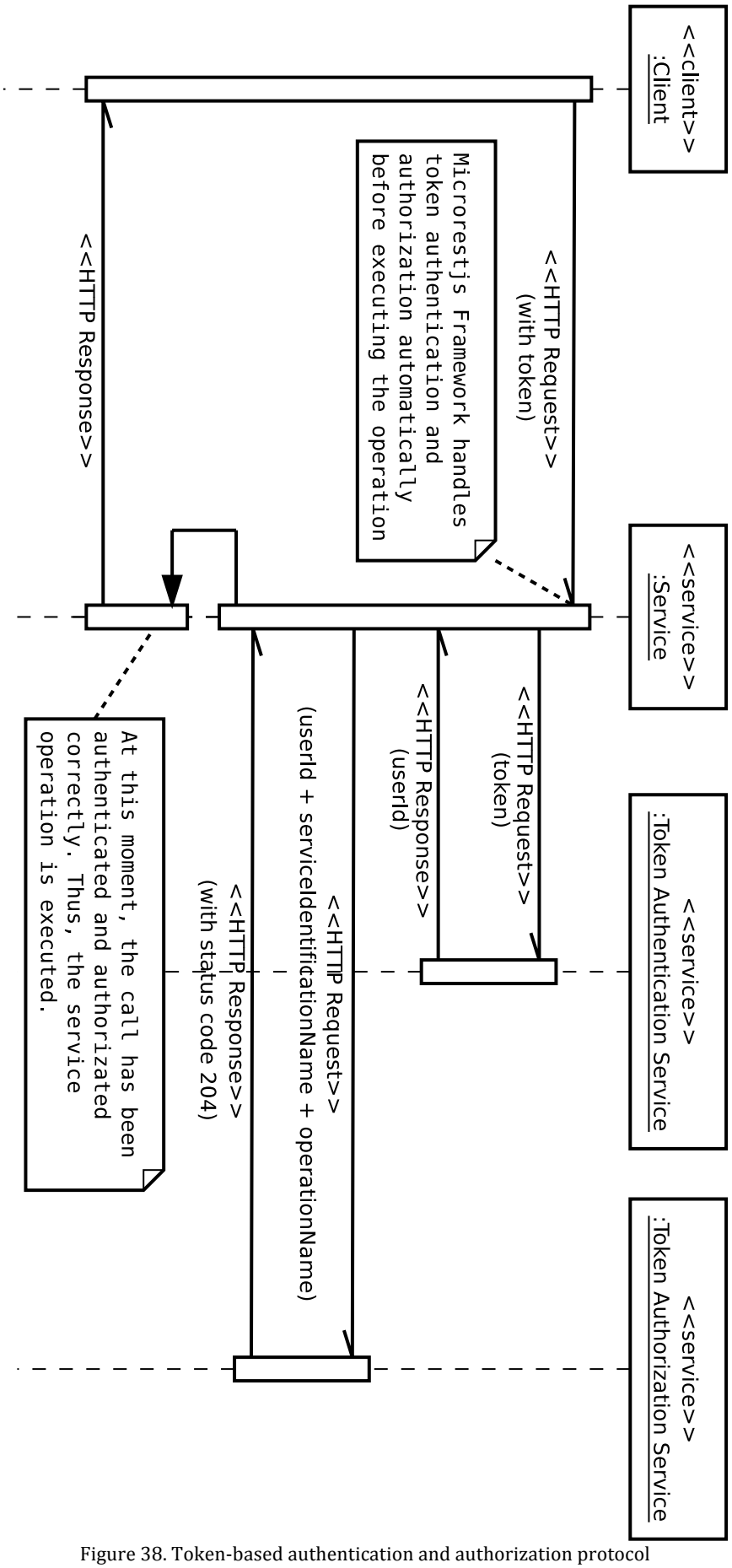


Figure 38. Token-based authentication and authorization protocol

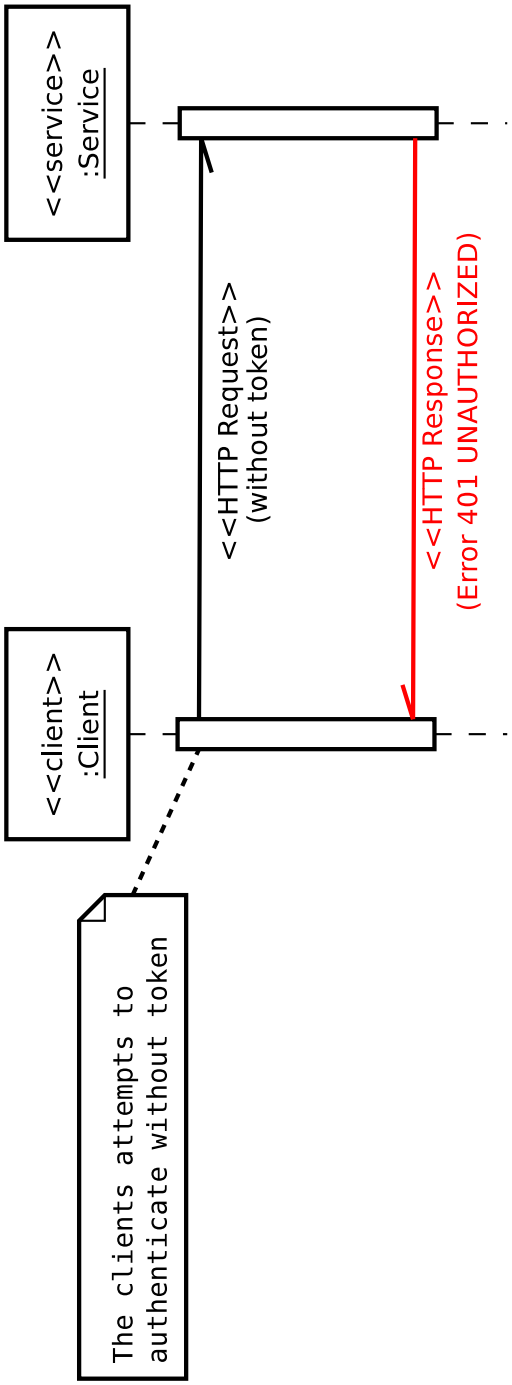


Figure 39. Token error

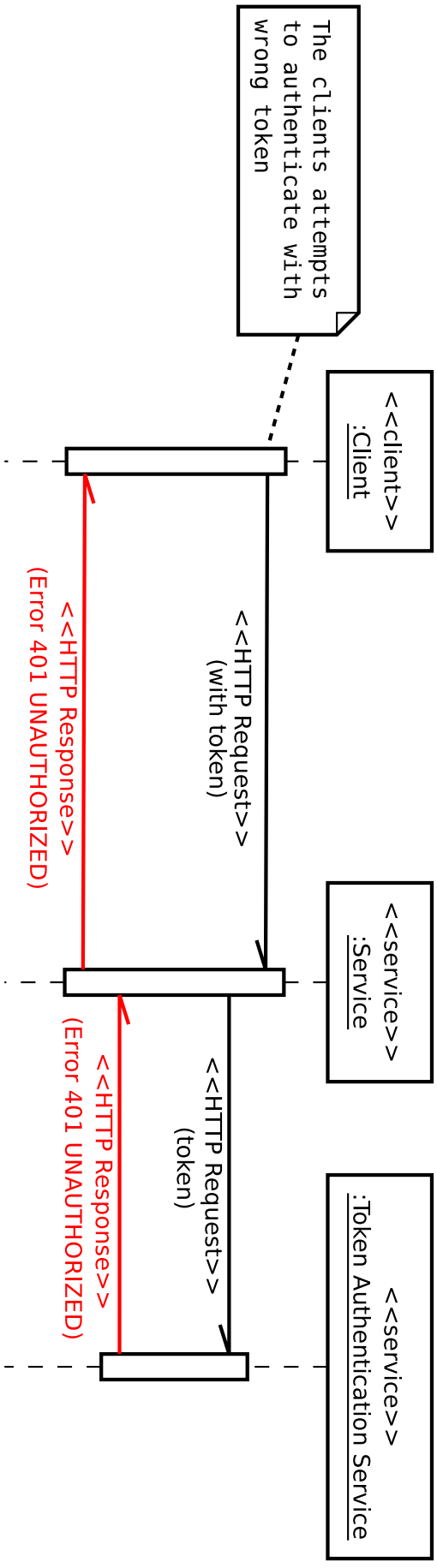


Figure 40. Token authentication error

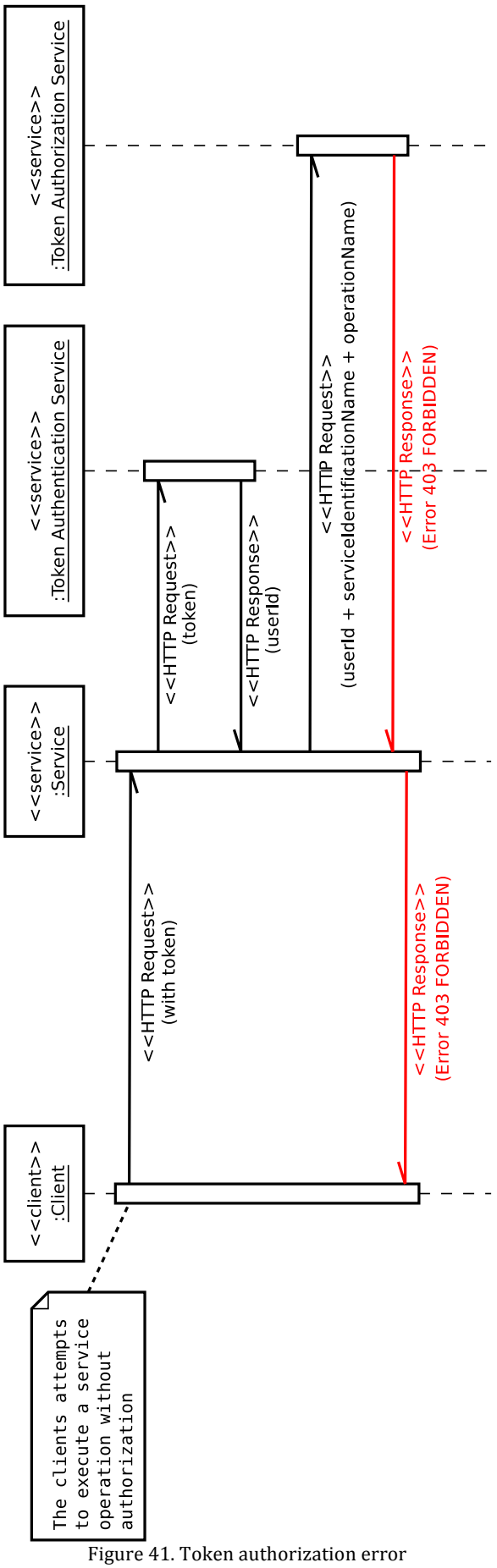


Figure 41. Token authorization error

Definitely, the token-based protocols provide some security properties that some microservices could consider important for their clients. For this reason, Microrestjs aims to implement this type of protocols in future versions. Until that moment, Microrestjs only supports the basic protocol officially. Although this basic protocol does not offer a high level of security, it is enough for many microservices. In any case, the microservices could change the authentication and authorization protocol in the future without requiring big changes because the objective is that Microrestjs Framework can use the token-based protocols transparently, similar to the basic protocol.

Apart from the protocols defined in this thesis, the literature is also researching other authentication and authorization mechanisms. With respect to the authentication, there seems to be some efforts focused on the use of public key infrastructures (PKI) for replacing passwords and tokens with certificates in order to guarantee unequivocally the client's identity. Regarding the authorization, there seems to be some efforts focused on the use of access control mechanisms (such as MAC, DAC, and RBAC, among others) for controlling and restricting the resources that each client can access and use.

In conclusion, the authentication and authorization is a very important aspect inside the Microrestjs environment because this core feature provides an extra security layer to the microservices without requiring any extra effort. Nevertheless, this feature is not perfect at this moment and requires more work and research in order to offer solutions to the necessities that some microservices may have.

4. Evaluation

This thesis has created Microrestjs, an innovative environment for designing, developing and deploying microservices. This environment is composed of new methods, tools, and technologies that have been designed to work together with the purpose of transforming the way developers create new software.

At this point, it is necessary to compare Microrestjs with the reference technologies in order to understand and evaluate the real impact of Microrestjs. In brief, the reference technologies are those technologies that have influenced the design and development of Microrestjs. In our case, the reference technologies are SOA, W3C Web services, RESTful Web services, and microservices architectures. Thus, the following points explain the relations between Microrestjs and the reference technologies with the purpose of demonstrating that Microrestjs has solid foundations.

In first place, SOA highlights the importance of four concepts (capability, service, consumer, and provider) and three aspects (visibility, interaction, and real world effects) for the development of any service-oriented system. Microrestjs agrees with SOA that these concepts and aspects are essential for any service-oriented system. Thus, Microrestjs uses analogous concepts and provides solutions for enabling the three aspects. For the visibility aspect, Microrestjs provides a service directory in order that microservices can share their information with others. For the interaction aspect, Microrestjs provides a framework that facilitates all the interactions with clients and other microservices. For the real world effects aspect, Microrestjs facilitates the development of microservices that can produce changes in the world. As can be observed, Microrestjs and SOA share the basic knowledge that allows the development of services.

On the other hand, SOA also proposes eight principles (reusability, formal contract, loose coupling, abstraction, composability, autonomy, statelessness, and discoverability) to design high-quality services. All these principles are completely valid for the development of microservices. Therefore, Microrestjs aspires to help developers in the application of these principles. Regarding the reusability principle, Microrestjs aims to develop several simple microservices in order to combine them for creating larger systems. Regarding the formal contract principle, Microrestjs proposes the use of service descriptions to define the microservices formally in order to be able to automate certain tasks. Regarding the loose coupling principle, Microrestjs provides mechanisms to manage transparently the dependencies between microservices as well as their interactions in order to reduce the coupling impact. Regarding the abstraction principle, Microrestjs proposes the use of service descriptions to define the interfaces of the microservices in order that clients can use them without knowing their implementations. Regarding the composability principle, Microrestjs provides mechanisms that facilitate the interaction between microservices, enabling the creation of façades that can act as intermediaries of other microservices. Regarding the autonomy principle, Microrestjs provides a lifecycle to the microservices in order to facilitate their development without interfering absolutely in their logic. Regarding the statelessness principle, Microrestjs

suggests the creation of stateless microservices in order to guarantee the scalability of microservices. Regarding the discoverability principle, Microrestjs provides a service directory and other mechanisms to enhance the discoverability of microservices. As can be observed, Microrestjs believes in these principles and attempts to facilitate their application, but some of them depend mainly on how the service-oriented systems are designed by developers.

Finally, SOA emphasizes other properties such as flexibility, uniformity, and interoperability that Microrestjs also considers important. Regarding the flexibility property, Microrestjs pursues to be able to adapt the microservices rapidly to changes in the environment. Regarding the uniformity property, Microrestjs aims to provide uniform mechanisms and interfaces for executing other microservices. Regarding the interoperability property, Microrestjs enables interoperability because it would not make sense if the deployed microservices could not interact between them. In conclusion, Microrestjs has been influenced considerably by SOA, making possible the development of SOA-compliant microservices with Microrestjs.

In second place, W3C suggests a standard architecture to design and develop SOA-compliant Web services using standard technologies (WSDL, SOAP, UDDI, XML, and HTTP). Microrestjs recognizes the importance of this standard architecture and technologies, but some of them are not suitable for Microrestjs because it would be like using a sledgehammer to crack a nut. Regarding the WSDL technology, Microrestjs prefers to use Microrestjs Service Description Specification, which has been inspired by WSDL, because WSDL does not satisfy accurately the requirements of Microrestjs in spite of being possibly much more powerful. Regarding the SOAP technology, Microrestjs does not require using any messaging framework because Microrestjs uses directly all the potential of the HTTP protocol to send and receive messages. Regarding the UDDI technology, Microrestjs proposes a simple service directory to publish the location of the microservices because Microrestjs does not require the large number of features that UDDI provides. Regarding the XML technology, Microrestjs prefers to use JSON because it is the new de facto standard for Web development and provides higher performance than XML due to its simplicity. Regarding the HTTP technology, Microrestjs uses also this protocol for all the network communications. As can be observed, Microrestjs requires similar technologies to W3C Web services in order to develop microservices; however, some of them do not satisfy the requirements of Microrestjs, forcing Microrestjs to find an alternative solution.

On the other hand, W3C highlights the importance of using standard technologies in order to support platform-independent services and guarantee machine-to-machine interoperability. Microrestjs agrees with W3C that standard technologies are preferable if they satisfy the requirements; otherwise, Microrestjs prefers the use of non-standard technologies. Regarding the platform-independent feature, Microrestjs also guarantees such independency because Node.js is cross-platform. Regarding the machine-to-machine interoperability feature, Microrestjs provides mechanisms to enable the interoperability between machines. As can be observed, Microrestjs and W3C shares similar concerns, but they handle them differently.

Finally, W3C also supports additional interesting features for security, transactions, and choreography, among others. In fact, the environment provided by W3C is more powerful than the Microrestjs environment because it allows developing any type of service-oriented system. Nonetheless, the W3C environment is heavyweight and requires configuring exhaustively every aspect and technology in order to work properly. On the contrary, Microrestjs provides fewer features, resulting in a lightweight environment that does not require an extensive configuration to work properly. In conclusion, Microrestjs is not able to replace W3C in all the cases for the moment, but Microrestjs could be a better solution than W3C Web services in certain cases.

In third place, RESTful Web services respects the rules defined by REST architectural style with the purpose of achieving services with low latency, high scalability, high adaptability, and high extensibility. Microrestjs knows the importance of being able to provide these properties to its microservices. Thus, Microrestjs has been designed considering the REST architectural style in order to be able to develop REST-compliant microservices. An example of this is that Microrestjs does not support the use of cookies because they can violate the REST principle that states that all the interactions are stateless. Nevertheless, Microrestjs cannot respect completely REST architectural style because of the use of the HTTP protocol, but it attempts to respect the largest number of aspects of the REST architectural style.

On the other hand, RESTful Web services usually provide non-standard APIs to allow clients to execute their operations through URLs using the HTTP protocol for the network communications and the JSON format for the representations of the resources. Microrestjs uses exactly the same technologies (URL, HTTP protocol, and JSON format), but it prefers to use service descriptions to define the APIs of the services formally because Microrestjs seeks to automate certain tasks and facilitate the interoperability between machines.

Finally, RESTful Web services have not been standardized as W3C Web services. As a consequence, it exists different technologies and approaches to design and develop RESTful Web services with different features and characteristics. Moreover, the statistics indicates that RESTful Web services are significantly more used than W3C Web services. Therefore, it could be concluded that developers prefer to sacrifice some aspects such as discoverability and formal contracts in order to gain others such as simplicity and lightweight. Thus, Microrestjs knows that it can only compete and succeed in the market if it is able to provide similar features to RESTful Web services without sacrificing simplicity. For this reason, Microrestjs shares a large number of commonalities with RESTful Web services. In conclusion, Microrestjs has been influenced considerably by RESTful Web services, making possible the development of REST-compliant microservices with Microrestjs.

In last place, microservices architectures were created as an alternative to monolithic architectures in order to separate the functionality into several services. Also, microservices architectures emphasize the importance of running each service in an independent process and enabling lightweight mechanisms for

communication between them. Microrestjs facilitates the implementation of the functionality in separated microservices and enables lightweight communications using HTTP protocol. Moreover, Microrestjs allows developers to decide if the microservices are deployed in independent processes or in the same process but separated in "sandboxes". As can be observed, Microrestjs and microservices architectures share the same basic aspects for developing microservices.

On the other hand, microservices architectures usually exhibit nine common characteristics: componentization via services, organized around business capabilities, products not projects, smart endpoints and dumb pipes, decentralized governance, decentralized data management, infrastructure automation, design to failure, and evolutionary design. Regarding the componentization via services characteristic, Microrestjs allows developing microservices that can be used as components of larger systems. Regarding the organized around business capabilities characteristic, Microrestjs does not force developers to divide the functionality following any specific approach, but it believes that a good division of the microservices is separating them according to the business capabilities. Regarding the products not projects characteristic, Microrestjs proposes a transversal methodology that requires performing certain tasks along the different phases of the development lifecycle. Thus, Microrestjs believes that it could be better to follow a product-oriented development instead of project-oriented development. Regarding the smart endpoints and dumb pipes characteristic, Microrestjs applies this aspect perfectly because Microrestjs allows implementing sophisticated microservices (smart endpoint) that use HTTP protocol (dumb pipe) for communicating with each other. Regarding the decentralized governance characteristic, Microrestjs does not force developers to use any concrete technology for implementing the logic of their microservices. Thus, developers have the freedom of choice the best solution according to their necessities. Regarding the decentralized data management characteristic, Microrestjs does not force developers to use any specific data management technology either. Regarding the infrastructure automation, Microrestjs provides a launcher that can load, deploy and register all the microservices automatically. Regarding the design for failure characteristic, Microrestjs presents several automatic verifications to reduce the number of failures and provides mechanisms to handle the failures when they occur. Regarding the evolutionary design characteristic, Microrestjs facilitates that microservices can evolve progressively because Microrestjs is able to adapt transparently the topology when new microservices are deployed. As can be observed, Microrestjs exhibits the nine common characteristics that microservices architectures usually have.

Finally, microservices architectures seek to ensure the scalability of the systems without affecting considerably their general performance. This property is considered completely essential for the microservices of Microrestjs. In conclusion, Microrestjs has been influenced considerably by the microservice architectural style because they pursue and share the same objectives and principles.

In summary, Microrestjs shares a large number of commonalities with the reference technologies demonstrating that has been created on a solid base of knowledge.

After comparing Microrestjs with the reference technologies and observing that Microrestjs has potential to transform the way developers create new software, it is essential to verify that Microrestjs actually includes all the expected features. Regarding the first feature "A method to design, develop and deploy microservices", Microrestjs suggests a methodology that includes some specific tasks in the development lifecycle. Regarding the second feature "A lightweight tool to support the design of microservices using simple contracts or descriptions", Microrestjs supports service descriptions that respect the Microrestjs Service Description Specification. Regarding the third feature "A lightweight tool to implement easily the functional logic of microservices without spending time in other secondary tasks such as network communication or discoverability", Microrestjs provides a framework that manages transparently secondary tasks in order to allow developers to focus exclusively on the logic of their microservices. Regarding the fourth feature "A uniform interface to interact with other microservices of the platform", Microrestjs provides a uniform mechanism that allows executing any other microservice. Regarding the fifth feature "An automated tool to deploy microservices into the platform", Microrestjs provides a launcher that is able to deploy microservices automatically without user interaction. Regarding the sixth feature "An automated tool to discover other microservices of the platform", Microrestjs provides several mechanisms that work together with the purpose of registering and looking up microservices in a service directory. Regarding the seventh feature "A solution to offer complete freedom to Web developers with respect to the logic implementation and type of data store", Microrestjs does not lock developers in concrete technologies and allows them to use any technology for implementing the logic of their microservices. Regarding eighth feature "A solution to obtain compatibility with some previous technologies such as RESTful Web services and RESTful APIs", Microrestjs offers certain compatibility with RESTful Web services. On the one hand, the microservices could interact with RESTful Web services if their APIs are encapsulated within service descriptions. On the other hand, the RESTful Web services could also interact with microservices using the encapsulated API of the microservices. In the case of W3C Web services, the compatibility requires the creation of adapters in order to be able to support the different technologies that those services use. Regarding the last feature "some other out-of-the-box tools for logging, authentication, and authorization, among others", Microrestjs provides some extra features ready to be used by developers. In conclusion, Microrestjs includes all the expected features that were considered essential to ensure an attractive environment for developers.

Apart from the previous functional features, Microrestjs aims to offer other important properties such as scalability, simplicity, interoperability, security, reusability, discoverability, uniformity, flexibility, adaptability, extensibility, maintainability, and readability. Regarding scalability, Microrestjs allows developing microservices-oriented systems that can scale according to the necessities. Regarding simplicity, Microrestjs allows developers to focus exclusively on the logic of their microservices, thus reducing the size and complexity of their systems. Regarding interoperability, Microrestjs provides mechanisms to enable the interaction with clients and microservices. Regarding security, Microrestjs provides by default several security measures (such as

network communication over TLS, deployment in sandboxes, and authentication and authorization protocols, among others) for all the microservices. Regarding reusability, Microrestjs makes possible to reuse microservices across different systems using the dependency system. Regarding discoverability, Microrestjs enhances the visibility of the microservices with the service directory, being easier to find and use other microservices. Regarding uniformity, Microrestjs allows developing and using microservices uniformly with the Microrestjs Framework. Regarding flexibility, Microrestjs is able to adapt its topology in real time when a new microservice is deployed or a running microservice is shut down, facilitating that the system can evolved progressively according to the necessities. Regarding adaptability, Microrestjs allows adapting and deploying new versions of the microservices easily. Regarding extensibility, Microrestjs facilitates the interactions between microservices, enabling the possibility of extending their base behavior. Regarding maintainability, Microrestjs facilitates the maintenance of the systems because developers only have to maintain the logic of their microservices and not other additional aspects such as network communications, service discoverability, and deployment of microservices, among others. Regarding readability, Microrestjs is able to reduce the size and complexity of the microservices, resulting in a higher readability. In conclusion, Microrestjs promotes certain non-functional properties that can be beneficial for developers and their systems.

From the previous analyses, it is possible to deduce that Microrestjs could improve the main research aspects of Software Engineering. For instance, Microrestjs may increase the overall quality of the software, may reduce the software complexity, may decrease costs, may shorten time-to-market, and may minimize the risk of project failure. In conclusion, Microrestjs might be considered a real alternative for the development of new software solutions.

To conclude, Microrestjs has demonstrated thoroughly the large potential of its innovative environment for designing, developing and deploying SOA- and REST-compliant microservices. Thus, it can be asseverated that Microrestjs has successfully met all the initial objectives and has even exceeded the expectations of the project.

5. Conclusion and future work

Every day, software systems are becoming more powerful and complex, resulting in the necessity of using new approaches for their development and management. Most of these approaches have emerged as a result of the large breakthroughs of last decades in different fields of Information and Communication Technologies. One of these approaches is the microservice-oriented approach, which is becoming popular among developers because of the benefits of using microservices. In particular, this thesis has studied the microservice-oriented approach with the purpose of creating a new way for developing the applications and tools of the future. Moreover, this thesis has proposed an innovative platform for designing, developing and deploying microservices in order to provide new technologies that allow the development of powerful and complex software systems.

Despite the large potential of the current platform, this thesis believes that the platform has not reached its full potential yet. Thus, this thesis proposes several aspects as future work in order to enhance the platform. First, the platform may include a monitoring system in order to know the status of all the deployed microservices in real time. Moreover, this monitoring system may also be able to apply some predefined actions automatically according to the status in order to be able to react instantaneously against failures or attacks. Second, the platform may offer the possibility of enabling a transparent cache in order to improve the overall performance of the microservices. Third, the platform may provide some mechanism to simplify and automate the testing of the microservices in order to allow developers to verify systematically their systems. Fourth, the platform may support and handle more communications protocols transparently in order to enhance the interaction between microservices and allow the integration with legacy systems. On the one hand, this thesis suggests implementing the MQTT and AMQP protocols in order to support the publish-subscribe pattern. On the other hand, this thesis also suggests implementing the second version of HTTP protocol in order to obtain the latest improvements of the protocol. Fifth, The platform may develop new core services and improve the current ones in order to provide better out-of-the-box features for developers. On the one hand, the platform may increase the number of authentication and authorization protocols in order to enhance the security of the microservices. On the other hand, the platform may improve the service directory implementing some persistence mechanism with the purpose of being able to recover the registered microservices after a failure or reboot. Sixth, the platform may support the use of several content types with the purpose of facilitating the development of rich microservice-oriented systems. Likewise, the platform may negotiate the content type with the clients as REST architectural style suggests. Seventh, the platform may verify automatically the body of the incoming HTTP requests and outgoing HTTP responses with the service description in order to detect possible anomalies. Last, the platform may develop a client library for external systems in order to facilitate the interaction between these external systems and the deployed microservices.

In conclusion, Microrestjs is an innovative platform that aims to be the basis of the future technologies, thus contributing to The Law of Accelerating Returns.

Bibliography

[1] Raymond Kurzweil. (2001, March) The Law of Accelerating Returns. [Online]. http://www.kurzweilai.net/the-law-of-accelerating-returns

[2] Tim Urban. (2015, January) The AI Revolution: The Road to Superintelligence. [Online]. http://waitbutwhy.com/2015/01/artificial-intelligence-revolution-1.html

[3] Yefim V. Natis. (2003, April) Service-Oriented Architecture Scenario. [Online]. https://www.gartner.com/doc/391595/serviceoriented-architecture-scenario

[4] IBM. Service Oriented Architecture Glossary. [Online]. http://www-01.ibm.com/software/solutions/soa/glossary/index.html

[5] OASIS Standard. (2006, October) Reference Model for Service Oriented Architecture 1.0. [Online]. http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf

[6] Thomas Erl, *Service-Oriented Architecture: Concepts, Technology, and Design*.: Prentice Hall, 2005.

[7] W3C. (2004, February) Web Services Architecture. [Online]. http://www.w3.org/TR/ws-arch/

[8] Roy T. Fielding and Richard N. Taylor. (2000) Principled Design of the Modern Web Architecture. [Online]. https://www.ics.uci.edu/~fielding/pubs/webarch_icse2000.pdf

[9] T. Berners-Lee. (1996, August) WWW: Past, present, and future. [Online]. https://www.w3.org/People/Berners-Lee/1996/ppf.html

[10] Roy T. Fielding et al. (1999, June) Hypertext Transfer Protocol - HTTP/1.1. [Online]. https://tools.ietf.org/html/rfc2616

[11] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. (2005, January) Uniform Resource Identifier (URI): Generic Syntax. [Online]. https://tools.ietf.org/html/rfc3986

[12] Ned Freed, Murray Kucherawy, Mark Baker, and Bjoern Hoehrmann. (2015, September) Media Types. [Online]. http://www.iana.org/assignments/media-types/media-types.xhtml

[13] Marc Hadley. (2009, August) Web Application Description Language. [Online]. http://www.w3.org/Submission/wadl/

[14] James Lewis and Martin Fowler. (2014, March) Microservices. [Online]. http://martinfowler.com/articles/microservices.html

[15] National Institute of Standards and Technology. (2015, September) Recommendation for Key Management - Part 1: General. [Online]. http://csrc.nist.gov/publications/drafts/800-57/sp800-57p1r4_draft.pdf

[16] Agence nationale de la sécurité des systèmes d'information. (2014, February) Référentiel Général de Sécurité B1 - Règles et recommandations concernant le choix et le dimensionnement des mécanismes cryptographiques. [Online]. http://www.ssi.gouv.fr/uploads/2015/01/RGS_v-2-0_B1.pdf

[17] Bundesamt für Sicherheit in der Informationstechnik. (2015, February) Kryptographische Verfahren: Empfehlungen und Schlüssellängen. [Online]. https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/TechnischeRichtlinien/TR02102/BSI-TR-02102.pdf?__blob=publicationFile

[18] J. Franks et al. (1999, June) HTTP Authentication: Basic and Digest Access Authentication. [Online]. https://tools.ietf.org/html/rfc2617

[19] Dick Hardt. (2012, October) The OAuth 2.0 Authorization Framework. [Online]. http://tools.ietf.org/html/rfc6749

[20] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and C. Mortimore. (2014, November) OpenID Connect Core 1.0. [Online]. http://openid.net/specs/openid-connect-core-1_0.html

[21] C. Neuman, T. Yu, S. Hartman, and K. Raeburn. (2005, July) The Kerberos Network Authentication Service (V5). [Online]. https://tools.ietf.org/html/rfc4120

[22] Markus Hillenbrand, Joachim Götze, Jochen Müller, and Paul Müller, "A Single Sign-On Framework for Web-Services-based Distributed Applications," in *Proceedings of the 8th International Conference on Telecommunications ConTEL*, Zagreb, June 2005. [Online]. http://www.researchgate.net/profile/Markus_Hillenbrand/publication/4151783_A_single_sign-on_framework_for_web-services-based_distributed_applications/links/0fcfd508e58dec71db000000.pdf

A. Acronyms

AMQP	Advanced Message Queuing Protocol
ANSSI	Agence Nationale de la Sécurité des Systèmes d'Information
API	Application Programming Interface
BDD	Behavior Driven Development
BSI	Bundesamt für Sicherheit in der Informationstechnik
CPU	Central Processing Unit
CSS	Cascading Style Sheets
DAC	Discretionary Access Control
DH	Diffie-Hellman
ECDH	Elliptic Curve Diffie-Hellman
ECMA	European Computer Manufacturers Association
FS	File System
GPL	GNU General Public License
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
IANA	Internet Assigned Numbers Authority
IBM	International Business Machines Corporation
IO	Input/Output
IoC	Inversion of Control
IP	Internet Protocol
IT	Information Technology
JS	JavaScript
JSCS	JavaScript Code Style
JSON	JavaScript Object Notation
JSON-WSP	JavaScript Object Notation Web-Service Protocol

MAC	Mandatory Access Control
MIT	Massachusetts Institute of Technology
MPL	Mozilla Public License
MQTT	Message Queuing Telemetry Transport
NIST	National Institute of Standards and Technology
NPM	Node.js Package Manager
OASIS	Organization for the Advancement of Structured Information Standards
PKI	Public Key Infrastructure
POODLE	Padding Oracle On Downgraded Legacy Encryption
RAML	RESTful API Modeling Language
RBAC	Role-based Access Control
REST	Representational State Transfer
RFC	Request for Comments
SHA	Secure Hash Algorithm
SOA	Service-oriented Architecture
SOAP	Simple Object Access Protocol
SPDX	Software Package Data Exchange
SSL	Secure Sockets Layer
TLS	Transport Layer Security
UDDI	Universal Description, Discovery and Integration
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WADL	Web Application Description Language
WSD	Web Service Description
WSDL	Web Service Definition Language
XML	Extensible Markup Language

B. Microrestjs Service Description Specification

Microrestjs Service Description Specification is a JSON specification that aims to describe and document microservices. This specification defines the structure of service descriptions, the format of each field, some constraints to avoid future problems, and some recommendations and good practices to guarantee a good design and documentation.

B.1. Main object

```
{
     "microrestSpecification": 1,
     "info": <Information Object>,
     "config": <Configuration Object>,
     "security": <Security Object>,
     "operations": <Operations Object>
}
```

Field Name	Type	Description
microrestSpecification	integer	**Required.** Specifies the version of Microrestjs Service Description Specification used to describe the service. This value must be 1.
info	Information Object	**Required.** Provides meta-information about the service.
config	Configuration Object	**Required.** Provides some necessary configuration properties to run the service.
security	Security Object	**Required.** Specifies the security of the service.
operations	Operations Object	**Required.** Defines the service operations. The service must have at least one operation.

B.2. Information object

```
{
     "name": <string: organization-service>,
     "version": <string: MAJOR.MINOR.PATCH>,
     "api": <integer>,
     "description": <string>,
     "keywords": [string],
     "date": <string: YYYY-MM-DD>,
     "authors": [Person Object],
     "contributors": [Person Object],
     "homepage": <string: URL>,
     "repository": <Repository Object>,
     "bugs": <Bugs Object>,
     "termsOfService": <string: URL>,
     "license": <string: SPDX Format>
}
```

Field Name	Type	Description
name	string	**Required.** Specifies the name of the service. It must be unique in the world and should describe the service. Therefore, it is highly recommendable to follow the format: organization-service OR product-service. Only alphanumeric characters, underscores and dashes are accepted.

Field Name	Type	Description
version	string	**Required.** Specifies the version of the service. The format of the version must be MAJOR.MINOR.PATCH and should follow the Semantic Versioning 2.0.0[1] definition where an increment in • MAJOR indicates that at least one service operation or the API has been modified and therefore the backwards-compatibility has been broken. • MINOR indicates that at least one new service operation has been added without changing the API of the previous implemented operations. The backwards-compatibility is assured. • PATCH indicates that an already implemented service operation has been fixed or re-implemented without modifying its API. The backwards-compatibility is assured.
api	integer	**Required.** Specifies the number of the service API. It must be a positive integer. An increase of this number means that the service API has been changed. Therefore, the API number must be increased when the service has MAJOR or MINOR changes.
description	string	Describes the service using natural language.
keywords	[string]	Specifies the list of keywords that characterize the service.
date	string	Specifies the release date of the current service version. The date format must be YYYY-MM-DD as the definition of full-date in RFC3339[2].
authors	[Person Object]	Provides information about the author/s of the service.
contributors	[Person Object]	Provide information about the contributor/s of the service.

[1] http://semver.org/

[2] http://xml2rfc.ietf.org/public/rfc/html/rfc3339.html#anchor14

Field Name	Type	Description
homepage	string	Specifies the URL to the official website of the service. The URL should follow the format: <http or https>://<hostname or IP>:<port>/<path>
repository	Repository Object	Specifies the public repository that manages the code of the service.
bugs	Bugs Object	Provides information to report service issues.
termsOfService	string	Specifies the URL to the Terms of Service webpage. The URL should follow the format: <http or https>://<hostname or IP>:<port>/<path>
license	string	Specifies the license/s of the service. Format: • If the license is part of the SPDX licenses, the SPDX identifier should be included. E.g. "GPL-3.0". • If the license has not been assigned a SPDX identifier or if the license is a custom one, the format should be "LicenseREF-LICENSE". And a LICENSE file should be included. • If more than one license is used, SPDX expressions should be used. E.g. "(MIT OR MPL-1.0)". For more information, please check the official specification of SPDX[1].

[1] http://spdx.org/

B.3. Person object

```
{
     "name": <string>,
     "email": <string: EMAIL>,
     "url": <string: URL>
}
```

Field Name	Type	Description
name	string	**Required** if included**.** Specifies the name of the person.
email	string	Specifies the email of the person. The email should follow the format: <local-part>@<hostname or IP>
url	string	Specifies the URL to the webpage or social profile of the person. The URL should follow the format: <http or https>://<hostname or IP>:<port>/<path>

B.4. Repository object

```
{
     "type": <string>,
     "url": <string: URL>,
}
```

Field Name	Type	Description
type	string	**Required** if included. Specifies the type of the code repository. E.g. “git”, “svn”, …
url	string	**Required** if included. Specifies the URL to the code repository. The URL should follow the format: <http or https>://<hostname or IP>:<port>/<path>

B.5. Bugs object

```
{
     "url": <string: URL>,
     "email": <string: EMAIL>,
}
```

Field Name	Type	Description
url	string	**Required** if included[1]. Specifies the URL to the project's issue tracker. The URL should follow the format: <http or https>://<hostname or IP>:<port>/<path>
email	string	**Required** if included[1]. Specifies the email address to which issues should be reported. The email should follow the format: <local-part>@<hostname or IP>

[1]At least one of the fields is required if this object is included.

B.6. Configuration object

```
{
     "location": <string: "directory" or URL>,
     "dependencies": <Dependencies Object>
}
```

Field Name	Type	Description
location	string	**Required.** Specifies the location of the service. If the service is dynamic and does not have a fixed URL, this field must be filled with: • "directory" to register the service in the default directory. • or "directory://<hostname or IP>:<port>/path" to register the service in an external directory. Otherwise, if the service is static and will be always deployed on the same URL, this field can also be filled with that URL. In that case, the URL must follow the format: https://<hostname or IP>:<port>/<path>
dependencies	Dependencies Object	Specifies the external services that are necessary for the correct operation of this service.

B.7. Dependencies object

```
“<serviceName>”: {
     "api": <integer>,
     "url": <string: "directory" or URL>
}
```

Field Name	Type	Description
serviceName	string	**Required** if included. Specifies the name of the external service. Only alphanumeric characters, underscores and dashes are accepted.
api	integer	**Required** if included. Specifies the API number of the external service.
url	string	**Required** if included. Specifies the location of the external service. If the external service has been registered in a directory, this field must be filled with: • “directory” if the external service has been registered in the directory defined by default. • or “directory://<hostname or IP>:<port>/path” if the external service has been registered in an external directory Otherwise, if the external service has not been registered in a directory and is always deployed on the same URL, this field can also be filled with that URL. In that case, the URL must follow the format: https://<hostname or IP>:<port>/<path>

B.8. Security Object

```
{
     "scheme": <string: "none" or "basic">
}
```

Field Name	Type	Description
scheme	string	**Required** if included. Specifies the security authorization scheme. If the scheme is defined as “none”, no authorization scheme is considered to authorize the execution of the service operations. If the scheme is defined as “basic”, the HTTP basic authorization scheme is considered to authorize the execution of the service operations.

B.9. Operations object

```
"<operationName>": {
     "security": <Security Object>,
     "request": <Request Object>,
     "responses": <Responses Object>,
     "errors": <Errors Object>
}
```

Field Name	Type	Description
operationName	string	**Required**. Specifies the name of the operation. This name must be equal to the name of the function that executes the operation. This name must be unique and therefore no other operation of this service can have the same name. Moreover, it is highly recommendable to write the operation name using camelCase format. Only alphanumeric characters are accepted and the first one must be an alphabetic character.
security	Security Object	Specifies the security of the operation. If it is not defined, the security of the operation will be the same as the security of the service.
request	Request Object	**Required**. Defines the properties of the HTTP request that executes this operation.

Field Name	Type	Description
responses	Responses Object	**Required**. Defines all the possible HTTP responses that the operation can reply. The operation must have at least one response.
errors	Errors Object	Defines all the possible errors and exceptions that the operation can reply.

B.10. Request object

```
{
     "path": <string: URL Path>
     "method": <string: HTTP Method>,
     "parameters": <Parameters Object>,
     "body": <Body Object>
}
```

Field Name	Type	Description
path	string	**Required**. Specifies the URL path where the requests must be sent to execute the operation. The path must follow the format: • /<path> • or /:<pathParameter> • or any combination of the previous ones **Important:** the pair (path, method) must be unique and therefore no other operation of this service can have the same path and method.

Field Name	Type	Description
method	string	**Required**. Specifies the method HTTP that is required to execute the operation. This field must be filled with one of the following methods: • "GET" • "HEAD" • "POST" • "PUT" • "DELETE" • "PATCH" **Important:** the pair (path, method) must be unique and therefore no other operation of this service can have the same path and method.
parameters	Parameters Object	Defines the query or path parameters of the request.
body	Body Object	Defines the body objects that must be sent in the request to execute correctly the operation. The definition of the body forces to send at least the defined objects correctly. In any case, other not-defined objects can be sent but their correctness will not be possible to check automatically.

B.11. Responses object

```
"<responseName>": {
     "statusCode": <integer>,
     "description": <string>,
     "body": <Body Object>
}
```

Field Name	Type	Description
responseName	string	**Required**. Specifies the name of this response. The response name must be unique and therefore no other response or error of this operation can have the same name. Furthermore, it is highly recommendable to use the name of the status code. E.g. "OK" if status code is 200, "ACCEPTED" if status code is 202, … Only alphanumeric characters, underscores and dashes are accepted.
statusCode	integer	**Required**. Specifies the status code of this response. The status code must follow the HTTP Status Codes[1] specifications.
description	string	Specifies the meaning of this response and the reasons to reply this response.
body	Body Object	Defines the body objects that are included in the response. The definition of the body forces to send at least the defined objects correctly. In any case, other not-defined objects can be sent but their correctness will not be possible to check automatically.

[1] http://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml

B.12. Errors object

```
"<errorName>": {
     "statusCode": <integer>,
     "description": <string>,
     "body": <Body Object>
}
```

Field Name	Type	Description
errorName	string	**Required** if included. Specifies the name of this error response. The error name must be unique and therefore no other response or error of this operation can have the same name. Furthermore, it is highly recommendable to use the name of the status code. E.g. "UNAUTHORIZED" if status code is 401, "NOT FOUND" if status code is 404, … Only alphanumeric characters, underscores and dashes are accepted.
statusCode	integer	**Required** if included. Specifies the status code of this error response. The status code must follow the HTTP Status Codes[1] specifications.
description	string	Specifies the meaning of this error response and the reasons to reply this error response.
body	Body Object	Defines the body objects that are included in the error response. The definition of the body forces to send at least the defined objects correctly. In any case, other not-defined objects can be sent but their correctness will not be possible to check automatically.

[1] http://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml

B.13. Parameters object

```
"<parameterName>": {
     "in": <string: "query" or "path">,
     "description": <string>,
     "required": <boolean>,
     "type": <string>,
}
```

Field Name	Type	Description
parameterName	string	**Required** if included. Specifies the name of the parameter. This name must be unique and therefore no other parameter of this operation can have the same name. Moreover, it is highly recommendable to write the operation name using camelCase format. Only alphanumeric characters are accepted and the first one must be an alphabetic character.
in	string	**Required** if included. Specifies if the parameter is a query parameter or a path parameter. Therefore, this field must be filled with “query” or “path”.
description	string	Specifies the meaning of the parameter and the accepted values.
required	boolean	**Required** if included. Specifies if the parameter is either required (true) or optional (false).
type	string	**Required** if included. Specifies the type of the parameter. This field can be filled with one of the following types: • “string” • “integer” • “number” • “boolean”

B.14. Body object

```
"<objectName>": {
     "type": <string>,
     "items": <string>,
     "description": <string>,
     "properties": <Body Object>,
     "required": <boolean>
}
```

Field Name	Type	Description
objectName	string	**Required** if included. Specifies the name of the object.
type	string	**Required** if included. Specifies the type of the object. This field can be filled with one of the following types: • "object" • "string" • "integer" • "number" • "boolean" • "array"
items	string	**Required** if type is "array". Specifies the type of array items. This field can be filled with one of the following types: • "object" • "string" • "integer" • "number" • "boolean"
description	string	Specifies the meaning of the object and how to fill it.
properties	Body Object	**Required** if type is "object". In this case, specifies the properties of the object. Each property is considered as a body object. This means that properties are like sub-objects and therefore can contain other sub-properties. **Required** if type is "array" and items is "object". In this case, specifies the kind of objects that the array can contain. Therefore, one and only one property can be specified in order to avoid the mixture of different object types.
required	boolean	**Required** if included. Specifies if the object is either required (true) or optional (false).

B.15. Body object reference

```
{
     "$ref": <JSON Reference>,
     "required": <boolean>
}
```

Field Name	Type	Description
$ref	JSON Reference	**Required** if included. Set a JSON reference to another object defined in the service description file. This is useful to reuse and avoid the duplication of objects.
required	boolean	Specifies if the object is either required (true) or optional (false). When it is specified, the value *required* defined in the referenced object is overridden.

C. Microrestjs Framework Configuration

The Microrestjs Framework can be configured through a configuration file. This configuration file uses JSON in order to configure easily certain aspects of the framework. The configuration file allows configuring the following aspects:

```
{
     "services": {
        "path": <string>
     },

     "server": {
        "port": <integer>
     },

     "directory": {
        "location": <string: URL>
     },

     "logger": {
        "enable": <boolean>,
        "level": <string>
     }
}
```

Field Name	Type	Description
path	string	**Required.** Specifies the path that contains the services to be deployed.
port	integer	**Required.** Specifies the port in which the server should listen for new connections. The port value must be between 0 and 65535.
location	string	**Required.** Specifies the URL of the default service directory.
enable	boolean	**Required.** Specifies if the logs of the framework should be enabled or disabled.
level	string	**Required.** Specifies the logging level. This field must be filled with one of the following values: “error”, “warn”, “info”, “verbose”, “debug”, or “silly”.