Generation of Web API Definition Files, using a single platform, accordingly to the Design First Approach

TIAGO EMANUEL ANDRADE DE CARVALHO
Outubro de 2021

Instituto Superior de Engenharia do Porto

Generation of Web API Definition Files, using a single platform, accordingly to the Design First Approach

Tiago Carvalho

A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Science, Specialisation Area of Computer Systems

Supervisor: Dr. Paulo Gandra de Sousa

Porto, October 15, 2021

Dedicatory

To my girlfriend Flávia without whom this dissertation would have been completed, with some luck, during 2025.

Abstract

With the increasing usage of the Internet, the development of Web Application Programming Interface (API)s became one of the most important areas of software development. Along the years, how this development was made, how the security was approached and how much time was spent in the project outside of the development part changed. One of these changes was the introduction of Web API definition files, that specify how clients and servers communicate and what resources are provided. With this, different methodologies of work to create clients and servers appeared. The approaches studied in this dissertation were Code First Approach, were the development happens before the generation of these files, and Design First Approach, were the definition files are created before the development of the software.

The objective of this dissertation is the study of a new solution to the creation of Web API definition files, that follows the Design First Approach, that are aided by a graphic interface, and a server, where definitions files can be saved, associated with projects. These can have multiple files associated.

To better understand how the solution defined in this dissertation brings new ideas and a different approach to the Design First Approach, multiple solutions were analyzed, along with the possible solution that responds to the objective of this dissertation. During this analysis, the functional and non functional requirements were defined, along with the use cases and data model, for the development of a REpresentational State Transfer (REST) API based solution. With this analysis, a prototype was created and tested, following a set of tests defined.

The defined tests were executed in a controlled environment, but don't completely reflect a real situation, were multiple requests are made by users. The tests still revealed that the prototype can generate correct specifications, to be used in the development of servers and clients and these can be correctly saved in the server, without taking much time to respond to the client (less than 400 ms on the tests, acceptable in the context of the test conditions).

It was concluded that the solution has high potential to be an alternative to the current solutions in the market, bringing the idea of projects and server-side file saving and a better functionality of the web site, thanks to its graphic interface that shows what content is needed in a specification, reducing errors on the creation of specifications.

Keywords: REST, Specification, Web API, Design First Development, Functionality

Resumo

Com o aumento do uso da Internet, o desenvolvimento de interfaces de programação de aplicações para web (Web APIs) tornou-se uma das áreas mais importantes do desenvolvimento de software. Ao longo dos anos, como é que este desenvolvimento foi realizado, como se abordou a segurança necessária e quanto tempo é utilizado para o desenvolvimento e para o planeamento destas aplicações foi mudando. Uma destas alterações foi a introdução de ficheiros de definição de web APIs, que permitem especificar como é que clientes e servidores comunicam e que recursos estão disponíveis. Com estes ficheiros, diferentes metodologias foram aparecendo para criar clientes e servidores. As abordagens estudadas nesta dissertação são a abordagem de código em primeiro lugar, onde o desenvolvimento das aplicações acontece antes de se gerarem ficheiros que definam a API, e a abordagem de *design* em primeiro lugar, onde os ficheiros que definem a API são criados primeiro, e o código de clientes e servidores é desenvolvido segundo estes ficheiros.

O objetivo desta dissertação é o estudo de uma nova solução para a criação de ficheiros de definição de Web APIs, que siga os princípios da abordagem de *design* em primeiro lugar, que tenha um cliente que utilize uma interface gráfica para gerar ficheiros e um servidor, onde ficheiros de definições possam ser guardados, associados a projetos. Estes projetos podem conter múltiplos ficheiros, de diferentes especificações.

Para melhor entender como é que a solução definida nesta dissertação traz novas ideias e uma diferente forma de seguir a abordagem de *design* em primeiro lugar, múltiplas soluções já existentes foram analisadas, juntamente com a possível solução que responda aos objetivos desta dissertação. Durante esta análise, os requisitos funcionais e não funcionais foram definidos, juntamente com os casos de uso e o modelo de dados, para o desenvolvimento de uma solução baseada numa API REST. Através desta análise, um protótipo foi criado e testado, segundo um conjunto definido de testes.

Os testes definidos foram executados num ambiente controlado, mas que não reflete completamente uma situação real, onde múltiplos pedidos são feitos pelos utilizadores. Mesmo nestas condições, os testes realizados revelaram que o protótipo consegue gerar especificações corretas, para serem utilizadas no desenvolvimento de servidores e clientes. Estes também conseguem ser guardados no servidor, com um tempo de resposta ao cliente inferior a 400 ms, o que é aceitável tendo em conta as condições do teste.

Conclui-se que a solução tem um potencial elevado para ser uma alternativa às soluções atualmente encontradas no mercado, com a capacidade de salvar ficheiros num servidor e permitir um maior acesso a estes, seja por parte de um único utilizador ou de equipas e funcionalidades melhoradas, graças à utilização da interface gráfica, que permite reduzir erros na criação de especificações através do conteúdo apresentado ao utilizador.

Contents

List of Figures

List of Tables

List of Source Code

List of Acronyms

API	Application Programming Interface.
CSS	Cascading Style Sheets.
CVS	Control Version System.
FEI	Front End Of Innovation.
HTML	Hypertext Markup Language.
HTTP	HyperText Transfer Protocol.
HTTPS	HyperText Transfer Protocol Secure.
IDE	Integrated Development Environment.
JS	JavaScript.
JSON	JavaScript Object Notation.
NCD	New Concept Development.
RAM	Random Access Memory.
REST	REpresentational State Transfer.
SOAP	Simple Object Access Protocol.
SQL	Structured Query Language.
WLAN	Wireless Local Area Network.
WSDL	Web Services Description Language.
XML	eXtensive Markup Language.
YAML	YAML Ain't Markup Language.

Chapter 1

Introduction

It is pretended, with this dissertation, to study how Web Application Programming Interface (API) definitions (REpresentational State Transfer (REST), Simple Object Access Protocol (SOAP) or others) can be developed and presented to the user and the logic of this type of platform.

In this first introductory chapter, the dissertation context, the problem and the expected objectives will be presented and it will be finished with an explanation of the document structure.

1.1 Context

Web API Development is one of the main areas of software development and is increasing its usage, following the increasing usage of the internet. Development of Web APIs can use different specification approaches. These can be created using SOAP protocol, REST architecture or other specification approach.

A Web API that follows a defined form can be translated into a definition file, that follows a defined syntax and is capable of being interpreted. These files can be generated manually, using a Design First Approach, or they can be generated from code, using a Code First Approach.

1.2 Problem

To create Web APIs two different approaches are used when developing these. Code First approach and Design First Approach. The Design First Approach defines the development as a stream where the service is defined as text before starting to code.

The problem with the current solutions for the Design First Approach is that they are almost always for a single definition (ex: Swagger only defines REST APIs) or are defined as long blocks of text, with easy ways to create errors, and very time consuming.

It is pretended that the solution to be defined can, accordingly to the Design First approach, create REST API definition files (YAML Ain't Markup Language (YAML)) and SOAP Web Services Description Language (WSDL) files, through the usage of a website, using usable interfaces that don't rely on written text to create the definition files.

1.3 Objective

Web API Development using the Design First Approach, puts the most important part of the project in the service definition and artifacts creation. These artifacts are definition files that can be used by development teams to implement servers and clients, that communicate, at the same time.

The objective of this dissertation is to define a **new solution to the creation of Web API definition files, using a graphic interface to create these definitions and to give the possibility to create multiple definition files for the same project**. The solution must be more intuitive than the existing in the market and less costly, while maintaining the base functionalities of those solutions.

The proposed prototype, GraphicAPIDefinition, will try to respond to this objective and the analysis made.

1.4 Structure

This document is divided in six chapters: Context, State of the Art, Solution Analysis and Design, Solution Implementation, Tests and Results and Conclusions and Future Work. In the context of the curricular unit, in which this dissertation is inserted, it is expected to answer the following set of outcomes:

- Outcome 1 – Problem Interpretation;
- Outcome 2 – Context and State of the Art;
- Outcome 3 – Evaluate Existing Solutions and Approaches;
- Outcome 4 – Solution Design;
- Outcome 5 – Solution Implementation;
- Outcome 6 – Solution Evaluation;

In chapter 2, identified as Context, *Outcome 1: Problem Interpretation* and part of *Outcome 2: Context and State of the Art* will be elaborated. In this chapter, will be presented the problem study and the impacts of this problem. In the end, will be presented the objective of this dissertation, that will guide the rest of the document.

In chapter 3, identified as State of the Art, part of *Outcome 2: Context and State of the Art* and *3: Evaluate Existing Solutions and Approaches* will be elaborated. In this chapter, it will be explained the current state of the Web API Development Approaches, the protocols and Architectures that are used, their popularity and software that is used to create definition files that respects them. For the Web API Development Approaches and the software used to implement this, an analysis of how they work, their advantages and disadvantages will be presented, comparing the current solutions in the market.

In chapter 4, identified as Solution Analysis and Design, *Outcome 4: Solution Design* will be elaborated. In this chapter, will be presented the Analysis of a solution that can respond to the problem presented in Chapter 5, and the design of it. The analysis includes the functional and non functional requirements for the solution and the design includes the architecture of the solution. In this chapter, diagrams are presented to define the analysis and design of the solution, and will serve as a base for the implementation of a prototype.

In chapter 5, identified as Solution Implementation, *Outcome 5: Solution Implementation* will be elaborated. In this chapter, the technical details and implementation, based on the previous chapter, will be explained.

In chapter 6, identified as Tests and Results, part of *Outcome 6: Solution Evaluation* will be elaborates. In this chapter, the tests to realize will be defined and the results obtained will be evaluated, accordingly to the problem defined.

In chapter 7, identified as Conclusions and Future Work, part of *Outcome 6: Solution Evaluation* will be elaborated. In this chapter, the final evaluation of the dissertation will be explained, analyzing the method followed and the decisions taken during the elaboration of this dissertation. In the end, it will be presented the future work possible for this dissertation, accordingly to the taken conclusions.

Chapter 2

Context

In the last chapter (1) the problem was identified in a simplistic form. In this chapter, where part of the **Outcome 2: Context and State of the Art** will be approached, a detailed study about Web API definitions and the development process of these. A value analysis will be presented for this dissertation.

In this chapter, the following questions will be answered:

What are the details of the proposed problem?

What approaches to Web API development exists?

What are the advantages and disadvantages of the different approaches when developing a Web API?

What artifacts need to be generated during the development process?

2.1 Problem Detail

Web API Development has increased in the last years, as the internet got more accessible and affordable to users and companies worldwide. For the development of a Web API, there must be a choice taken for the approach to take. During the analysis phase, it is taken the decision to follow one of the two approaches to API development.

While Code First Approach is already established, with many tools to generate artifacts, the Design First Approach, while having different advantages than those of the Code First Approach, has less tools to aid developers during the development process. These tools are, in some parts, hard to use, making the development process more difficult, prone to errors and time consuming, removing the advantages that define this approach. These problems are present independently of the protocol or architecture used, with different platforms having different problems.

To respond to the problems existent in the Design First Approach, it is needed a solution that can create artifacts, which isn't prone to errors and is easier to use, for different protocols and architectures, while maintaining the same base functionalities as the existing solutions in the market.

2.2 Web API Development Approaches

Web API development have two main approaches: Code First Approach and Design First Approach. In this section, both approaches will be presented, along with the advantages and disadvantages associated with them.

2.2.1 Code First Approach

The Code First approach relies in coding the solution first and generate the definition files after everything is defined in the code.

This approach has advantages over development speed, synchronization and rapidly generated artifacts, reducing the time taken in the analysis phase.

This creates problems: clients can't be developed at the same time (In different organizations), maintenance of code is increasingly difficult, code annotations aren't always well defined or even compatible with generating software and can create problems when developers leave the organization (Karanam 2019a).

Advantages

- Fast Service Development

  The main advantage for using the Code First Approach is the capability to deliver a single solution fast.

- Low-effort Artifacts

  Artifacts can be easily generated from the source code, using multiple annotations.

- Artifact and Code stay synchronized

  Since generating artifacts can be done easily, these are also synchronized with the code, maintaining a correct state of artifacts.

Disadvantages

- No Parallel Development

  Parallel Development isn't easy to happen when artifacts needed to the development of the other part (be it the client or the server) only are generated after the code is created.

- No Target for Teams

  Different teams don't have the artifacts that can define an API, until these are generated from the original code and shared. This can increase the development times of the other applications and can also create many errors and desynchronization between all parts.

- No Cross-Platform Compatibility

  This is a problem that appears when using frameworks that don't support artifact generation, since the definition artifacts must be written by hand, increasing the time of development. When the framework used has the capability to generate the definition artifact, it can introduce language constraints, increasing the difficult to use different languages for the clients and servers.

2.2.2 Design First Approach

To solve the disadvantages of Code First approach there is the Design First approach where, after the requirement definition, the developers take time to create a Web API definition, that can be maintained more easily than code and can be used to generate code. These can be used to develop both server-side and client-side applications, in parallel, reducing the time of development and are less prone to error (Karanam 2019b). The main disadvantages is the higher effort in the analysis phase and the time taken across the project.

Advantages

- Parallel development

  Parallel development is possible, since the artifacts that define the APIs to be used are already defined and development can be made both for servers and clients.

- Target for teams

  Different teams have all the artifacts and can code, knowing the full requests expected. In case of desynchronization because of the different development speeds, this can be mitigated with the usage of mock requests or mock responses.

- Reuse of artifacts

  If different parts of a service are to be reused in other services, the artifacts can also be reused, reducing development costs for those services.

- Cross-platform compatibility

  Since the definition artifacts are written first, the languages and frameworks used to develop the service don't matter as much, even if they can't generate the service structure automatically. This also facilitates the use of different languages in clients and servers, since the definition artifact is agnostic.

- Reduced costs of development

  Creating artifacts first, while more time consuming in the phases prior to the development, reduces the time taken to define and redefine the services and permits artifact and code reuse across different projects.

- Reduce risk of failure

  Defining the API first gives the business context and is more adaptable to changes and, since the specification is shared between all teams, the development is more streamlined.

Disadvantages

- High effort in the initial phase of the project

  One of the main problems in the development of Web APIs using the Design First Approach is the high effort that is needed in the first phases of the project, both planning and analyzing, because of the need to define the API Definition. This effort is mitigated by the parallel development and the clear target for all involved.

2.3 Web API Definition Artifacts

One of the most important parts of Web API development is the generation of Artifacts that define the services available to be consumed and that can provide the service. In this section, both the definition files and the service files will be explained.

2.3.1 Web API Definition Files

Web services are defined in files, corresponding to its specification type. These can be generated in the code (Code First Approach), or written before coding and generate the code after them (Design First Approach). These follow a specific syntax for its type and must be shared across the development teams that will implement the servers and clients, since these correspond to a contract between all parties and define all methods available in the servers.

Syntax

Artifact Syntax is a set of rules and principles that define the correct form to structure an Artifact definition. In general, the syntax of a Web API definition file contains the methods that are available, the inputs that each method takes, the outputs that are returned from the method (both success and failure outputs) and the endpoint where the service can be accessed.

Other possible options in syntax, dependent on the type of the specification, can be Hyper-Text Transfer Protocol (HTTP) status code returns, method, entity and service descriptions and the content type of the payload of a request, among others.

2.3.2 Web API Service Files

Web services are provided in their compiled files. These provide the services defined in the definitions files, with its corresponding endpoints. These are the last files to generate in the development process.

2.4 Summary

In this chapter, the context of the problem was presented, highlighting the different approaches to Web API development and the advantages and disadvantages of these.

It was presented, in a general form, the Web API definition files and their syntax and why they are needed in the development process.

What are the details of the proposed problem?

The proposed problem is defined by the necessity to create better tools for the Design First Approach, that is, in theory, a better solution to the development of Web APIs.

What approaches to Web API development exists?

There are two main approaches to the Web API development, the Code First Approach and the Design First Approach.

What are the advantages and disadvantages of the different approaches when developing a Web API?

Both approaches have advantages and disadvantages to its use in the development process of Web APIs, with the Design First Approach having more and better advantages. While this is true, there are situations where Code First Approach can be used in a more advantageous way.

What artifacts need to be generated during the development process?

During the development process there are two types of artifacts that need to be generated. The first is the service definition file, that stipulates all the methods that are defined in the service. The second is the compiled executable file, that will provide the service.

Chapter 3

Value Analysis

In the last chapter (2) the problem was defined in a detailed form, along with context of Web API Development. In this chapter, where part of the **Outcome 2: Context and State of the Art** will be approached, a detailed value analysis about this study will be presented for this dissertation.

In this chapter, the following questions will be answered:

What steps were taken to find the value of the work of this dissertation?

What is the value of the work of this dissertation for the Web API Design First Approach?

3.1 New Concept Development (NCD) Model

The NCD model defines a common language between the different components of the Front End Of Innovation (FEI), where five different key elements are integrated. The five key elements are opportunity identification, opportunity analysis, ideas generation, ideas selection and concept definition.

The interaction between these elements, the possible starting points for defining a new idea and the possible output points are defined in Figure 3.1.

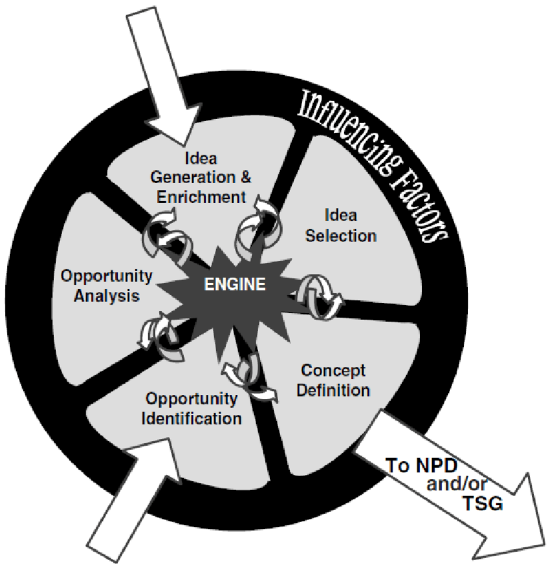


Figure 3.1: NCD Model

3.1.1 Key Elements

The five key elements define the NCD Model. In this sub-chapter, the five elements will be explained and how they contributed to the work of this dissertation.

Opportunity Identification

As explained in the beginning of this chapter, and in chapter 2, there is a need for better solutions when designing new Web APIs, following the Design First Approach, since there are few to chose from, but there is real advantages over the Code First Approach.

This need is translated into an opportunity to develop a new solution for this type of approach to Web API development, using a better user interface, capable of reducing human error, and a possibility to use multiple API Definitions on the same project.

Opportunity Analysis

This opportunity analysis was made in an area where development works within the same parameters, normally proposing a new technology or improving the features already in existence. After studying the theoretic concepts and the implementations of these concepts, it was verified the need to create a new tool to help defining Web APIs.

Single solutions exist, but are rudimentary or costly, while the proposed system is multi solution and defines a new form to interact with users.

Ideas Generation

The ideas generation occurred after the opportunity analysis and an analysis to the development process of the Design First Approach. This analysis, the previous knowledge acquired in academic and professional environments, and various meeting with the supervisor, helped giving a better understanding of the theme and a more refined set of ideas.

From this analysis, the following set of ideas were defined:

- Developing a solution where Web APIs can be specified;
- Multiple specification protocols and architectures must be available to users;
- Developing a solution where a logged user can create projects, with different specifications inside;
- The developed platform must be user friendly;
- The developed platform must reduce the probability of errors;
- The developed platform must permit the import and export of definition files;
- The solution must be secure.

Ideas Selection

The ideas were selected based on the requirements needed, the direct and indirect benefits and implementation costs. This was made both for a final solution and the proposed prototype, giving each idea a priority in the development process.

Concept Definition

The concept definition is made after the ideas are selected. The prototype *GraphicAPIDefinition* was defined as a tool to help development teams, in a more efficient way, define Web APIs, reduce development time, reduce development costs and better work with other development teams.

3.1.2 External Factors

External factors are all the factors that can affect a business and aren't in the control of the a business. It is important to evaluate the impact of different outside factors, so that the business model can be defined accordingly, and the business doesn't fail, or loses an opportunity, because of this.

The most important factors to follow for this idea are the economic conditions, the technological advancements and political and legal forces.

Economic conditions are important both in the country where the business is based, but also when it depends on external companies, since the economic conditions of those countries can affect prices for the business. In this case, it could affect the price of hosting servers and ad placement. On the other hand, taxation from the country where the business is based must be followed, since it will impact the expenses of the business.

Technology advancements are both an opportunity and a risk for the business. Every time a new Web API definition protocol or architecture is created, it creates an opportunity to add it to the product, increasing the number of solutions it offers. The changes in server hosting,hosting and development technologies are important to look at, since these could bring increase in speed and security. On the other side, new platforms that can define Web APIs are a threat to the business, pushing the product to be always innovating.

Political and legal forces are important, because they define the law (or ensure that the law is followed). A change in the government normally equates to a change to the fiscal, business and workers laws, and a business needs to prepare itself for these changes. For now, the government is stable and it isn't expected that the laws change will affect the business.

3.2 Perceived Value

Value is a key part of a business and it depends on exchanging something tangible or intangible good or service and being accepted and rewarded by costumers or clients (Nicola and Ferreira 2012).

From this definition can be inferred that value is difficult to create and define, since it depends on the customers and how they perceive a product or service. A good way to understand value is looking at the relations between benefits and sacrifices perceived by the customer.

As shown in Table 3.1, there is a high value for customers with this solution, since it resolves problems that exist in existing solutions, but it has a small market, since this kind of solutions is important to developers of Web APIs.

Table 3.1: Cost / Benefits relationship

	Product	Relationship
Benefits	Development of a multidefinition solution Development of a graphic user interface	Better user experience Reduced number of errors
Costs	Cost of Server Cost of Development Team	Small market

3.3 Value Proposal

The proposed solution promotes a better interface to developers, that can reduce errors while giving a better understanding of the new service, and a better user experience while creating it. It also provides the possibility to create multiple definitions, without restriction of the protocol or architecture used, inside the same project. This is translated in its value proposition, *The easy way to define your web services*.

3.3.1 Network Value

This project has the possibility to create a network value between the owner of the project, the partners of the project and Web API developers.

The value for developers is already defined in the value proposal. While it doesn't grant monetary values directly, it generates value in their time and experience, that can translate into different values when they apply the gained value in other activities.

The value for partners and the owner is mostly monetary, but it also has a recognition value, especially for the owner of the project.

Developers can also bring another type of value to the project, the feedback, through the support area. With this, it is possible to increase the value for the developers and to the other members of the network.

3.3.2 FAST Technique

The most important value for a user is the capability of generating Web API definition files. Using the FAST technique it is possible to analyze the process to generate a Web API definition file, as shown in Figure 3.2. This figure shows how the process to generate a file occurs. The functions Define a service, Generate definition file, Validate specification and Download website content (after choosing a specification type) are functions that result from user actions, while the others involves user intervention.

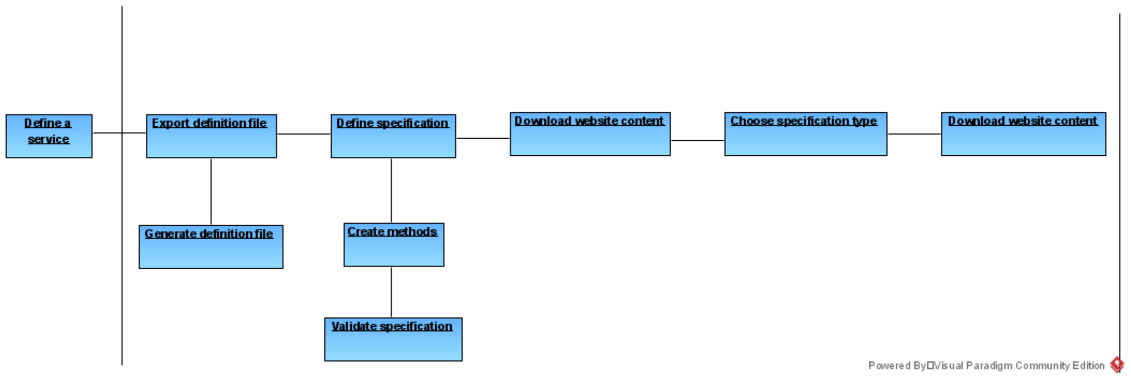


Figure 3.2: FAST Diagram for generating Web API definition files

3.3.3 Business Model Canvas

The value proposed and all the parts that influence the value can be shown in a business model canvas. For the proposed solution, the Figure 3.3 shows its business model canvas.

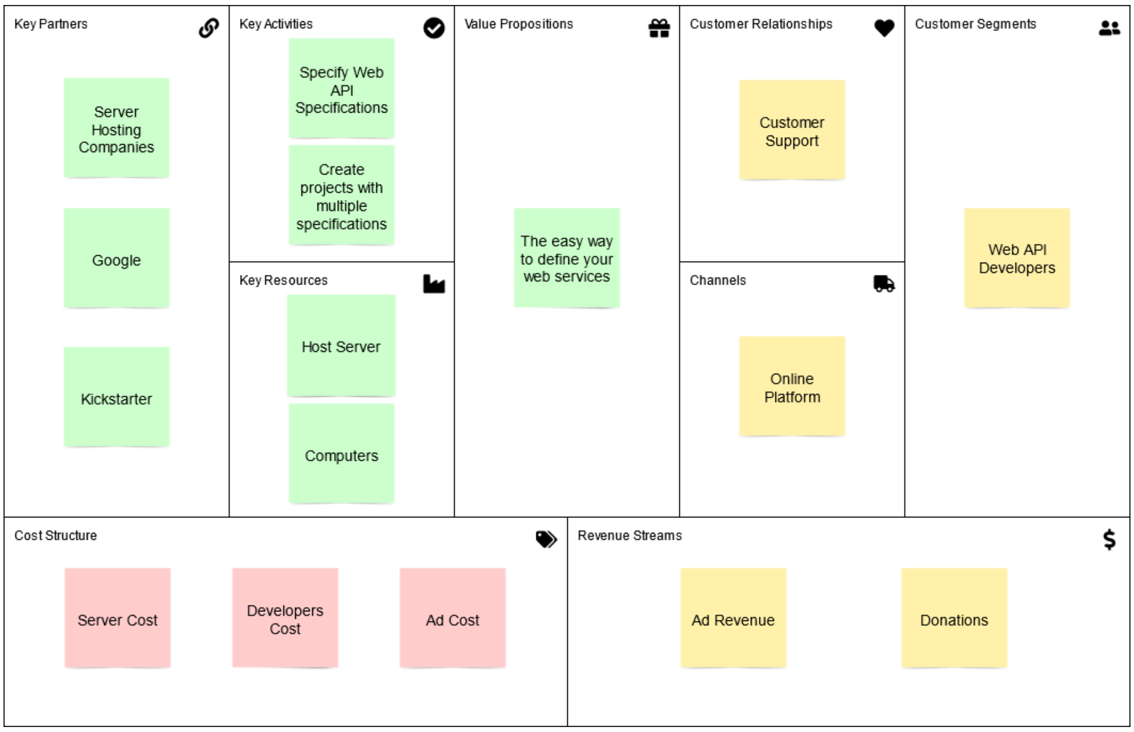


Figure 3.3: Business Model Canvas for GraphicAPIDefinition

3.4 Summary

In this chapter, the value analysis for the proposed project was presented with the aim to present the real value of the project. It was verified that the possible users of the solution are Web API developers, that receive value as a service provided to them and they can generate value for the partners and the owner of the solution.

The following questions were answered:

What steps were taken to find the value of the work of this dissertation?

This project came from the identification of an opportunity to create new tools to define Web APIs, following the Design First Approach. The generated ideas follow this opportunity, with the possibility to define Web APIs using different protocols or architectures being the most important, followed by the possibility to create a project with different specifications.

After analyzing the different factors, benefits and sacrifices to be made, a business model was defined.

What is the value of the work of this dissertation for the Web API Design First Approach?

As detailed in the chapter 2, the problem of the Design First Approach is the scarcity of tools and the deficiency of the user user experience in the existing ones. The work of this dissertation has the objective to mitigate these problems, adding a new tool, that has multiple specification types and a good user experience.

Chapter 4

State of the Art

In the last chapter (3) the value of the proposed solution was analyzed in a detailed form. In this chapter, where part of the **Outcome 2: Context and State of the Art** and the **Outcome 3: Evaluate Existing Solutions and Approaches** will be approached, a detailed analysis of the current state of the art for the Design First Approach will be presented. After this, the current solutions will be analyzed and compared.

In this chapter, the following questions will be answered:

What are the major protocols and architecture to define services?

What solutions exists that follows the Web API Design First Approach?

4.1 WEB API Specification Types

Web API Services can be defined using protocols or architectures. These define a set of request methods, with a specific name, inputs and outputs, in structured files that define the service. While other protocols and architectures exist, the protocols and architecture presented are the most dominant ones in the market.

4.1.1 Protocol

Protocol, in computer science, is defined as a standardized set of rules, for processing and transfer data, based on simplistic principles. Protocols are agnostic of languages and technologies, and different implementations have the same behavior (*Protocol* 2019).

HTTP

HTTP is the main protocol of data transmission on the internet, because of its simplicity of usage and standardization. It defines a set of methods for communication between two endpoints. The main methods of HTTP permits the create (POST), read (GET), update (PUT) and delete (DELETE) operations across the internet.

This protocol is not commonly used alone and is normally associated with another protocol or architecture.

SOAP

Simple Object Access Protocol is a protocol for communication between two hosts, represented in a eXtensive Markup Language (XML) format and doesn't depend on any lower-level protocols, but is mainly used over HTTP (Box et al. 2000).

It was designed to facilitate communication between clients and servers, independent of the development language, thanks to the use of XML. Today, SOAP is entering in a phase of decline, because of the popularity of the REST Architecture.

4.1.2 Architecture

Architecture, in computer science, is defined as the conceptual structure of a designed system, and defines a set of rules to describe the capabilities of a system, but do not define its implementation (*Architecture* 2012).

REST

Representational State Transfer is a Software Architecture, built on top of HTTP methods, that defines a series of standards for HTTP communication between two hosts, one working as client and the second working as a server.

The REST specification doesn't define methods, but requests for resources and doesn't define a structure for requests, with multiple options, as XML or JavaScript Object Notation (JSON), which gave REST its popularity (*What is REST?* 2021).

4.2 Existing Solutions

In this section, different solutions to define Web APIs will be analyzed, each one with two different technologies. This study has the objective of identifying functionalities, approaches and technologies used to define Web APIs.

The selected projects for this study respect, at least, one of the following criteria: cost, Web API Definition Files creation capacity, usage difficulty and Web API definition maintenance. When analyzing existing solutions, none of the found solutions could permit multiple specification type, focusing on one of them.

4.2.1 Swagger Project

The Swagger project, developed by Tony Tam, now owned by SmartBear Software, and the basis for the OpenAPI Specification, is based on the idea that RESTful API specifications should be standardized to facilitate the general communication between multiple companies. It uses a specific set of keywords, key components and structured user inputs to define an API.

While the definition files can be written in a web editor, its usage is prone to errors and is hard to use when the definition starts to grow. SmartBear Software also created the SwaggerHub, which brings the functionalities of team collaboration, file saving in the cloud, peer review and mock servers, but still has the same verbose editor as the web version (*Fast Simple API Development | SwaggerHub* 2021).

4.2.2 Visual Paradigm

Visual Paradigm is a software for modeling and designing solutions, not exclusive for software, that relies in a graphic interface and user experience.

Visual Paradigm incorporates the definition of REST APIs in their class diagram, using the REST Resource object, where each object defines a method from a resource. The designing of Web APIs is possible in the free (non-commercial) version, but the possibility to generate artifact files or have team collaboration is only available on paid plans (*How to Design REST API with UML?* 2021).

4.2.3 Eclipse Integrated Development Environment (IDE)

The Eclipse IDE is an environment to develop software, with the capability to create SOAP definition files (WSDL files) using a graphic or text interface.

This solution is free and has the possibility to validate WSDL files and these can be integrated in working projects. The support of Control Version System (CVS), like GIT, gives the possibility of team collaboration (*Introduction to the WSDL Editor* 2008).

4.2.4 Altova XMLSpy

The Altova solution, XMLSpy, is a XML and JSON editor, with the possibility to create and edit multiple XML-type files, in a graphic or text interface.

This solution incorporates the WSDL tools in the Enterprise plan, and its graphic editor resembles that of the Eclipse IDE. The tools include WSDL creation and validation, and integrates a client and debugger for requests (*XMLSpy Highlights* 2021).

4.2.5 Existing Solution Evaluation

To evaluate solutions, it is needed to define a set of variables to analyze, that can give a global picture of how they compare with each other. For the solutions given, the analyzed variables are the Web API definition writing capability, Web API definition saving capabilities, Web API definition file generation capability, team capability, cost and user experience.

In terms of writing capability, the solutions that stand out are the SOAP editors, Eclipse IDE and Altova XMLSpy. They have graphic and textual editors capable of fully defining a service. The Visual Paradigm solution has the most problems, trying to integrate graphic and textual parts inside the same solution, inside a Class Diagram.

In terms of saving capability, every editor permit file saving in the local disk and only the web editor of Swagger didn't have the cloud saving capabilities. The REST solutions have full cloud integration, while the SOAP solutions only have support for CVSs.

In terms of file generating capability, all presented solutions permit the generation of definition files with their basic plan, except Visual Paradigm, that needs a paid plan.

In terms of team capability, the web version of swagger is the only solution that doesn't support it in any form. The SwaggerHub has paid plans for teams and enterprises, that allows this capability. Eclipse IDE and Altova XMLSpy supports it, through the CVSs. Visual Paradigm partially supports this in the free version and has all the capabilities available in the paid version.

In terms of cost, only the web version of Swagger and Eclipse IDE are completely free to use, and Swagger Hub has a free plan for 1 user and paid plans for teams and enterprises. Visual Paradigm is free for non-commercial use and has paid plans for commercial use. Altova

XMLSpy only has one paid plan (capable of service definition) and is the most costly of the existing solutions.

In terms of user experience, the analyzed existing solutions follow the same pattern of the writing capability, where the easiest editors to use, as a user are the SOAP editors with their graphic editor. Then it is the Swagger solutions, with a mixture of a text editor and graphic view. The hardest to use is the Visual Paradigm solution, that incorporates a graphic editor that takes time to understand and use it to its full capacity.

The comparison of the analyzed variables is shown in Table 4.1

Table 4.1: Existing solutions comparison

	Swagger Web Editor	Swagger Hub	Visual Paradigm	Altova XMLSpy (Enterprise)	Eclipse WSDL Editor
Web API definition writing capability	Yes No graphic editor	Yes No graphic editor	Graphic definition Specific parts are written	Graphic editor Text editor	Graphic Editor Text Editor
Save Web API definition	Local Files	Local Files Cloud	Local Files Cloud	Local Files Cloud integration	Local Files Cloud integration
Web API definition file generation	Yes	Yes	Paid version	Yes	Yes
Team Functionalities	No	Paid Version	Partial in free version Full in paid versions	Support	Support
User Experience	Verbose editor Graphic view of API Hard to use in big definitions	Verbose editor Graphic view of API Hard to use in big definitions	Graphic editor Hard to define Easy to understand the API	Graphic editor Easy to define	Graphic Editor Easy to Define
Cost	Free	Free for 1 user Paid plan for teams and enterprises	Free (Non-commercial) Paid plan	Paid plan only	Free

4.3 Summary

In this chapter the technologies available and different solutions in the market, for the Design First Approach, were presented. Of the solutions presented, none of them respond to all the requirements of the objectives of this dissertation. But, some of them have different functionalities outside of this dissertation, like the team integration functionalities.

The following questions were answered:

What are the major protocols and architecture to define services?

The main protocol to define services in use today is the SOAP protocol and the main architecture is the REST architecture. Today, REST architecture is the most used way to define services, with developers choosing it over the SOAP protocol.

What solutions exists that follows the Web API Design First Approach?

There are some solutions capable of responding to the needs of Design First Approach, like the Swagger solutions and Visual Paradigm for REST APIs and Eclipse IDE and Altova XMLSpy for SOAP. Of all the analyzed solutions, is the Eclipse IDE that better responds to the objectives of this dissertation but, as observed in the analysis, fails to fully respond to them.

Chapter 5

Solution Analysis and Design

In this chapter, the approach to the development of a solution to the defined problem will be explained, giving a presentation of the functional and non-functional requirements and the domain model of the proposed application. The solution will be specified accordingly to the best approach, based in the realized studies and low costs.

In this chapter, the **Outcome 4: Solution Design** will be approached. In the end of this chapter, the following questions will be answered:

What is the proposed solution for the studied problem?

What possible alternatives could be used?

5.1 Analysis

In this section the proposed solution to create a platform that can create Web API definition files will be explained. This analysis has as an objective to prove the possibility to create projects with multiple Web API specification files, using a graphic interface.

The proposed solution is based on the Design-first approach to the development of Web APIs, which first defines the API, and uses that same definition to start the server and the clients development. The proposed solution has the possibility to define APIs in two defined structures, REST architecture and SOAP Protocol. The proposed solution will have the capability to export the files to the user computer, import files from it and create projects with multiple API definitions (the project capability is accessibly only to registered users). The registered users capabilities are provided by a REST server, with the capability to scale and to be redundant. The web site to be implemented, that will serve as the point of interaction with the user, will use a graphic interface. The approach to follow will be divided in three parts, the web site, the REST server and the database, which will be explained in this chapter.

After analyzing and defining what is pretended for the development of the solution and its prototype, it is needed to define the functional and non functional requirements and the use cases of the system.

5.1.1 Functional Requirements

Functional Requirements are the descriptions of the proposed functionalities to be implemented in the developed platform.

For the proposed solution, the following functional requirements are defined:

- Creation of REST Web API Specification;
- Creation of SOAP Web API Specification;
- Creation of Projects;
- Export Web API Specification;
- Import Web API Specification;
- Create User Account.

5.1.2 Non Functional Requirements

Non Functional Requirements are the descriptions of how the platform should be implemented, including user interfaces, security constraints, architectures to be used, among others.

For the proposed solution, the following non functional requirements are defined:

- A graphic Web interface must be defined;
- The Web site and Web API used must be reliable and accurate;
- The Web API must be maintainable and easy to scale;
- The Web API must be implemented using REST architecture;
- The system must be secure, with the usage of HyperText Transfer Protocol Secure (HTTPS) and others mechanisms.

5.1.3 Use Cases

Defined the requirements of this solution, it is necessary to compile these into a use case diagram, a diagram that represents requirements in a graphic view. The following diagram, shown in Figure 5.1, represents the use cases that the proposed solution will try to achieve and what responsibilities each actor will have.

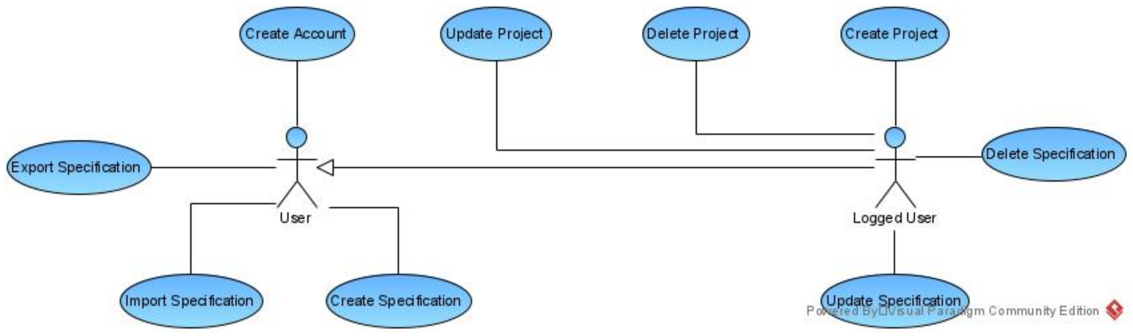


Figure 5.1: Use Case Diagram.

Create Specification

The main aspect of the proposed solution is the creation of the specification files. In the case of the proposed solution, the specifications (REST Specification, SOAP Specification or HTTP Specification) are to be created using a graphic interface. This use case is associated with both actors, with the difference that a Logged User can create a specification inside a project.

Create Project

A Logged User can create a project, with more than one specification associated with it.

5.1.4 Domain Model

In the domain model, shown in Figure 5.2, shows the interaction between the users, the projects and the specifications.

After a user is logged in the system, he has access to the list of its projects, and he can modify them. The projects have a list of specifications associated, that the user can access and modify.

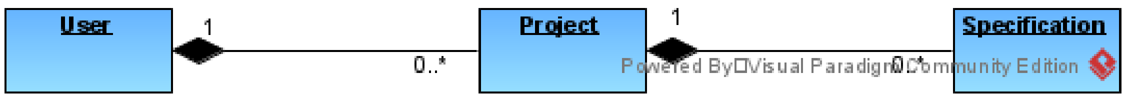


Figure 5.2: Domain Model

5.2 Design

In this section, the components diagram and the data model of the project will be presented.

5.2.1 Components Diagram

In the components diagram of the proposed solution, shown in Figure 5.3, the three main components of the system are represented and how they interact with each other.

The first component, the website, will be the component that interacts with the users. All the needed requests to the server will be made using a REST API.

The second component, the server, will receive requests from the website, process them and respond to the website. It will communicate with a database through a Structured Query Language (SQL) Connector and with the computer through the file system.

The third component, the database, will communicate with the server only and is responsible to save all the information from the user and the projects created, except the definition files.

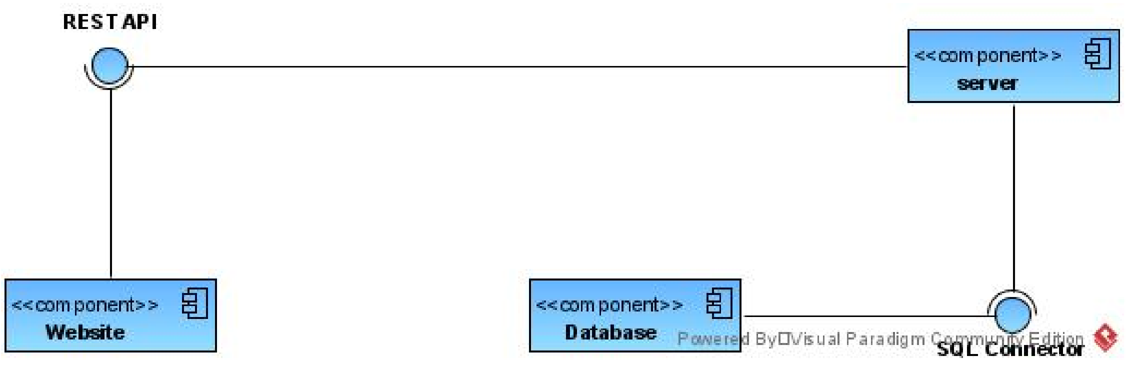


Figure 5.3: Solution Components Diagram.

5.2.2 Data Model

The following diagram, shown in Figure 5.4, represents the proposed data model for the solution that is proposed

A Registered User can have multiple projects, and each project can have multiple specifications. A specification has a specification type associated (REST, SOAP or other). A generated token is saved for all logged users, with an expiration date to use that token.

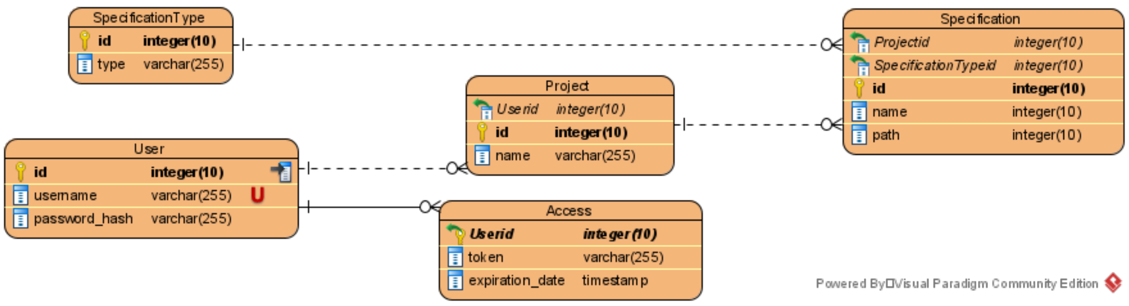


Figure 5.4: Data Model

5.3 Alternatives

In this section, different alternatives to the presented solution will be analyzed. The first alternative is a possible desktop application.

5.3.1 Desktop Application

The first alternative would be a possible desktop application, where a user could create projects and specifications. This software could be made standalone, losing the feature to work anywhere, or work alongside the server and the website, having the possibility to work on the desktop application or the website. The advantage of this alternative is being able to work on projects that are localized in the personal computer, when access to the internet is not possible. The disadvantage for the users is that local projects are not accessible by internet and for a developer team, the increased costs to develop a desktop app along the other parts of the solution.

5.4 Summary

In this chapter, the proposed solution was described in terms of its requirements and design, giving a better understanding of a possible solution to the problem referred in chapter 2.

The following questions were answered:

What is the proposed solution for the studied problem?

The proposed solution to resolve the problems defined in this dissertation involves a website, capable of creating Web API specifications, with a graphic user interface that is user friendly, and a server, that can receive requests from the website, that saves logged users projects and specifications. The website must have the capability to import and export Web API specifications from the user computer.

What possible alternatives could be used?

Any solution that can respond to the objectives of this dissertation could be a viable alternative. A possible alternative was using a desktop application, standalone or with connection to a server. A standalone application would have a big disadvantage in terms of accessing projects and specifications from other devices.

Chapter 6

Solution Implementation

In the last chapter (5) the design of the proposed solution was analyzed in a detailed form. In this chapter, where the **Outcome 5: Solution Implementation** will be approached, the implementation of the solution will be presented.

In this chapter, the following question will be answered:

The implementation is compliant with the presented solution?

How was implemented the front-end

How was implemented the back-end

6.1 Front-end Application

In this section, the front-end of the implemented prototype will be explained, along with the decisions taken for this approach.

Since the proposed solution has a non functional requirement to use a graphic web interface as the front-end, the development of the solution was made using Hypertext Markup Language (HTML), JavaScript (JS) and Cascading Style Sheets (CSS). Since the problem refers to the functional aspect of the web page, and not its graphic design, the CSS files only support the website as a form to understand where all components may stay and what are their differences.

The prototype was implemented for the REST specification generation only, because of its wide usage.

6.1.1 Dynamic Web Page

When defining a specification, independent of the architecture or protocol associated, there isn't a set number of requests or components to be specified. Since there is no limit, a web page that creates its content dynamically is necessary. This was achieved with JS files, that handle the dynamic parts of the web page.

Static Content

The static content of the website is composed of the elements that define a specification, the save file buttons and the dynamic generation area for entities and requests, as shown in Figure 6.1.

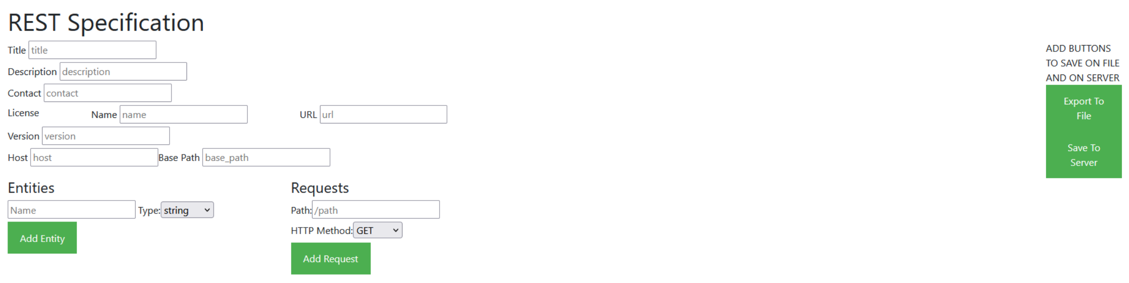


Figure 6.1: Static Content to Define a Specification

Dynamic Content

These files construct the web page accordingly to the user commands from two different parts, the entities, where the REST components are created, and the requests, where the REST requests are created, as shown in Figure 6.2.

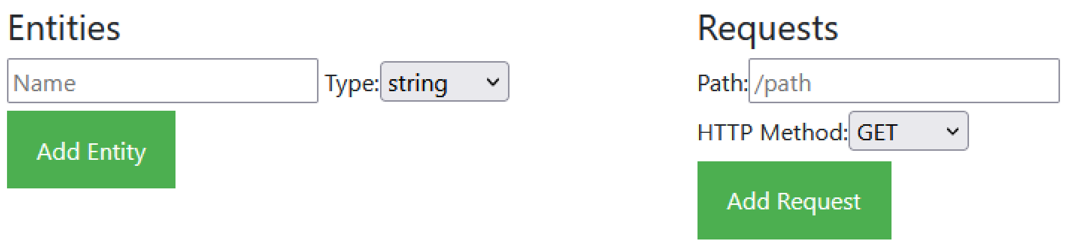


Figure 6.2: Static Content to Generate Entities and Requests

The Entity area button *Add Entity* generates an entity based on its name and its type. For complex type *object*, as shown in Figure 6.3, an area to add the type properties is added.

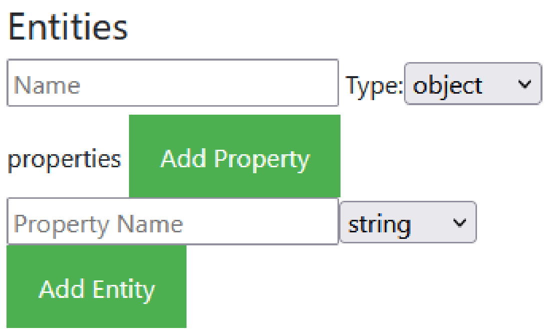


Figure 6.3: Static Content to Generate Entitiy Properties

For complex type *array*, as shown in Figure 6.4, an area to choose the type of the items is added. The list is updated every time an entity is added to the specification, along with the original types.

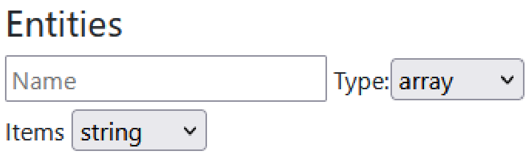
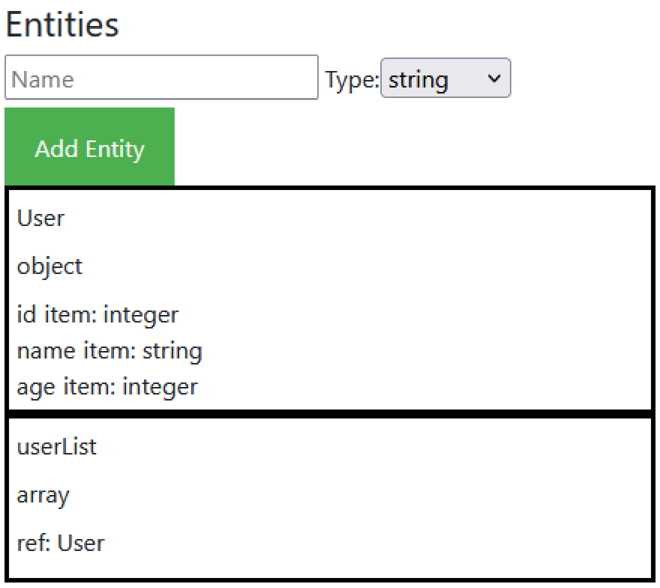


Figure 6.4: Static Content to Generate an Entity Array

The entities created are then saved and shown in the web page as shown in Figure 6.5.



User

object

id item: integer
name item: string
age item: integer

userList

array

ref: User

Figure 6.5: Generated Entities

The Request area button *Add Request* generates requests, with different HTTP methods grouped by their paths, as shown in Figure 6.6, where the paths are delimited by a black box. In the same figure, it is possible to see the path parameter generated from the path, that is separated from the HTTP methods, since it is the same for all of them.

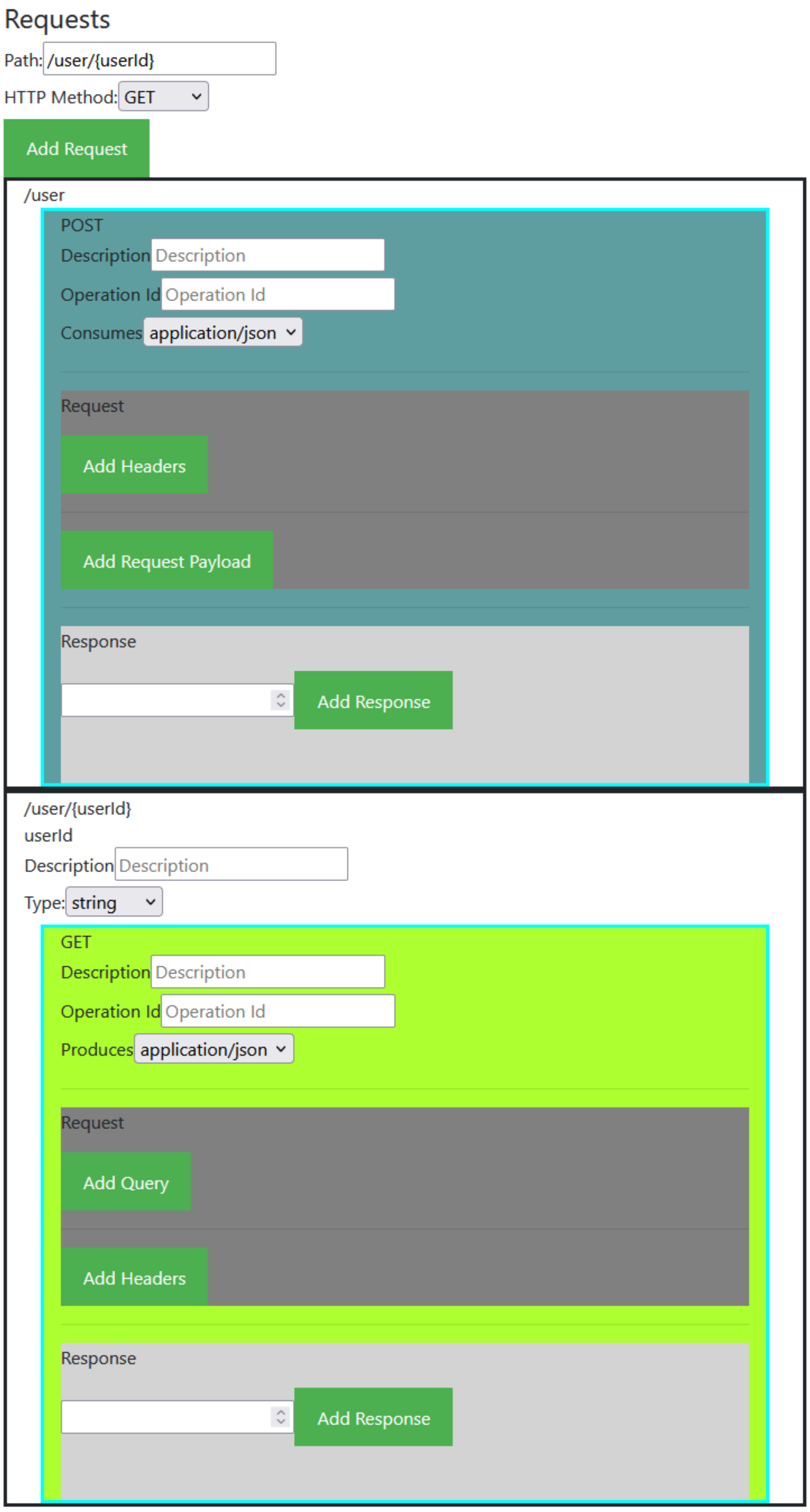

Requests
Path: /user/{userId}
HTTP Method: GET
Add Request
/user
POST
Description: Description
Operation Id: Operation Id
Consumes: application/json
Request
Add Headers
Add Request Payload
Response
Add Response
/user/{userId}
userId
Description: Description
Type: string
GET
Description: Description
Operation Id: Operation Id
Produces: application/json
Request
Add Query
Add Headers
Response
Add Response

Figure 6.6: Generated Requests

Each method definition is dependent on its type, only showing the possible configurations for each. This means the method GET is the only method that can have a query string and a response payload defined, while the methods POST, PUT and PATCH have a request payload, but no query string or response payload. The method delete does not have any of these configuration, and, besides the path parameters, it only has the capability to add headers to the request and responses.

Both the query string configuration and the Header configurations were coded with the same HTML parts, except for the type of parameter, that exists only on the query string, as shown in Figure 6.7. The other possible values for each are the parameter name, description and if it is required.

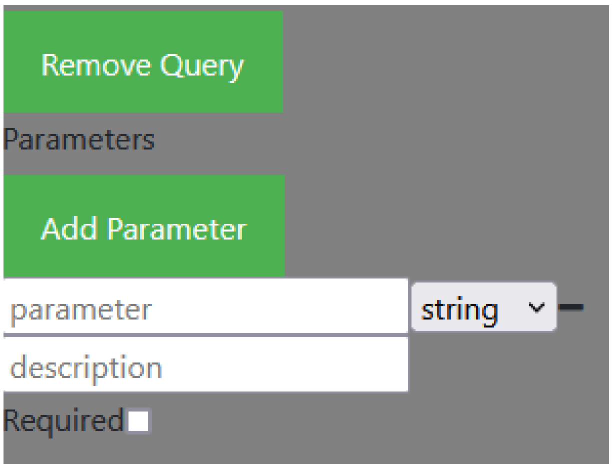


Figure 6.7: Query String definition

As said before, the methods responsible for creating or updating values on a server have a request payload as shown in Figure 6.8. This is composed of an entity, its descriptions and if it is required.

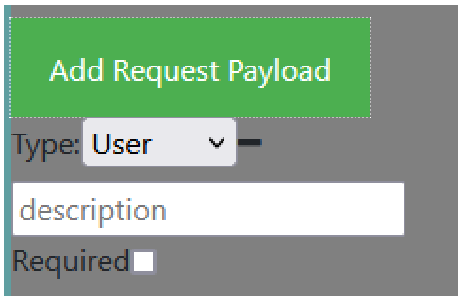


Figure 6.8: Request Payload definition

The last configurable part of a method are the responses. These are added using the button *Add Response*, as seen in Figure 6.6, after providing a response status code on the input box. The generated response configuration can be seen in Figure 6.9. The area corresponding to a specific status code response can be switched, by choosing the different codes inserted. The header fields are implemented following the same logic of the request headers and the response payload, while being unique for the GET method, has a select box, where an entity can be chosen.

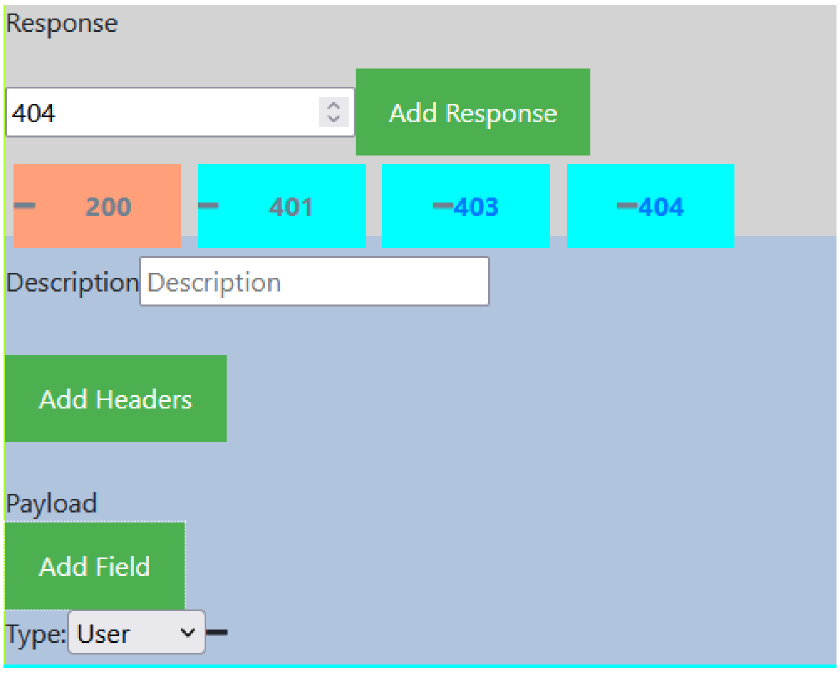


Figure 6.9: Response definition

6.1.2 Generate Specification

The main function of the website is to permit the user to create and generate a specification. This is achieved through the both contents, explained before. After creating the specification, this needs to be saved into the user disk or the server.

The file is generated using the same method, independently of the destination of the file, and uses ids that were generated when adding requests to retrieve the correct information, as shown in Listing 6.1.

```
 1 function generateContent() {
 2 ...
 3   let requests = document.getElementById("request_list").childNodes;
 4 
 5   for (let i = 0; i < requests.length; i++) {
 6     let node = requests[i];
 7 
 8     let request_nodes = node.childNodes;
 9 
10     let request = "";
11     let parameter_paths = "";
12     for (let j = 0; j < request_nodes.length; j++) {
13       if (request_nodes[j].tagName === "DIV") {
14         if (request_nodes[j].className.includes("row")) {
15           let id = request_nodes[j].firstElementChild.id;
16 
17           let request_type = document.getElementById(id + "_label").
    innerText.toLowerCase();
18           let request_description = document.getElementById(id + "
    _description_input");
19           let request_operationId = document.getElementById(id + "
    _operation_input");
20           ...
21         }
22       }
23     }
24   }
25 }
```

Listing 6.1: Retrieving a request

As shown in line **15**, the id of the specific HTTP request method is retrieved, and used to access the inputs required or areas with inputs required (Query String, Headers and Payload areas). At these ares, a generic id is used to retrieve information from the web page, as shown in Listing 6.2.

```
 1 let queries = document.getElementById(id + "_query");
 2 if (queries) {
 3     parameters = "      parameters:\n";
 4 
 5     queryList = queries.querySelectorAll("div");
 6 
 7     for (let k = 0; k < queryList.length; k++) {
 8         let query = queryList[k].querySelector('*[id=parameter]').value;
 9 
10         let query_description = queryList[k].querySelector('*[id=
    description]').value;
11         let query_required = queryList[k].querySelector('*[id=required]'
    );
12         let query_type = queryList[k].querySelector('*[id=select]').
    value;
13 
14         parameters += "       - name: " + query + "\n         in: query\n"
15             + "         description: " + query_description + "\n
    required: ";
16 
17         if (query_required.checked) {
18             parameters += "true\n";
19         } else {
20             parameters += "false\n";
21         }
22 
23         parameters += "         schema:\n           type: " + query_type +
     "\n";
24     }
25 
26 }
```

Listing 6.2: Retrieving the query string of a request

After retrieving all the queries of a request, in line **1**, and verifying the existence of them (in line **2**), the queries are retrieved using generic ids (lines **8** through **12**), associated with the query in question and saved in the content of the file in line **23**. The headers, request payload and response payload work in the same format.

The entities are retrieved in a similar form, taking advantage of a list of entities created, as shown in Listing 6.3.

```
1  let entity_list = document.getElementById("entities").querySelectorAll("*[id=entity]");
2  
3  for (let i = 0; i < entity_list.length; i++) {
4  ...
5      let name = entity_list[i].querySelector("*[id=entity_name]").textContent;
6      let type = entity_list[i].querySelector("*[id=entity_type]").textContent;
7  
8      let entity_special_items = "";
9      if (type === "object") {
10         let property_list = document.getElementById("entity_" + name + "_properties").childNodes;
11 
12         entity_special_items += "      properties:\n"
13         for (let j = 0; j < property_list.length; j++) {
14         ...
15         }
16     }
17     if (type === "array") {
18         let ref_item = document.getElementById("entity_" + name + "_array_item_ref");
19         if (!ref_item) {
20             ref_item = document.getElementById("entity_" + name + "_array_item_type");
21             entity_special_items += "      items:\n        type: " + ref_item.textContent.split(": ")[1] + "\n";
22         } else {
23             entity_special_items += "      items:\n        $ref: '#/components/schemas/" + ref_item.textContent.split(": ")[1] + "'\n";
24         }
25     }
26 
27     components += "    " + name + ":\n      type: " + type + "\n" + entity_special_items;
28 }
29 
30 return spec + paths + components;
```

Listing 6.3: Retrieving the entities of a specification

After retrieving all entities from the web page (line **1**), each entity is saved into the components string (line **21**), along with its type, and in case its type is an object, its properties (lines **9** through **15**), or in case its type is an array, the type of the array is also saved, as a reference (line **23**) or as a predefined type (lines **19** through **21**). As shown in line **30**, the string generated is returned to the method which called for the generated file.

6.1.3 Save Specification

After generating a specification, it is needed to be saved. To achieve this, two functions were defined. The first one saves the specification in the user's computer after generating it, as shown in Listing 6.4.

```
 1 function exportToFile() {
 2     let content = generateContent();
 3     if (content !== "") {
 4 
 5         let a = document.createElement('a');
 6         let file = new Blob([content], { type: 'text/yaml' });
 7 
 8         a.href = URL.createObjectURL(file);
 9         a.download = "specification.yaml";
10         a.click();
11 
12         URL.revokeObjectURL(a.href);
13     } else {
14         /*...*/
15     }
16 }
```

Listing 6.4: Save specification to computer

After generating the specification (line **2**), it is created a hyperlink element in the document (line **5**), that is used to save the file in the computer (line **10**). In the end, the URL object is revoked (line **12**).

The second function saves the specification in the server, after generating it, as shown in Listing 6.5.

```
 1 async function saveToServer() {
 2     let content = generateContent();
 3     if (content !== "") {
 4         let payload = { name: $('#title_input').val(), specificationType
     : 'rest', file: content };
 5         const response = await fetch('...', {
 6             method: 'POST',
 7             body: payload, // string or object
 8             //...
 9         });
10         const myJson = await response.json(); //extract JSON from the
     http response
11     }
12 }
```

Listing 6.5: Save specification to server

Like in the first function, the specification is generated to be saved as a file (line **2**). The text is then sent to the server in line **5**, with the name of the file and the specification type and the project id in the url part of the fetch method. The response is then read and processed in line **10**.

6.2 Back-end Application

In this section, the back-end of the implemented prototype will be explained, along with the decisions taken for this approach.

The language chosen to develop the back-end application was JS, with the node.js runtime, along with a database created using PostgreSQL. The server supports the website, managing user projects and specifications, with create, update, read and delete operations for projects, specifications and users, following, according to the non functional requirements, a REST specification.

6.2.1 Security

The system uses token authorization for most features, except when creating a user or logging in into the system. The first doesn't need any specific authorization, while the second uses basic authentication. The server generates a one use token for the user and a new one is generated every time a request is made, except for logging out, where the token is removed.

For the prototype, as a demonstration, the communication is not encrypted in HTTPS. It still is expected to be used, as referred in the non functional requirements (Subsection 5.1.2), in a future released product.

6.2.2 Client-Server Communication

For being lightweight, easy to parse and human-readable, all the requests and responses use JSON for data interchange.

6.2.3 Create Specification

The main function of the server is to permit the user to save the specification on the server and make it available for him at any time. This is achieved through the method *createProjectSpecification*, that is called using the HTTP method POST, with the data shown in Listing 6.6.

```
Specification:
  type: object
  properties:
    id:
      type: integer
      readOnly: true
    name:
      type: string
    specificationType:
      $ref: '#/components/schemas/SpecificationType'
    file:
      type: string
      format: binary
SpecificationType:
  type: string
  pattern: 'REST|SOAP|GRAPHQL'
```

Listing 6.6: Create project specification data

The server, after validating the authorization and the project id, creates the file using the code shown in Listing 6.7.

```
1  createProjectSpecification = function (body, fileBuffer, projectId) {
2      let specType_id = 0;
3      let type = body.specificationType.toLowerCase();
4
5      return new Promise(function (resolve, reject) {
6          try {
7              db.get(`${config.getSchema()}.specificationtype`, "
       specificationtype.id", `specificationtype.type like '${type}'`)
8                  .then(res => {
9                      if (res) {
10                         //...
11                         db.get(`${config.getSchema()}.specification`, "
       specification.id", `specification.path like '${path}'`)
12                             .then(res => {
13                                 if (res.rowCount > 0) {
14                                     //...
15                                 } else {
16                                     let writerStream = fs.
       createWriteStream(path, { flags: "wx" });
17                                     writerStream.write(fileBuffer);
18                                     writerStream.on('error', e => {
19                                         //...
20                                     });
21
22                                     db.insert("specification", "
       projectid,specificationtypeid,name,path", `${projectId},${specType_id
       },'${body.name}','${path}'`)
23                                     //...
24                                 }
25                                 //...
26                             });
27                     }
28                     //...
29                 });
30             //...
31         }
32     });
33 }
```

Listing 6.7: Create project specification

To save the specification in the server it is needed to obtain the specification type id (line **7**) and to verify if the server does not have this specification already saved (line **11**). After this, it is needed to create the file in the server (line **18**) and to save the information into the database (line **22**).

6.3 Summary

In this chapter, the implemented solutions was described, along with the decisions taken for the key areas of the system.

The following questions were answered:

The implementation is compliant with the presented solution? It partially reflects the presented solution, since parts of the system were not implemented, like HTTPS for communication security, SOAP in the front-end and various methods of the back-end. The implemented solution works as a demonstration for the principal functionalities of the proposed solution and responds to the principal problems in study.

How was implemented the front-end The front-end was implemented using HTML, CSS and JS to allow the user to create specifications.

How was implemented the back-end The back-end was implemented using JS, using the node.js runtime and the PostgreSQL database, to allow the user to create specifications.

Chapter 7

Tests and Results

In the last chapter (5) the design of the proposed solution was analyzed in a detailed form. In this chapter, where the **Outcome 6: Solution Evaluation** will be approached, a detailed set of tests and their results will be presented.

In this chapter, the following question will be answered:

What tests can be defined to test the solution?

What results were obtained from the tests?

Could other tests be realized?

7.1 Defined Tests

Since the work of this dissertation should result in a solution that can produce Web API definition files, and save them on server, the tests to be made must encompass the performance of the server and the capability to create Web API definition files.

7.1.1 Hypothesis to Test

It is pretended that the tests validate the capability of the prototype in responding to the objectives of this dissertation.

The objective of the prototype is to be a distinct alternative solution to the ones existing in the market, that can improve on them, using a graphic interface and a server that can save Web API definition files, associated with projects.

7.1.2 Testing Environment

The test environment is a controlled environment with two different computers. The first computer is a Personal Computer, acting as the client. The second computer is a Raspberry Pi 4, with 4 gigabytes of Random Access Memory (RAM), acting as the server.

Both computers are connected to the same private network, through a Wireless Local Area Network (WLAN) connection.

7.1.3 Performance Tests

The usage of servers to perform operations over resources brings the problems of execution time. If a server takes too much time to respond to the client, users will stop using the system. The performance tests to be made must test the time of execution of the methods that will create data on the server and will try to prove that the execution time is less than 1 second in the server (*What is Response Time Testing?* 2021).

For this, it is needed to record a set of results, where each result is the time that takes from the call of the method until it ends its execution (Different than the response of the method being sent, since these can be different). The set of results must be evaluated for maximum time, minimum time, mean time of execution and its standard deviation.

Network state and client side requests must not be taken into account, since they can be influenced by latency and user input, reducing the objectiveness of the test.

7.1.4 Acceptance Tests

The acceptance tests to be made must test the capability to create valid Web API definition files from the web page.

Web API Definition Files validation

There are 2 different tests to be made that can test the validity of Web API definition files generated by the developed prototype.

The first involves uploading the definition file to one of the studied existing solutions in Chapter 4. This will prove that the definition was correctly validated and generated by the developed prototype.

The second test to perform involves generating code, using the Web editor for Swagger or the Eclipse IDE. This will prove that the generated definition files were correctly generated and are capable of generate code.

7.2 Results

In this section, the results of the tests made to the prototype will be described.

7.2.1 Performance Tests

24 different specifications were generated using the developed solution and were sent to the server, to test the performance of the server. An example of the specifications used is in Appendix A. The results obtained are discriminated in the Table 7.1

Table 7.1: Performance Testing - Running Time for Each Request

Test 1	259 ms
Test 2	149 ms
Test 3	208 ms
Test 4	332 ms
Test 5	158 ms
Test 6	174 ms
Test 7	184 ms
Test 8	143 ms
Test 9	165 ms

Test 10	195 ms
Test 11	286 ms
Test 12	329 ms
Test 13	165 ms
Test 14	165 ms
Test 15	124 ms
Test 16	154 ms
Test 17	213 ms
Test 18	148 ms
Test 19	155 ms
Test 20	108 ms
Test 21	218 ms
Test 22	296 ms
Test 23	123 ms
Test 24	183 ms

For this set of results, will be calculated the mean time and its standard deviation.

The mean time ($\overline{X}$) of the requests was calculated using the following formula:

$$\overline{X} = \frac{259 + 149 + 208 + ... + 296 + 123 + 183}{24}$$

$$\overline{X} = \frac{4634}{24}$$

$$\overline{X} = 193.083$$

The mean time of the requests is 193.0833 ms, a good average time for requests when passing a file and creating it on the server.

For calculating the standard deviation of the sample, the following function is applied:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{N} (t_i - \overline{X})^2}{N}}$$

$$\sigma = \sqrt{\frac{\sum_{i=1}^{24} (t_i - 193.0833)^2}{24}}$$

$$\sigma = \sqrt{\frac{93115,83}{24}}$$

$$\sigma = \sqrt{3879.826}$$

$$\sigma = 62.28825$$

The standard deviation of the sample is 62.28825, which shows that the sample has a wider range of values than what is intended, but is expected, taking into account the resources used in the test.

7.2.2 Acceptance Tests

12 different specifications were generated using the developed solution and were tested against the Swagger web editor and the Eclipse IDE. The specifications generated varied in complexity, number of requests created and number of components used. An example of the specifications used is in Appendix A.

All the tests passed in the Swagger web editor, which shows the potential of the prototype and the described solution and that the generated specifications work as intended in both platforms and are able to generate working code.

7.3 Data Significance

In this chapter, the results of the tests have shown that the server is responding as expected, considering the code used and the environment of the tests and the specifications generated are correct.

The results shows that all parts of the prototype are working as intended and the solution has potential in functional terms. The solution reduces the number of possible errors, since most of the problems associated with solutions based on text-insertion are reduced, and the user only has to be preoccupied with the actual specification and let the system handle the language itself.

In terms of performance, the results shows that the obtained values are lower than the defined time in the current environment. Response times for the whole system were not tested, but are required for a more profound study, since it impacts the user experience.

7.4 Summary

In this chapter, the tests to be made, their results, along the reasons for them, were presented. All the tests were realized as defined and the results have shown that the prototype already responds to some of the requirements defined in Chapter 5.

The following question was answered:

What tests can be defined to test the solution?

The tests to be defined must test the performance of the server when using methods that will generate data on the server and test the validity of the generated files.

What results were obtained from the tests?

The obtained results show the prototype potential to be a distinct alternative to current solutions, being lightweight, fast and reliable.

Could other tests be realized? Other tests could be used to better determine the capabilities of the system. The tests used always assume single requests, but multiple requests were not tested. Along with these, a real environment test, with users would be necessary, to better understand how users would receive the prototype and to improve it.

Chapter 8

Conclusions and Future Work

In this document, an improvement to the current methods of crating Web API definition files was studied. The principal objective of this study was to found a possible solution that could achive this improvement. For this, a prototype, *GraphicAPIDefinition*, was developed, along with tests that could prove its viability.

In the last chapter of this dissertation, the conclusions of the elaborated work will be presented, with future work to be made and a final appraisal. The conclusion will explain if the solution presented to the problem can resolve it, how it works and if it is an effective solution to the proposed objectives. The future work will elaborate on the areas that need to be worked on after the implementation of the prototype. Finally, an appraisal of the work done will be made.

8.1 Conclusions

With different approaches existing to the creation of Web API development, a developer or team can choose the best approach for the development of their application. Both approaches have advantages and disadvantages in its use, and the choice of one over the other must be well thought and must try to respond to the requirements of the development of the application. The objective of this dissertation is to give more tools for the developer or teams to better apply the Design First Approach.

After analyzing the solutions in the market, technologies and methodologies, a prototype, *GraphicAPIDefinition* was developed, with the objective to respond to the problems identified, in a simplified format for the user of the solution. This was achieved through a graphic interface that prioritizes functionality, along with a server that that manages specifications of registered users.

The performance tests realized over the server shows that it works and can be used to save projects of registered users, without having a significant waiting time for many requests, in a single request basis. For it to be completely accepted, the tests done to it must be different, mainly stress and acceptance tests. It has the potential to be scaled and maintainable, since it uses a lightweight development framework, but this brings different problems like the definition of the paths to save the files and multiple database accesses.

The acceptance tests realized over the web site generated REST specifications show the potential of the solution to provide correct specifications, independent of the architecture or protocol chosen. On the other side, there is still work to be done, in developing verifications for user inputs and developing a better user interface for the visual part. And the most important part for a project of this category is the user reception of the product, that was not studied. Without Web API developers who use the system, this document is mostly a rhetoric study of different approaches to develop a solution that can generate Web API specifications.

In conclusion, the main objectives of the dissertation were achieved, through a Web site with a graphic interface that generates Web API definition files . These can be saved by the user in its own computer, or sent to the server, where they are associated with a project, that can contain files, of different specification types (like REST or SOAP). Along with the objectives, some functional and non functional requirements were implemented, and only one was discarded for the prototype (usage of HTTPS for communications between client and server).

8.2 Future Work

While the achieved work demonstrates the objectives of the thesis, it still is a very simple prototype. For a working solution to be developed and then officially launched, the solution needs to be mainly worked in four different areas:

- Front-end development

  The first part to develop is the visual area of the web page. The prototype has a structured HTML, but it was not developed in the visual area, mainly in the CSS files, and this is an essential part to be worked in the future.

  The second part is to optimize the logic area of the web page. The prototype was created without regards to the optimization of code, with some parts to be optimized and changed. Some code also needs to be reworked in terms of input verification.

- Back-end development

  The principal part to work is the development of the methods that were deemed not necessary for the solution demonstration. A review of the database is necessary to improve the application in security areas.

- Security

  The prototype did not implement enough security during its development and this is one of the main concerns for future work. The first part to work is the implementation of authentication and authorization across the system. The second part of the system to work is to protect the communication between server and clients, using HTTPS and encrypted messages using a hash algorithm. The last part to do is to work in encrypting the saved files in the server.

- Different Specification types

  The prototype only has implemented the creation of specifications for REST architecture, using the YAML notation. To achieve all the objectives of this dissertation, the possibility to generate other specifications, following different architectures, protocols and notations needs to be implemented.

8.3 Final Appraisal

This dissertation was a developing moment for the author, since it was outside of its area of comfort, mainly in the development and design of the front-end of the solution. The main objectives of the dissertation, in the context of the master degree of informatics engineering, were achieved, since it used many of the knowledge acquired in it, like security, server-client communication, API specification creation and solution analysis. The project was very interesting and a great challenge, with multiple technologies being used, with almost all of them for the first time outside a classroom. This took some time of the development of the prototype, but gave the author some ideas to use in the process and in future work.

Bibliography

Architecture (Apr. 2012). url: https://techterms.com/definition/architecture (visited on 2021).

Box, Don et al. (May 2000). *Simple Object Access Protocol (SOAP) 1.1*. url: https://www.w3.org/TR/2000/NOTE-SOAP-20000508/ (visited on 2021).

Fast Simple API Development | SwaggerHub (2021). url: https://swagger.io/tools/swaggerhub/features/ (visited on 2021).

How to Design REST API with UML? (2021). url: https://www.visual-paradigm.com/support/documents/vpuserguide/276/3420/85154_modelingrest.html (visited on 2021).

Introduction to the WSDL Editor (Mar. 2008). url: https://wiki.eclipse.org/Introduction_to_the_WSDL_Editor (visited on 2021).

Karanam, R. (Nov. 2019a). *Designing REST API - What is Code First Approach?* url: https://www.springboottutorial.com/rest-api-code-first-approach (visited on 2021).

– (Nov. 2019b). *Designing REST API - What is Contract First?* url: https://www.springboottutorial.com/rest-api-contRact-first-approach (visited on 2021).

Nicola S., E.P. Ferreira and J. Pinto Ferreira (2012). "NOVEL FRAMEWORK FOR MODELING VALUE FOR THECUSTOMER, AN ESSAY ON NEGOTIATION". In: *International Journal of Information Technology Decision Making* 11.3, pp. 661–703. url: https://repositorio.inesctec.pt/bitstream/123456789/2595/1/PS-07876.pdf.

Protocol (Mar. 2019). url: https://techterms.com/definition/protocol (visited on 2021).

What is Response Time Testing? (Oct. 2021). url: https://www.guru99.com/response-time-testing.html (visited on 2021).

What is REST? (Oct. 2021). url: https://restfulapi.net (visited on 2021).

XMLSpy Highlights (2021). url: https://www.altova.com/xmlspy-xml-editor#web_services (visited on 2021).

Appendix A

Test Specification

```yaml
openapi: 3.0.1
info:
  title: test2
  description: Specification tests
  contact:
    email: tests@server.pt
  license:
    name: GPL V3
    url: https://www.gnu.org/licenses/gpl-3.0.html
  version: v1
servers:
- url: localhost/server/v1
paths:
  /entity:
    get:
      description: get a List of entities
      operationId: getEntities
      responses:
        200:
          description: Entity List
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/entity1'
    post:
      description: Create a new Entity
      operationId: createEntities
      requestBody:
        description: New Entity
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/entity1'
        required: true
      responses:
        201:
          description: Created
        400:
          description: Bad Request
    put:
      description: Update or Create Entity
      operationId: updateEntity
      requestBody:
        description: Entity to update
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/entity1'
        required: true
```

Listing A.1: Example of a REST specification

```
      responses:
        201:
          description: Created
        204:
          description: No Content
        400:
          description: Bad Request
  /user:
    get:
      description: Get a list of user
      operationId: GetUserList
      responses:
        200:
          description: Get User List
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/userList'
    post:
      description: create User
      operationId: createUser
      requestBody:
        description: user to create
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/user'
        required: true
      responses:
        201:
          description: User created
          headers:
            location:
              description: location of user
              schema:
                type: string
              required: true
        400:
          description: Bad Request
    put:
      description: Update or Create User
      operationId: updateUser
      requestBody:
        description: user to update
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/user'
        required: true
```

Listing A.2: Example of a REST specification

```
      responses:
        201:
          description: USer created
          headers:
            location:
              description: user location
              schema:
                type: string
              required: true
        204:
          description: Updated
        400:
          description: Bad Request
components:
  schemas:
    entity1:
      type: string
    entityList:
      type: array
      items:
        $ref: '#/components/schemas/entity1'
    user:
      type: object
      properties:
        name:
          type: string
        age:
          type: integer
        email:
          type: string
        active:
          type: boolean
        type:
          $ref: '#/components/schemas/entity1'
    userList:
      type: array
      items:
        $ref: '#/components/schemas/user'
```

Listing A.3: Example of a REST specification