

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Designing, Implementing, and Deploying a Better Customer-oriented, Secure REST API for Invoicing Software

Miguel Rodrigues Gomes



Mestrado em Engenharia Informática

Supervisor: Jácome Cunha, Eurico Inocêncio, André Catita

January 31, 2023

Designing, Implementing, and Deploying a Better Customer-oriented, Secure REST API for Invoicing Software

Miguel Rodrigues Gomes

Mestrado em Engenharia Informática

January 31, 2023

Resumo

A utilização de APIs (Application Programming Interfaces) é muito frequente no âmbito do panorama atual da programação. Ainda assim, existe uma grande dificuldade por parte dos programadores ao aprender a utilizar uma nova API. Quando uma empresa disponibiliza a sua API aos clientes, o tempo dispendido pelos funcionários na instrução dos clientes, e o tempo que estes levam a aprender poderia ser evitado, caso a API tivesse sido construída de uma forma mais intuitiva e focada no cliente. Uma das formas mais comuns de combater este problema é disponibilizar uma boa e bem construída documentação, não sendo, porém, uma tarefa simples. Dado que uma API corresponde a uma ligação quase direta entre um utilizador e a base de dados da aplicação, a segurança é de capital importância. Um pequeno erro pode conduzir ao acesso a informação privilegiada por utilizadores maliciosos, isto é, pode levar a problemas de privacidade e fugas de informação. Estes são os principais objetivos do desenvolvimento desta API, além de assegurar a atual performance. A segurança foi testada utilizando uma ferramenta de teste automática, que ajudou a identificar possíveis pontos fracos em termos de segurança e performance. Este trabalho permitiu desenvolver uma nova API, mais simples e segura, estando neste momento a ser utilizada pelos clientes da empresa. Esta foi considerada mais simples de usar, tendo em conta o número de pedidos de ajuda recebidos, e o parecer de trabalhadores da empresa. A segurança foi objetivamente avaliada com um resultado positivo. Finalmente, a performance revelou-se semelhante à da versão anterior da API, que já estava provada como suficiente aos olhos da empresa para o caso em mão, com uma melhoria temporal ao utilizar as novas funcionalidades. Em suma, durante todo o trabalho desenvolvido, a versão inicial da API foi enriquecida com recursos que facilitam a sua utilização, nomeadamente nova documentação, gerada em parte manualmente, e em parte gerada automaticamente por uma ferramenta desenvolvida no âmbito deste trabalho. Uma nova API foi desenvolvida, com cerca de 30 rotas implementadas desde raiz. Esta foi desenhada com o cliente em mente, e conta com uma melhor usabilidade, segurança, e performance semelhante. Esta é também acompanhada de nova documentação. Um novo processo de integração na API foi desenvolvido, sendo que um conjunto de novos menus e páginas foram lançados no software da empresa, que auxiliam o utilizador no início da sua integração com a API de uma forma mais segura, e que permite recolher dados sobre o integrador desta API.

Abstract

Modern programming frequently requires the use of APIs (Application Programming Interfaces). Yet many programmers struggle when trying to learn them. When a company provides its API for customers to use, the time spent learning it, and the time spent by the employers teaching them, is a waste of effort, which could be avoided by building a more intuitive and user-friendly API. One of the main resources used to teach APIs is by providing a detailed, clear, documentation - however, that is not a straightforward task. Another aspect is that an API is a direct connection between a user and the application's database, and thus security is of utmost importance. A small error can provide an ill-intentioned user with full access to the application's data, which can lead to privacy issues and sensitive data leaks. Security and usability are the main goals of the development of this API, aside from maintaining the current performance. The security was tested using an automated API security tool, which helped to identify weak points, and guarantee a secure API in the end. The new API was evaluated with much better usability, according to the number of help requests received, and feedback from the company's employees. The security was objectively evaluated, and returned a good result. Lastly, the performance tested was similar to the previous version of the API, which had already been proved as satisfactory, with a great time improvement when using the new features. This work allowed the development of a more secure, and simple API, which is currently being used by the company's clients. During the development of this work, the original API was improved by developing external tools and documents to aid its usage. A great number of pages describing the usage of the API was written by hand, and some other pages were generated automatically by a tool developed in this work which scrapes the information of the existing endpoints, and generates documentation. A new API was developed, with around 30 new endpoints implemented and created from scratch. This was developed with the customer in mind, and is more usable, more secure, with a similar performance. The new API also comes with a great documentation developed similarly to the previously mentioned documentation. Lastly, a new enrollment process in the API was developed. New pages and menus were created and developed in the company's software which aids the clients in beginning their integration with the API, while simultaneously helping the company gather data on the API's developer.

Acknowledgments

Firstly, I would like to thank all of my friends and family. They are the most important people in my life, and are the ones who kept pushing me further and further, and made me who I am today. If it were not for them, I would have never made the leap to Computer Science, and would never have accomplished what I have today. I would also like to thank the remaining people who made this work possible. Thank you to Cloudware, S.A for giving me the opportunity to work for them during this last year, and for allowing me to develop this work. Thank you to FEUP, for allowing me to study in this degree, and giving me the vast majority of knowledge I have in Informatics. A special thanks to my university advisor, Jácome Cunha, and my advisors at Cloudware, S.A: Eurico Inocêncio, Jorge Morais, and André Catita.

Miguel Gomes

Contents

1	Introduction	1
1.1	Context	1
1.2	Motivation	1
1.3	Goals	2
1.4	Document Structure	2
2	The original API and the problem at hand	4
2.1	The original API	4
2.1.1	Authentication	4
2.1.2	Document-related endpoints	5
2.1.3	Obtaining a pdf of a document	9
2.1.4	Customers	9
2.1.5	Suppliers	10
2.1.6	Products and Services	10
2.2	Problem	10
2.2.1	Usability	10
2.2.2	Security	11
2.2.3	Performance	11
3	State of the art	12
3.1	Fundamental Concepts	12
3.1.1	API	12
3.2	Related work	15
4	Security and Performance testing tool analysis and selection	19
4.1	Tool Analysis	19
4.1.1	JUnit	19
4.1.2	cURL	20
4.1.3	Postman	21
4.1.4	Advanced REST Client	22
4.1.5	Mocha and Chai	22
4.1.6	TestNg	23
4.1.7	EvoMaster: Automated API Testing	23
4.1.8	Automated API Testing	24
4.1.9	RESTler: Stateful REST API Fuzzing	24
4.1.10	QuickREST: Property-based Test Generation of OpenAPI-Described RESTful APIs	24
4.2	Tool Selection	25

5	Developing and implementing a new API	26
5.1	Documenting the existing API	26
5.2	A new design for the API	33
5.3	Implementating the new API	38
6	Studying and Improving the Application's Security	40
6.1	Unit Testing	40
6.2	Fuzzing	41
6.3	Malformed Payload Injection	43
6.4	Malicious Content Injection	44
6.5	DDoS	44
6.6	Man in the middle	47
6.7	Social engineering	48
7	Evaluating the Security of the Application	50
7.1	Setup and Design of the Evaluation	50
7.1.1	Application Modeling	50
7.1.2	Metric Selection, Test Case Definition, Test Case Execution, and Measure- ment Collection	51
7.2	Results	52
7.2.1	Assurance Metrics	52
7.2.2	Level Calculation	54
7.3	Result Analysis	56
8	Evaluating the Performance of the Application	57
8.1	Performance Evaluation	57
8.1.1	Setup and Design of the Evaluation	57
8.1.2	Results	58
8.1.3	Result Analysis	60
9	Deployment and release of the API	62
9.1	Documentation	62
9.2	Enrollment Process	63
10	Conclusions and Future Work	66
10.1	Goals met	66
10.2	Future work	67
	References	68

List of Figures

4.1	Postman GUI	21
4.2	Advanced REST Client GUI	22
5.1	Swagger API documentation	30
5.2	Authentication page snippet	31
5.3	Postman OAuth as imported by the script-generated YAML	32
5.4	Postman request with all the necessary headers, body, and parameters, imported by the same YAML	33
5.5	Snippet of a documentation page, describing the customers route	34
8.1	Performance results of the new API, in comparison to the old version	59
8.2	Performance results of the new API, in comparison to the old version	60
9.1	Bill page of the new documentation	63
9.2	First screen of the enrollment process	64
9.3	Second screen of the enrollment process	64
9.4	New API credentials page	65

List of Tables

3.1	API testing tools [16]	17
3.2	Tools, Methodologies and Frameworks for API Testing [17]	18
4.1	Statement coverage results for the generated tests compared to the ones obtained by the already existing, manually written tests. [8]	24
7.1	Example of applying GQM approach to quantify the fulfillment factor for the security requirement (sub-goal): Use input sanitization.[28]	52
7.2	Evaluation of the Vulnerabilites and Security Requirements for the application . .	53
7.3	Degree of Security [28]	55

Listings

2.1	cURL command to obtain the authorization code	4
2.2	cURL request to obtain the Bearer Token	5
2.3	Header creation request	5
2.4	Payload JSON for the header	6
2.5	Line creation request	7
2.6	Payload JSON for the line	7
2.7	Finalize document request	8
2.8	Document void request	8
3.1	REST API request structure example	13
3.2	JSON-RCP call and response	14
3.3	SOAP API message	14
3.4	SOAP API response	15
4.1	JUnit test example	19
4.2	cURL example	20
5.1	Customers API route YAML specification file header	26
5.2	Customers API route YAML specification file relations	28
5.3	YAML API description example	29
5.4	JSONAPI SQL code	33
5.5	Original JSONAPI payload structure	34
5.6	New payload structure	35
5.7	New sales document payload	36
5.8	New add lines request	37
5.9	Example of gatekeeper entry	39
6.1	Ruby rspec example	41
6.2	Defining RESTler strings	42
6.3	Log information structure	45
6.4	Request example in log	45
6.5	API request log metadata	47

Abbreviations and Symbols

API	Application Programming Interface
ASVS	Application Security Verification Standard
CRUD	Create Read Update Delete
DDOS	Distributed Denial of Service
GQM	Goal Question Metric
GUI	Graphical User Interface
JSON	JavaScript Object Notation
JWT	JSON Web Token
OWASP	Open Web Application Security Project
PostgreSQL	Postgres SQL
PDF	Portable Document Format
PID	Process Identifier
REST	Representational State Transfer
RPC	Remote Procedure Call
SQL	Structured Query Language
SUT	System Under Test
TDD	Test Driven Development
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language

Chapter 1

Introduction

To start this work, the first chapter will provide a simple introduction. The context and motivation for this work will be presented, followed by the goals and document structure.

1.1 Context

Cloudware S.A. has developed an integrated cloud native solution which provides the ability for any certified accountant to perform all the accounting and invoicing. This can be done when representing an individual or a company. This software is commercialized in three different 'flavors', depending on the customer it is sold to - TOCOnline is the original, sold to accountants; Business is sold to companies; Ensino is made available to accounting students at selected universities, to help teach some courses, and make students familiar with the product. Despite this, all the three softwares are identical, providing the same functionalities, and sharing the same code base. During the development of this software, an internal JSONAPI was developed to be used by the software itself in its backend. As the number of requests for the company to provide an API rose, the company decided to make some of the API's endpoints available to its customers. Slowly, more endpoints were made available, and thus an official client API was born. Given its origin, the original API was not the most customer-friendly, nor the most secure. As it was not designed to be a commercial API, its usage is difficult, there is no documentation available, and a security study has never been performed.

1.2 Motivation

APIs are one of the most frequently used pieces of software for any programmer. Despite this, it can still be very difficult for a programmer to learn to use a new API. This is costly both for the programmer trying to learn to use this API, and for the entity who makes it available [21]. Therefore, it is of utmost importance to design an API that is as easy as possible to start using. As it was mentioned in the previous section, the API made available for Cloudware's customers was not designed with the user in mind. Thus, the API provided was not tested for security purposes

and was not designed to be easy to use. This resulted in a large number of help requests by the clients which, at the beginning of this work, took up many hours of the company's time. Hence, it has surged the necessity to create a more user friendly API, considering security requirements, while keeping the same performance. It is also necessary to improve the original API, as it will still be used in the near future.

1.3 Goals

The main problems with the original API are the possible lack of security, and the difficulty to be used. Therefore, the two first goals of this work consist of solving these problems. Although its performance is, at the start of this work, admissible, it is important to study possible improvements, and verify if it is not affected when improving security and usability. Thus, the three main goals of this work are to create a new API which:

- Has usability in mind, and is user-friendly
- Has security in mind, and is objectively secure in the end
- Has performance in mind, and keeps similar results as the previous version

Aside from the creation of this new API, this work also aims to create tools to automatically generate documentation for the existing API, as it will continue to be used in the near future; use this tool to create documentation for both APIs; and lastly create a new enrollment process for both APIs which allow the company to register information regarding their clients' API developers, and provide the API credentials in a more secure manner.

In order to facilitate the client's learning curve, the API had to be redesigned. At the beginning of this work, the API is a pure implementation of JSONAPI [44] which uses overly complicated payload structures for its requests, and requires many different steps for simple actions such as creating an invoice. Therefore, a study will be performed with the objective of determining what is the best new design, in terms of different endpoints and payloads. This study comprises the customers needs, and tries to provide a new application with the easiest and most intuitive usage possible. The main goal of this section is to provide an application and utilities that all customers can use without requiring help from the company's employees. On the other hand, it is also necessary a study to determine what is the most reliable strategy to test the API's security. Therefore, in this work, after the strategy is chosen, a study on the API's security will be presented. All of the possible weak spots will then be fixed, and described in this work. Lastly, the API's performance will be tested to guarantee it has not been affected by the introduced improvements.

1.4 Document Structure

Aside from the introduction, described thus far, this work is composed by more chapters. Chapter 2, will be divided in two different subchapters. First, the original API is described, with all of the

different requests and functionalities detailed. Next, the problem at hand is described in terms of usability, security, and performance. Chapter 3 will detail the state of the art, and is also divided in two subchapters. First, some fundamental concepts essential to the reading of this work will be described and explained. Next, a study about different works, related to this work's theme, will be done. With a direct connection to this subchapter, the existing technologies in this work will also be described. Next, in Chapter 4, a plethora of security and performance testing tools are studied, with the objective of determining the most appropriate one to test the application at hand. In Chapter 5, the usability improvements are described. These consist of the entire rework of the API, and the construction of the new version. Apart from this, so as to improve the existing version of the API, I developed documentation for the entire application, and a few programs which aided this process and helped the users' enrollment in the API. Lastly, I performed a security study detailed in Chapter 6, which culminates in an objective evaluation of the application's security performance present in Chapter 7. This is followed by Chapter 8 in which the performance and usability of the new API is evaluated. Chapter 9 details the deployment of the solution devised. It consists of the new documentation developed, as well as the new enrollment process for the API. The last chapter, Chapter 10 details the goals met, and the possible work that can be done to continue and improve the application.

Chapter 2

The original API and the problem at hand

The present chapter will describe the existing API, made available to the customers. All of the steps necessary to its usage, as well as all of the commonly used requests will be detailed. Lastly, the existing problems in this application, which ended up motivating this work will be detailed.

2.1 The original API

The original API used by TOCOnline, and given to their customers, currently contains a number of endpoints, and will be detailed in the present section.

2.1.1 Authentication

The authentication routes are the first to be used in this API. Currently, the OAuth 2.0 authorization framework is used [24]. The authentication process is done as described in the OAuth2.0 official documentation, and has no customization in place. The first step is to obtain some data, available in the TOCOnline interface: `client_id`, `client_secret`, `oauth_url`, and `api_url`. The `client_id` is unique, and immutable. The `client_secret` is a base64 hash, used as a secret key, which used to be immutable, but was made available to be changed by the clients, in order to improve security. The `oauth_url` is the URL for which the authentication requests should be performed, and the `api_url` is the URL for which the api requests should be performed. The second step is to obtain the authorization code. To do so, the following request should be performed:

```
1 curl -v -H 'Content-Type: application/json' \  
2 '<OAUTH_URL>/auth?client_id=<client_id>\  
3 &redirect_uri=http://127.0.0.1:4080/oauth/callback\  
4 &response_type=code&scope=commercial'
```

Listing 2.1: cURL command to obtain the authorization code

This leads to a response containing the authorization code, necessary for the next step. The third and final step in the authentication process is to obtain the Bearer Token, which will be used in every subsequent request, as a way of identifying ourselves to the API. For this, the following request should be performed:

```
1 curl -v -X POST -H 'Content-Type: application/x-www-form-  
    urlencoded' \  
2 -H 'Accept: application/json' \  
3 -H 'Authorization: Basic <client_id + ':' + secret, coded in  
    Base64>' \  
4 -d 'grant_type=authorization_code&\n  
5 code=<authorization_code>&scope=commercial' \  
6 '<OAUTH_URL>/token'
```

Listing 2.2: cURL request to obtain the Bearer Token

This leads to a response containing the Bearer code. The process is now complete, and the authentication is done.

2.1.2 Document-related endpoints

All the following endpoints follow the same path, and are related to the creation of different documents: bills (sales documents), receipts, purchases documents, and purchases payments. All these documents follow the same structure as they are constituted by a header, and a set of lines. The header contains information about the entities involved in the transaction, and information about the transaction itself such as date, and document type. The remainder of the document is composed of the lines, each of which identifies and describes a single item of the document. For example, in a receipt, each line describes a single product purchased in the transaction described. In order to create one of these documents through the API, several steps are needed. What follows is the description of the process of a bill creation. The creation of the remaining documents is similar, and describing each one would be redundant. The first step is to create the header. To do so, the following request should be performed.

```
1 curl -v -X POST -H 'Content-Type: application/vnd.api+json' \  
2 -H 'Accept: application/json' \  
3 -H 'Authorization: Bearer <access_token>' \  
4 -d '<payload JSON>' '<API_URL>/commercial_sales_documents'
```

Listing 2.3: Header creation request

In this request, the payload JSON should comply with the following structure.

```
1 {
2   data: {
3     type: commercial_sales_documents,
4     attributes: {
5       document_type: FT,
6       date: 2020-01-01,
7       due_date: 2020-02-01,
8       customer_tax_registration_number: 999999990,
9       customer_business_name: Test customer,
10      customer_address_detail: Example Street 777A,
11      customer_postcode: 4200-224,
12      customer_city: Porto,
13      customer_country: PT,
14      settlement_expression: 7.5,
15      vat_included_prices: false,
16      currency_iso_code: USD,
17      currency_conversion_rate: 1.21,
18      notes: Document notes,
19      external_reference: Reference to the external document,
20      payment_mechanism: MO
21    },
22    relationships: {
23      commercial_document_series: {
24        data: {
25          type: commercial_document_series,
26          id: <id of the associated document series>
27        }
28      },
29      tax_exemption_reasons: {
30        data: {
31          type: tax_exemption_reasons,
32          id: <id of the tax exemption reason>
33        }
34      },
35      bank_accounts: {
36        data: {
37          type: bank_accounts,
38          id: <id of the bank account>
39        }
40      }
41    }
42  }
43 }
```

```
40     }
41   }
42 }
43 }
```

Listing 2.4: Payload JSON for the header

At this point, the bill header is created. The API responds with an id for the newly created bill header, as well as all of the information stored regarding it.

Next, for each product or service that is related to this bill, the following request should be performed.

```
1 curl -v -X POST -H 'Content-Type: application/vnd.api+json' \
2 -H 'Accept: application/json' \
3 -H 'Authorization: Bearer <access_token>' \
4 -d '<payload JSON>' \
5 '<API_URL>/commercial_sales_document_lines'
```

Listing 2.5: Line creation request

The payload JSON should be as follows:

```
1 {
2   data: {
3     type: commercial_sales_document_lines,
4     attributes: {
5       quantity: 1,
6       unit_price: 20,
7       item_type: Product,
8       item_code: PTEST,
9       tax_code: NOR,
10      description: Line description,
11      settlement_expression: 3
12    },
13    relationships: {
14      document: {
15        data: {
16          type: commercial_sales_documents,
17          id: <document id>
18        }
19      },
```

```
20     unit_of_measure: {
21       data: {
22         type: units_of_measure,
23         id: <unit of measure id>
24       }
25     }
26   }
27 }
28 }
```

Listing 2.6: Payload JSON for the line

At this point the document is created. It is still not valid, as it needs to be finalized. This guarantees that the document can no longer be altered, a unique QR code is generated, and a pdf can be requested, using a different endpoint. In order to finalize the document, the following endpoint should be used.

```
1 curl -v -X PATCH -H 'Content-Type: application/vnd.api+json' \
2 -H 'Accept: application/json' \
3 -H 'Authorization: Bearer <access_token>' \
4 -d '
5 {
6   data: {
7     type: commercial_sales_documents,
8     id: <document id>,
9     attributes: {
10       status: 1
11     }
12   }
13 }
14 ' \
15 '<API_URL>/commercial_sales_documents/<document_id>'
```

Listing 2.7: Finalize document request

The document can also be voided, should it be necessary.

```
1 curl -v -X PATCH -H 'Content-Type: application/vnd.api+json' \
2 -H 'Accept: application/json' \
3 -H 'Authorization: Bearer <access_token>' \
4 -d '
```

```
5 {
6   data: {
7     type: commercial_sales_documents,
8     id: <document_id>,
9     attributes: {
10      status: 4,
11      voided_reason: Text explaining the reason to void the
        document
12    }
13  }
14 }
15 '\
16 '<API_URL>/commercial_sales_documents/<document_id>'
```

Listing 2.8: Document void request

2.1.3 Obtaining a pdf of a document

As it was mentioned before, it is possible to obtain the pdf of a document, through the API. To do so, a request to the `<API_URL>/url_for_print` endpoint should be performed. This request, if successful yields a result containing a number of attributes - namely scheme, host, port, and path. Using these, it is possible to obtain a link to download the PDF of the requested document. To do so, a number of string operations has to be performed to the various attributes that are returned. This is still a difficult and confusing process.

2.1.4 Customers

The API allows for CRUD operations on the data of customers for a given company. It is possible to create a customer by performing a POST request to `<API_URL>/customers`. Similarly, it is possible to delete a customer by performing a DELETE request to the URL, identifying the customer unique id in the URL. A PATCH performed in the same manner allows the alteration of the specified customer. Lastly, it is possible to obtain the information on existing customers by performing a GET request, either to `<API_URL>/customers` to obtain the information on all customers, or to `<API_URL>/customers/id`, to obtain the information on a single customer. As for any request performed to a JSONAPI, it is possible to filter the results received, by selecting a specific field. The customers also contain an address and email, both of which can be inserted or altered, using the `<API_URL>/addresses` and `<API_URL>/email_addresses` routes.

2.1.5 Suppliers

The suppliers behave exactly in the same manner as the customers routes. CRUD operations are possible. POST to <API_URL>/suppliers creates a new entry; GET to <API_URL>/suppliers or <API_URL>/suppliers/id returns the information on all or a single supplier, respectively. a DELETE to <API_URL>/suppliers/id deletes the entry; a PATCH to the same endpoint alters the information on the specified supplier. The same endpoints for the addresses and email addresses of customers can be used for suppliers in the same manner.

2.1.6 Products and Services

A company can sell products or provide services. In order to create new products or services, a request can be performed to the endpoint <API_URL>/products, or <API_URL>/services, respectively.

2.2 Problem

Usability, security, and performance are the main problems to solve. The first two problems go hand-in-hand, as it has been shown that usability impacts security. For example, a study on a collection of almost 14000 applications on the Play Store has revealed that 8.0% had misused either Secure Sockets Layer (SSL) or its successor, the Transport Layer Security (TLS), and were vulnerable to man-in-the-middle attacks, as a consequence [18]. Lastly, adequate performance must be guaranteed. A certain performance is required for the users to want to use the API. If this condition is not met, the clients will not use the API. In this scenario, usability loses its meaning. If a software is not used, its usability is irrelevant.

2.2.1 Usability

Any programmer can identify at least one situation in which they had difficulty learning and using a new API. From the perspective of an API designer, this should not happen - an API should be as easy to pick up as possible. In order to design an API which does not lead to this problem, first it is necessary to identify all of the possible problems. In order to help the users of the TOCOnline API, Cloudware S.A. provides support through the called 'tickets', which are essentially e-mail conversations. By analyzing these tickets, it is clear that the majority of the difficulties the users encounter relate to the following:

- The structure of the JSON payload sent in some requests
- The authentication process
- The creation of any document requires a large number of steps

All of these problems have one core characteristic in common: they are all related to the complexity of the API. This is one of the most common problems in the usage of APIs [35].

2.2.2 Security

Compared to conventional GUI-based application testing, API testing is unique. It evaluates the programmatic interface that permits usage of the business logic or data. API testing focuses on verifying the business logic of the remote software component and its communication method rather than the appearance and feel of the application. As a result, API testing is carried out using specialized software that queries the API and analyzes the responses it returns. Chapter 4 examines the value of API testing, its difficulties, various testing factors, and methods for performing API testing [16]. APIs provide a close connection between a client, and the database. In the case of JSONAPI, this connection is almost direct. Therefore, APIs are a very common point of attack. Furthermore, their security is of utmost importance. In the case of TOCOnline, the security is even more crucial, as this software deals with very sensitive data relative to companies, and their customers. The security testing will be divided in several components, related to the different types of attacks that can occur.

2.2.3 Performance

It is important to evaluate the performance of the API, as it is not useful if it is not capable of responding to client's requests within a feasible time. In order to test the performance, it is necessary to obtain some information about its typical usage. In Chapter 7, both the existing API and the new version were tested, in order to determine their response times, and to verify if there were no performance losses.

Chapter 3

State of the art

As a first start to this work, the state of the art relative to the science area in which this work is encompassed must be studied. To start this chapter some concepts, fundamental to the reading of this work will be detailed. These are mainly related to the accounting world, and the concepts regarding APIs. Lastly, the related work regarding the development, and study of APIs will be presented. This will shed light on good practices on API development, as well as guidelines to study security and performance in this type of applications.

3.1 Fundamental Concepts

3.1.1 API

An API, or Application Programming Interface is, in its core, a form of communication between two or more computer programs [37], and is one of the most important concepts in software development. According to programmableweb, there are currently over 24 000 public APIs [15]. Although the term API is usually used to refer to web APIs, it can be used in many different ways. APIs can be used in operating systems, as a specification between an application and the operating system [31]. They can also be used as Remote APIs, with applications such as the Java remote method invocation API. Lastly, they can also be used in libraries and frameworks; for example, they can be used as language bindings, which map features and capabilities of different languages, allowing for a service written in one language to be translated into another language [34]. Web APIs will be referred to as APIs, for the fact that it is the most relevant use case for this work. When it comes to API architecture, there are three main categories: REST, JSON-RPC/XML-RPC, and SOAP.

RESTful APIs are arguably the most widely used flavour of APIs today. REST stands for Representational state transfer and thus, a RESTful API is one that conforms to the constraints of the REST architectural style. This style's criteria consists of [20]:

- HTTP is used to manage requests in a client-server architecture, which consists of clients, servers, and resources.

- Stateless client-server communication means that each request is distinct from the others and that no client data is kept between get requests.
- Data may be cached to speed up client-server communications.
- A consistent interface between components to communicate data in a common format. This demands that:
 - The resources sought be distinguishable from the client's representations.
 - The information contained in self-descriptive messages sent back to the client is sufficient to specify how the client should handle them.
 - After accessing a resource, a client should be able to use hyperlinks to locate all other currently accessible activities they may perform since hypertext/hypermedia is available.
- Each type of server (those in charge of security, load-balancing, etc.) is involved in retrieving requested data from invisible to the client hierarchy.
- Code-on-demand is an optional feature that allows the server to provide the client executable code upon request, enhancing client capability. One of the most well-known examples of this feature is Adobe's ActionScript language for the Flash Player, which allowed a user to run a web game, client-side, only having access to the language and engine when needed [13].

This flavour of APIs is used by performing a request, and receiving back a response. A request consists of an endpoint which will receive the request, a method which indicates the operation we wish to perform, a set of headers which can define security protocols, timeouts, what content will be sent, what should be received, etc. They essentially serve to add extra information to the request. Lastly, a body can be sent when data is meant to be delivered.

```
1 curl -v -X
2 POST --method
3 -H 'Content-Type: application/vnd.api+json' -H 'Accept:
   application/json' -H 'Authorization: Bearer <access_token>'
   --headers
4 -d '<payload JSON>' --body
5 '<API_URL>/v1/commercial_sales_documents' --endpoint
```

Listing 3.1: REST API request structure example

JSON-RPC/XML-RPC, further referred to as RPC, consists of the use of the RPC pattern, which is fundamentally a function-oriented pattern, in which procedures, methods, or functions are called, passing parameters, and return a result. This approach is widely used, and many languages have their own RPC model. For example, Java has its own implementation, the RMI [40]. Using this

technology, different machines can communicate, by sending requests which directly invoke a method call in the other machine. This request can be encoded using either the JSON or XML formats. JSON-RPC has a narrower use-case set, while XML is widely used, and is one of the most important standards in data representation for semantic data interoperability [42]. A simple example of this framework, can be seen using an example of the official website [33]. In this example, as described by the author ' ->' indicates data being sent to the server, while '<- ' indicates data being sent to the Client.

```
1 --> {
2     "jsonrpc": "2.0",
3     "method": "subtract",
4     "params": [42, 23],
5     "id": 1
6 }
7 <-- {
8     "jsonrpc": "2.0",
9     "result": 19,
10    "id": 1
11 }
12
13 --> {
14     "jsonrpc": "2.0",
15     "method": "subtract",
16     "params": [23, 42],
17     "id": 2
18 }
19 <-- {
20     "jsonrpc": "2.0",
21     "result": -19,
22     "id": 2
23 }
```

Listing 3.2: JSON-RCP call and response

Unlike REST, which defines an architectural style for the development of an API, SOAP refers to an underlying protocol. It can be referred to as a messaging standard, which was defined by the World Wide Web Consortium. SOAP can be described as an example of an implementation of RPC. As an example of this protocol, a simple request to a GetUser route, passing a UserId as the only argument, would be written as described in listing 3.3 [40].

```
1 <?xml version="1.0"?>
2 <soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-
  envelope">
3   <soap:Header>
4   </soap:Header>
5   <soap:Body>
6     <m:GetUser>
7       <m:UserId>123</m:UserId>
8     </m:GetUser>
9   </soap:Body>
10 </soap:Envelope>
```

Listing 3.3: SOAP API message

On the other hand, the return would look like the following, described in listing 3.4.

```
1 <?xml version="1.0"?>
2
3 <soap:Envelope
4   xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
5   soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">
6
7   <soap:Body>
8     <m:GetUserResponse>
9       <m:Username>Tony Stark</m:Username>
10    </m:GetUserResponse>
11  </soap:Body>
12
13 </soap:Envelope>
```

Listing 3.4: SOAP API response

Even in such a simple example, the SOAP message already has a complicated structure. This is because of its standard messaging protocol structure, in which the messages are composed of: Envelope, Body, Header, and Fault. Although only the first two are mandatory [25].

3.2 Related work

In previous work [17], the authors provide an extensive overview on the state of the art in testing RESTful APIs. This is a review work, which succeeded in researching and detailing the various works which discuss different tools, approaches, and frameworks for API testing. These are detailed

in Table 3.2, and the tools selected and displayed in this table will be analyzed in detail as possible tools to be used to test the application developed. The website, appropriately named API Usability [14] is usually the one referred to, when talking about API usability. The site itself contains a large collection of articles relating to the topic. This website immediately recommends an overview about API Usability [35]. The theoretical component of this work is highly based on this referenced work, as well as the work API Testing Strategy [16] as it presents a great amount of important information relevant to it. This work [16] describes the must-have and nice-to-have features of an API testing tool. This is an extensive list, for my work many of these points are not relevant. Thus, the selected must-have features for this work are presented below, and are the requirements for the tool to be chosen.

For security and usability purposes:

- Automated API testing, both for success and error conditions.
- Repeatable testing for multiple environments.
- Creation of HTTP requests.
- Payload generation in JSON format.
- Import API requests from Postman.
- Parametrized test creation.
- Data validation for requests/responses.
- Define test flow logic.
- Authentication security testing.
- Message encryption security testing.
- Penetration attack testing.

For performance purposes:

- Test response times.
- Test error rates.
- Simulate regular performance loads.
- Simulate unusual spikes of usage.

This work presents Table 3.1, with a number of tools commonly used for API testing, which satisfy the requirements aforementioned [16]. The Unit Testing Tools will be used to guarantee functionality and security in the application, while the Performance Testing Tools will measure

Unit Testing Tools	Performance Testing Tools
JUnit	JMeter
Curl	LoadUI
Postman	Grinder
Advanced REST Client	Curl-Loader
Mocha	Wrk
Chai	Vegeta
TestNg	BlazeMeter

Table 3.1: API testing tools [16]

the API's performance, to ensure it does not worsen during development, and maintains its current standards.

Aside from these technologies, the work of Ehsan et al. [17] developed a list of state-of-the-art tools, methodologies and frameworks for API testing. This list is shown in Table 3.2, and will be studied in detail in the next section. The relevant column for this work is the Tools column. Each entry on this table will have its own subsection, in the following section of this document, with the objective of studying the tool, and determining if it is useful or not, to evaluate the API in development.

In summary, this section provides insight into the existing work for two of the great goals of this work: usability and security. The last goal, performance, will be studied in a simpler manner during the remainder of this document, as it is simply required that it is similar to the existing API.

Title	Tool	Approach / Methodology	Framework / Model
Property-based Testing of JSON based Web Services		✓	
Model-driven Testing of RESTful APIs		✓	
RESTful API Automated Test Case Generation	✓		
Study on REST API Test Model Supporting Web Service Integration			✓
Automatic Generation of Test Cases for REST APIs: a Specification-Based Approach		✓	
Metamorphic Testing of RESTful Web APIs		✓	
Automated API Testing	✓		
RESTler: Stateful REST API Fuzzing	✓		
QuickREST: Property-based Test Generation of OpenAPI-Described RESTful APIs	✓	✓	
AI-Driven Web API Testing		✓	
A simple, lightweight framework for testing RESTful services with TTCN-3			✓
Differential Regression Testing for REST APIs		✓	
RETTETGEN: Automated Black-Box Testing of RESTful APIs	✓		
Deep Learning-Based Prediction of Test Input Validity for RESTful APIs		✓	
REStest: Automated Black-Box Testing of RESTful Web APIs			✓
A Black Box Tool for Robustness Testing of REST Services	✓		

Table 3.2: Tools, Methodologies and Frameworks for API Testing [17]

Chapter 4

Security and Performance testing tool analysis and selection

In Section 3.2 a state of the art study was performed, namely in terms of the existing technologies for API testing. A list of potential tools to be used was described. The tools in this list will be analyzed in detail in this Section, and the most appropriate will be chosen. As there is a unit test mechanism already in place, used by the company itself: Rspec in Ruby, the tool I aim to find should be able to perform some sort of randomized test, in order to find edge cases and not before thought cases, to find new bugs and weak points. This is for the security section. For the performance section, a tool to register response times, and response codes will be selected.

4.1 Tool Analysis

4.1.1 JUnit

JUnit is a unit testing framework developed for the Java programming language. It has been a truly important technology in the area of test-driven development, and belongs to a family of unit testing frameworks often referred to as xUnit. In this case, J stands for Java, while the x is a placeholder for the initials of any language a unit testing framework may be developed for, such as CppUnit, for C++, PyUnit, QUnit, etc. Although it is a common and powerful unit testing tool, JUnit allows only to write unit tests, and does not contain any sort of test generation, which is one of the main characteristics necessary for this work. A simple example of a JUnit test can be seen in Listing 4.1

```
1 import static org.junit.jupiter.api.Assertions.assertEquals;
2 import example.util.Calculator;
3 import org.junit.jupiter.api.Test;
4
5 class MyFirstJUnitJupiterTests {
6     private final Calculator calculator = new Calculator();
```

```
7
8     @Test
9     void addition() {
10         assertEquals(2, calculator.add(1, 1));
11     }
12 }
```

Listing 4.1: JUnit test example

This is an example of a simple unit test. An instance of the class to test is created, a method is called, and its output is evaluated against the expected result, using the `assertEquals` JUnit method. Aside from the `assertEquals` there are other assertions, such as `assertTrue`, `assertTimeout` for execution times, `assertTimeoutPreemptively`, etc. Aside from the `@Test` annotation, it is also possible to use annotation such as `BeforeAll`, `BeforeEach`, `AfterAll`, `AfterEach`, among others. These capabilities, combined with conditional test execution, nested tests, and test class inheritance provide a powerful tool, although being only for unit tests, manually written [10].

4.1.2 cURL

Developers utilize the command line tool `cURL`, which stands for client URL, to send and receive data to and from servers. At its most basic level, `cURL` allows you to communicate with a server by indicating the destination (in the form of a URL) and the information you wish to transmit. Almost every platform can run `cURL`, which is multi-protocol and supports HTTP and HTTPS. This makes `cURL` the perfect tool for testing communication from practically any device to the majority of edge devices (as long as it has a command line and network access). Although this tool is not sufficient to perform automated unit tests by itself, it can complement some other tools, and serves for a great debugging tool. As an example, a `cURL` request is performed as follows:

```
1 curl -i -H "Accept: application/json" -H "Content-Type:
   application/json" -X GET http://hostname/resource
```

Listing 4.2: cURL example

Using flags, different resources can be defined, such as `-H` for headers, `-d` (data) for payloads. At the end of the command, the type of request (GET, POST, PATCH, DELETE, or PUT), the hostname and the resource are described. `cURL` is also the software library encompassing these requests, with a number of other features.

Despite this, `cURL` is merely able to perform HTTP and HTTPS requests, and provides no utility in terms of testing [3].

4.1.3 Postman

Postman is a more advanced tool for API requests, and testing. Aside from the features seen before (performing requests and comparing responses with known values), Postman provides some more powerful capabilities. Postman provides a well-constructed GUI, although it also allows for requests to be performed using a command-line, and cURL.

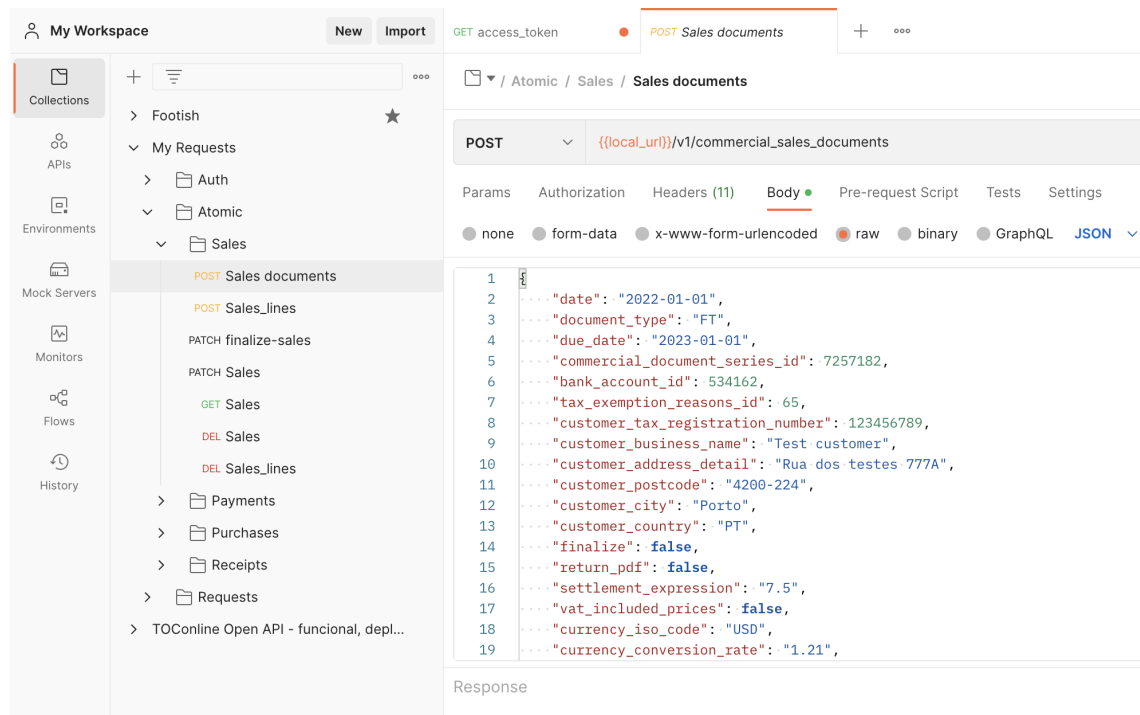


Figure 4.1: Postman GUI

As Figure 4.1 shows, there can be multiple requests saved, and organized into workspaces and folders. For each request, all the information required can be altered using the GUI, such as the request body, headers, request type, etc. This tool allows for pre-request scripts to be written, which can be useful when some request's information depends on previous requests. These can be allocated to a single request, or to a collection of requests. The same goes for tests. A script of tests can be written in JavaScript, and will be executed after a specified request or collection of requests. Despite this, Postman allows for more complex testing. An entire API can be imported from a JSON or YAML, and documentation is automatically generated. Afterwards, test scripts written in JavaScript can be developed, for specific requests, or for the entire API. These tests use the Postman test framework, which allow for similar tests to the JUnit framework. Requests are performed, and assertions about the response are performed. These assertions can be of the response code, the body, etc. Aside from the response, environment variables or global variables, which can be used by the requests in the workspace can have CRUD operations performed on them. This API can then be deployed and monitored. Monitoring can be performed using Postman's specific tools, or by connecting to another tool such as Big Panda, Microsoft Power Automate, etc. It is also possible to create a mock server, using only this JSON or YAML. This allows to simulate

endpoints and their expected responses without having to actually develop the back-end necessary. To help with sequential requests, through the GUI it is possible to define a Flow. This flow allows for the software to send requests, evaluate the responses, follow conditional paths, alter variables, etc. When it comes to authorization, the GUI is prepared to set-up various types of protocols, such as Bearer Token, Basic Auth, Digest Auth, OAuth 1.0, OAuth 2.0, Hawk Authentication, AWS Signature, NTLM Authentication, and Akamai EdgeGrid.

4.1.4 Advanced REST Client

Similarly to Postman, Advanced REST Client, further referred to as ARC, is a request performer and unit tester, with an accompanying GUI. It provides some of the same features, such as sending different requests with all the possible information, unit tests although more rudimentary, workspaces, variables, and API documentation [1]. A snapshot of the GUI of this tool can be seen in Figure 4.2.

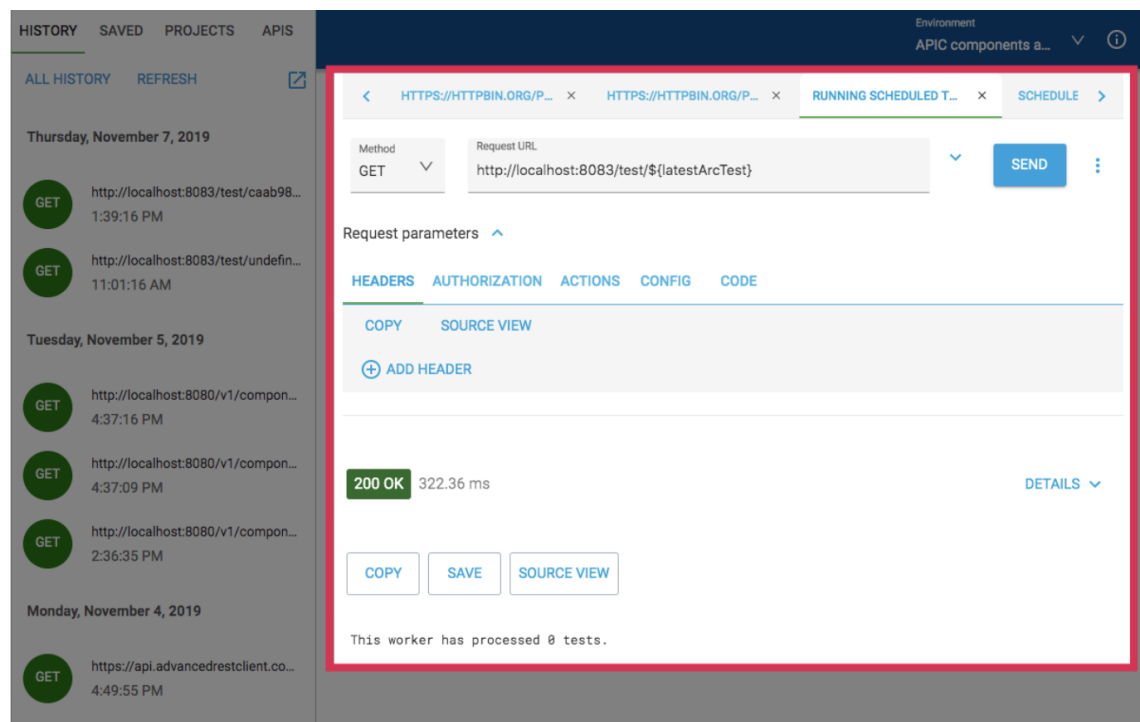


Figure 4.2: Advanced REST Client GUI

4.1.5 Mocha and Chai

Mocha and Chai are two JavaScript test frameworks developed in node.js, commonly used together. These frameworks provide a large number of interesting capabilities. Being developed in JavaScript, it may be easy to integrate with Postman, which would yield a very powerful combination. The Postman capabilities were already mentioned, and aside from these, Mocha and Chai also provide dynamically generated tests, asynchronous code, inclusive/exclusive/pending/retry tests, etc. [2, 5].

The most important feature would be the dynamically generated tests. Despite this, some tools designed specifically for this use case will be detailed shortly, and will be more powerful in doing so.

4.1.6 TestNg

TestNg unit testing framework, developed for the Java programming language. It is heavily inspired, and based off of JUnit and NUnit. This tool offers some new functionalities, namely [6]:

- Annotations.
- Run your tests in arbitrarily big thread pools with various policies available (all methods in their own thread, one thread per test class, etc...).
- Test that your code is multithread safe.
- Flexible test configuration.
- Support for data-driven testing (with @DataProvider).
- Support for parameters.
- Powerful execution model (no more TestSuite).
- Supported by a variety of tools and plug-ins (Eclipse, IDEA, Maven, etc...).
- Embeds BeanShell for further flexibility.
- Default JDK functions for runtime and logging (no dependencies).
- Dependent methods for application server testing.

4.1.7 EvoMaster: Automated API Testing

As it was seen in Section 3.2, it is essential to have automatically generated tests, in order to reach as many test cases as possible. This is necessary in order to be more confident of the tool security. For this, the tool to be analyzed at this point allows for automated test generation. This work [8] presents a new technique for generating new unit tests through an evolutionary algorithm, rewarded with code coverage and fault finding metrics. This tool was created for the developers of the application to test, which is how it is able to measure code coverage. The tool developed using this technique is called EvoMaster, which is open source, and was developed in java [8]. In the experimental section of the work, some tests were manually developed, and measured against the tool's automated tests using a code coverage metric. The results are as follows.

As it is shown in Table 4.1, this tool does not appear to be ideal to create new tests automatically, since its code coverage results are lower than the manual tests provided.

SUT	Coverage	Manual Coverage
FeaturesService	41%	82%
Industrial	18%	47%
ScoutAPI	20%	43%

Table 4.1: Statement coverage results for the generated tests compared to the ones obtained by the already existing, manually written tests. [8]

4.1.8 Automated API Testing

This work presents a new tool to automate API testing. It addresses some limitations of other existing tools, such as the lack of support for sequenced API calls, scheduled tests, code-less testing, etc. It is able to include all of these features, listed below [26].

- Code-less API Testing.
- Sequencing API calls.
- Unpredictable JSON response handling.
- Parameter dependency.
- Scheduled tests.
- Parallel API execution preserving orderliness.

Despite these features, the tool is not capable of generating test cases, which is the main feature needed for the testing of the developing API.

4.1.9 RESTler: Stateful REST API Fuzzing

RESTler is the first stateful REST API fuzzer. This tool analyzes a Swagger specification, which is ideal for this use-case, as one of the first tasks performed for the practical component of this thesis was to generate a tool that automatically generates Swagger (now renamed OpenAPI) documentation. After analyzing the specification, it generates and executes tests in a stateful manner [9]. This tool has been actively maintained since its launch in 2019, and even at the time of publication, this tool had already found 28 bugs in GitLab. The experiments ran in said work show a great code coverage, in a feasible amount of time, which are greatly detailed in the work's results.

4.1.10 QuickREST: Property-based Test Generation of OpenAPI-Described RESTful APIs

Similarly to the previous work studied [9], this work presents a tool that analyzes an API's specification through the OpenAPI format [27]. This tool provides many of the same features, such as stateful sequences, automatic random test generation (fuzzing), among others. Despite this, the tool is not available for use at the moment, and thus cannot be used for the purpose at hand.

4.2 Tool Selection

The study performed in Section 3.2 provided a great list of important features a testing tool should have, as well as a list of state-of-the-art technologies and works that may implement these features, and serve an important role for this work. Out of all of the tools found in that section, and analyzed in Section 4.1, the ideal one appear to be RESTler and Postman. Postman provides a great GUI for the entire management of the API. It will allow the entire collection to be stored in an independent environment, after being imported from an OpenAPI file which, as it has been described before, is being automatically generated by a devised program. Given the fact that the API's performance is to be measured, although not to a very detailed and exhaustive manner, Postman is again a great choice as it provides a monitoring feature, which is simple to use, and will yield the necessary results for this work. RESTler will allow for the remainder of the tests to be executed. It will add to the unit tests developed using Ruby's Specs by generating pseudo-random inputs for the API's routes, in order to ensure everything is protected.

Chapter 5

Developing and implementing a new API

In this chapter, the majority of the practical work developed for this project will be detailed. Initially, the improvements made to the existing API are described. Lastly, the new API is described, along with all of the improvements in regard to the existing version, as well as with all of the external tools developed to aid its usage.

5.1 Documenting the existing API

When it comes to usability, it is of utmost importance to make the API usage as simple as possible, while retaining all of its functionalities, security, and performance, and turning it easy to use. As a first attempt to improve the usability of the API, I developed an extensive documentation. This documentation started by a Ruby program consisting of several different components which scrape all the information available about the API, and generate documentation. The first component is essentially a YAML parser, which reads all of the information available in the API routes' YAML file. Among other information, these YAML files contain the following information, relevant to the documentation, present in Listings 5.1 and 5.2.

The first portion of the YAML file contains the information about the attributes needed for the document header, and can be seen in Listing 5.1

```
1 # [public]
2   # [group=Customers]
3   # Customers
4   customers:
5     pg-table: "customers"
6     pg-condition: "customers.is_deleted = false"
7     request-sharded-schema: true
8     request-table-prefix: false
9     pg-company-column: company_id
10    attributes:
```

```
11      # Document unique identifier
12      # - id
13      # Customer tax registration number (NIF) (ex:
14          5555555550)
15      - tax_registration_number
16      # Business name (ex: Test customer)
17      - business_name
18      # Main contact name (ex: Mr. Test)
19      - contact_name
20      # Customer website url (ex: http://www.testurl.pt)
21      - website
22      # Customer phone number (ex: 2299999999)
23      - phone_number
24      # Customer mobile number (ex: 9299999999)
25      - mobile_number
26      # Customer e-mail address (ex: test@testcustomer.pt)
27      - email
28      # Notes about the customer to be added to each
29          document (ex: This is only a test)
30      - observations
31      # Internal notes about the customer (ex: This is
32          good customer)
33      - internal_observations
34      # Indicates is a company or a final customer (Default
35          true - Company)
36      - not_final_customer
37      # Indicates if is a cashed VAT customer
38      - cashed_vat
39      # Customer tax country region (ISO 3166 1-alpha-2) +
40          'PT-AC' + 'PT-MA' (Used for VAT tax suggestion) (
41          ex: PT-MA)
42      - tax_country_region
43      # Customer country (ISO 3166 1-alpha-2) + 'PT-AC' + '
44          PT-MA'
```

Listing 5.1: Customers API route YAML specification file header

As these documents do not exist on their own, and relate to other tables, having more information stored in them, it is necessary to describe these relations. This information is stored in these YAML files, as seen in Listing 5.2.

```
1      to-one:
2      -
3        company:
4          resource: current_company
5      -
6        main_address:
7          pg-table: "addresses"
8          request-sharded-schema: true
9          pg-parent-id: "addressable_id"
10         pg-child-id: "id"
11         pg-condition: "addressable_type='Customer' and
12                       is_primary=true"
13         resource: "addresses"
14      -
15      # Relation to many customer addresses
16      main_email_address:
17        pg-table: "email_addresses"
18        request-sharded-schema: true
19        pg-parent-id: "email_addressable_id"
20        pg-child-id: "id"
21        pg-condition: "email_addressable_type='Customer'
22                      and is_primary=true"
23        resource: "email_addresses"
24      -
25      tax_exemption_reason:
26        resource: "tax_exemption_reasons"
27      -
28      defaults:
29        pg-table: "customer_defaults"
30        request-sharded-schema: true
31        pg-parent-id: "customer_id"
32        pg-child-id: "id"
33        resource: "customers_defaults"
```



```

32     to-many:
33         -
34             # Relation to many customer addresses
35             addresses:
36                 pg-table: "addresses"
37                 request-sharded-schema: true
38                 pg-parent-id: "addressable_id"
39                 pg-child-id: "id"
40                 pg-condition: "addressable_type='Customer'"
41                 resource: "addresses"
42         -
43             # Relation to many customer addresses
44             email_addresses:
45                 pg-table: "email_addresses"
46                 request-sharded-schema: true
47                 pg-parent-id: "email_addressable_id"
48                 pg-child-id: "id"
49                 pg-condition: "email_addressable_type='Customer'"
50                 resource: "email_addresses"

```

Listing 5.2: Customers API route YAML specification file relations

This contains the information parsed by the script: the attributes required in the request body, as well as the different relationships (to-one and to-many) which are a JSONAPI convention for relations to other resources.

The next component connects to the database, and parses all of the information relative to the different existing tables, as well as its attributes. It then crosses this information with the information parsed from the previous component, and creates a JSON OpenAPI file describing the entirety of the API in a standardized format. In order to output this information as a YAML file (which according to the OpenAPI documentation allows for reusability of components and better readability) another script was devised, which looks for common payloads and attribute lists, aggregates them in a reusable component, and outputs the result as a YAML file, which ends up being much cleaner and more elegant, as shown on the Listing 5.3.

```

1  "/commercial_sales_receipt_lines":
2      get:
3          responses:
4              '200':
5                  description: OK

```

```

6      content:
7      application/json:
8      schema:
9      "$ref": "#/components/schemas/
        commercial_sales_receipt_lines-response"

```

Listing 5.3: YAML API description example

After the YAML was generated, I uploaded it to Swagger, and made it available to the users of the API, providing a concise and easy-to-use documentation. Part of the documentation of a specific route of the API is shown in Figure 5.1. The entirety of this automatically generated documentation is available at: <https://app.swaggerhub.com/apis/cloudware-deploy/CWApi/1.0.0>. It should be noted that this documentation is written in Portuguese, due to a company decision, which is based on the fact that the company and its clients speak this language.

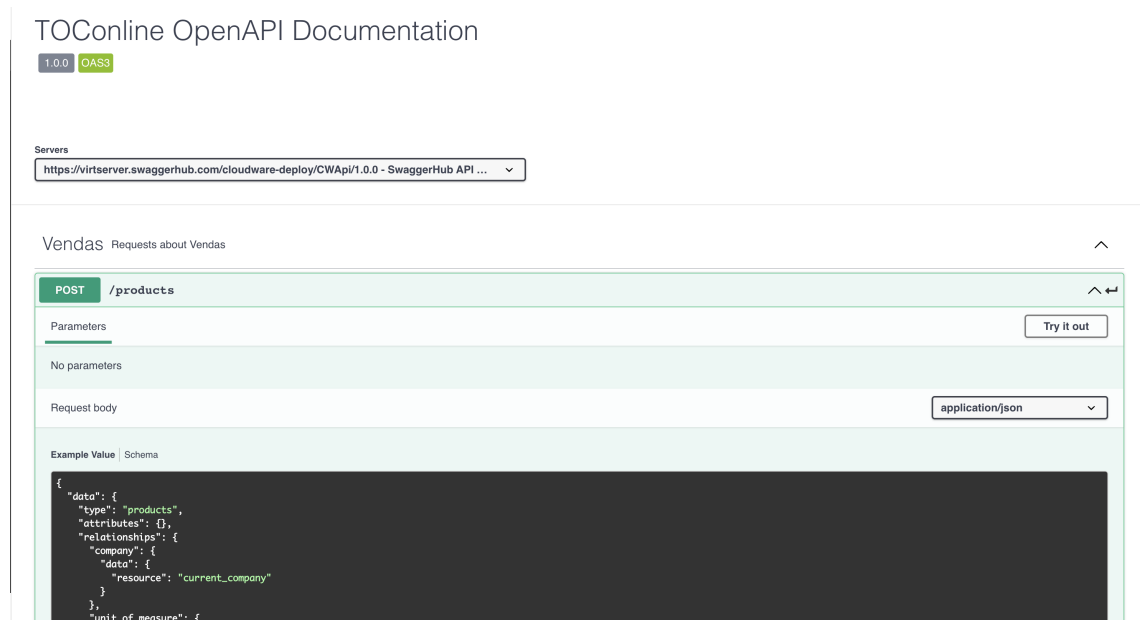


Figure 5.1: Swagger API documentation

Afterwards, I developed a more detailed and descriptive documentation. This describes all of the endpoints of the API, written in the form of a story, with the objective of facilitating its reading. The first few pages of the documentation describe the authentication. One page describes it succinctly, in a simple manner. The second page describes the authentication in a more detailed approach, showing every error that may occur, if the authentication is not done properly. A portion of this page can be seen in Figure 5.2. The entirety of this documentation is available to the clients, and can be consulted through the following link: <https://cloudware.gitbook.io/documentacao-api>.

The next page describes the usage of the provided Postman example.

Another component of the aforementioned Ruby program is responsible for creating a JSON file able to be imported to Postman. This component uses the same YAML fed to Swagger. When

Documentação API

Introdução
Setup
Autenticação
Autenticação simplificada
Caraterísticas dos pedidos
API-V1
Introdução à API v1
Faturas
Recibos
Compras
Pagamentos
API-V0
Faturas
Clientes e Moradas
Fornecedores
Produtos e serviços
Recibos
Compras
Pagamentos
Descarregar PDF documentos
Envio de documentos por email
Anexar ficheiros
Comunicação de documentos à AT

Passo 3: Acesso autenticado à API comercial, usando o "access_token" obtido de 2.2 e guardado

Todos os pedidos à API, cujos endereços são os documentados em <https://cloudware.gitbook.io/documentacao-api>, são feitos para o endereço "API_URL" seguido do nome do recurso. Além disso, todos os pedidos deverão conter obrigatoriamente os seguintes headers:

- Content-Type: application/vnd.api+json
- Accept: application/json
- Authorization: o texto "Bearer", seguido dum espaço, seguido do "access_token" obtido do passo 2.2.

Os parâmetros adicionais a passar na query (no caso de GET) ou no body dos pedidos são os especificados pelo padrão JSONAPI, a que a nossa API obedece, e que estão documentados em jsonapi.org.

Como exemplo, indica-se um pedido GET ao recurso "commercial_sales_documents" (documentos de venda), paginado para devolver apenas os 5 primeiros registos.

```
curl -v -H 'Content-Type: application/vnd.api+json' \
-H 'Accept: application/json' \
-H 'Authorization: Bearer <access_token>' \
'<API_URL>/commercial_sales_documents?page[size]=5'
```

A resposta esperada, neste caso, é como a seguinte:

```
GET /commercial_sales_documents?page[size]=5 HTTP/1.1
Content-Type: application/vnd.api+json
Accept: application/json
Authorization: Bearer <access_token>
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json; charset=utf-8
{"data":[{"type":"commercial_sales_documents","id":"...", "attributes": {...}}, ...]}
```

Ou seja, e se não ocorrer nenhum erro (como o tentar uma consulta a algo que não existe, ou criar um registo com valores incorrectos), a resposta esperada é um status 200 (OK) e um JSON, no formato JSONAPI, contendo o registo ou os registos devolvidos (no caso dos GET), criados (POST) ou alterados (PATCH). Como se disse no passo 2.2, se o pedido devolver um status 401 (Unauthorized), provavelmente o "access_token" já expirou, e terá que ser actualizado, ou pedido um novo efectuando novamente o passo 2.

Powered By GitBook

Figure 5.2: Authentication page snippet

imported to Postman, this file creates a workspace with all the existing API routes correctly described and set-up, as well as the OAuth 2.0 security protocol - thus, allowing for an easy way for a new customer to experiment with the API, which was not possible before. An example of one of the requests made available is displayed in Figure 5.4. Initially, the user had to consult some information, available in their profile in TOCOnline, and then input it in Postman. This was not simple enough for the end-user, as some dozen e-mails were sent claiming they could not complete this step. Thus, I developed a Ruby script to provide the Postman file with the specific data for each user. Thanks to this new program, it is possible to perform the authentication, and any request to the API, with a single button click. This process is performed using the menu visible on Figure 5.3.

Before the development of this Postman feature, in order for a client to use the API for the first time, they had to consult multiple authentication credentials in their personal page, create a Postman workspace, follow some pages of instructions to create their sandbox environment, and only then could they perform their first request. Once again, this process was error prone, and lead to a great number of hours spent by the company team helping their clients. Now, the client must simply download an automatically generated file, open Postman, click the 'Authenticate' button, and click the 'Send Request' button on any request they wish to perform.

The following pages describe each of the possible requests. Each page contains some descriptive

The screenshot shows the Postman interface for configuring an OAuth 2.0 authorization method. The collection is named 'TOConline Open API'. The 'Authorization' tab is selected, showing a description: 'This authorization method will be used for every request in this collection. You can override this by specifying one in the request.' The configuration includes:

- Type:** OAuth 2.0
- Add auth data to:** Request Headers
- Current Token:** A section explaining that the access token is only available to the user and can be synced for collaborators. It shows an 'Access Token' field with a value '8-459-552-02d11d358b29a8...' and a 'Header Prefix' set to 'Bearer'.
- Configure New Token:** A section with two tabs: 'Configuration Options' (selected) and 'Advanced Options'. It includes fields for:
 - Token Name:** bearer
 - Grant Type:** Authorization Code
 - Callback URL:** https://oauth.pstmn.io/v1/callback
 - Auth URL:** {{base_url}}/oauth/auth
 - Access Token URL:** {{base_url}}/oauth/token
 - Client ID:** {{client_id}}
 - Client Secret:** {{client_secret}}
 - Scope:** (partially visible)

The bottom of the interface shows a sidebar with icons for 'Online', 'Find and Replace', and 'Console'.

Figure 5.3: Postman OAuth as imported by the script-generated YAML

text, request examples, as well as automatically generated request documentation, imported from the mentioned Swagger. A section of one of these pages can be seen in Figure 5.5.

POST

▼

{{base_url}}/api/commercial_document_series

Params

Authorization

Headers (8)


Body

Pre-request Script

Tests

Settings

Headers

 Hide auto-generated headers









	KEY	VALUE
<input checked="" type="checkbox"/>	Authorization	<div> Bearer 8-459-552-02d11d358b29a864ea15126a0c396e6b0e3...</div>
<input checked="" type="checkbox"/>	Postman-Token	<div> <calculated when request is sent></div>
<input checked="" type="checkbox"/>	Content-Length	<div> 0</div>
<input checked="" type="checkbox"/>	Host	<div> <calculated when request is sent></div>
<input checked="" type="checkbox"/>	User-Agent	<div> PostmanRuntime/7.29.2</div>
<input checked="" type="checkbox"/>	Accept	<div> */*</div>
<input checked="" type="checkbox"/>	Accept-Encoding	<div> gzip, deflate, br</div>
<input checked="" type="checkbox"/>	Connection	<div> keep-alive</div>
	Key	Value

Figure 5.4: Postman request with all the necessary headers, body, and parameters, imported by the same YAML

5.2 A new design for the API

Although all this new documentation has reduced the number of help requests received, this was merely a quick fix, which bought the company time to develop the new API.

In terms of actual improvements to the API itself, the first suggestion I made to this simplification was to abandon the JSONAPI payload convention structure, and use a simpler and cleaner structure. As seen in the previous chapter, the JSONAPI has a payload convention with a high level of nesting, as well as a large number of redundant attributes. This, of course, has a reason. Using this structure, the JSONAPI framework will handle the majority of the request's work: it will sanitize inputs, check for malicious content, and access the database tables required, with no additional efforts needed [44]. For any given request, the SQL line in Listing 5.4 should be executed, and everything will be taken care of, namely input sanitization.

```
1 SELECT * from public.jsonapi(api_method, api_route, payload);
```

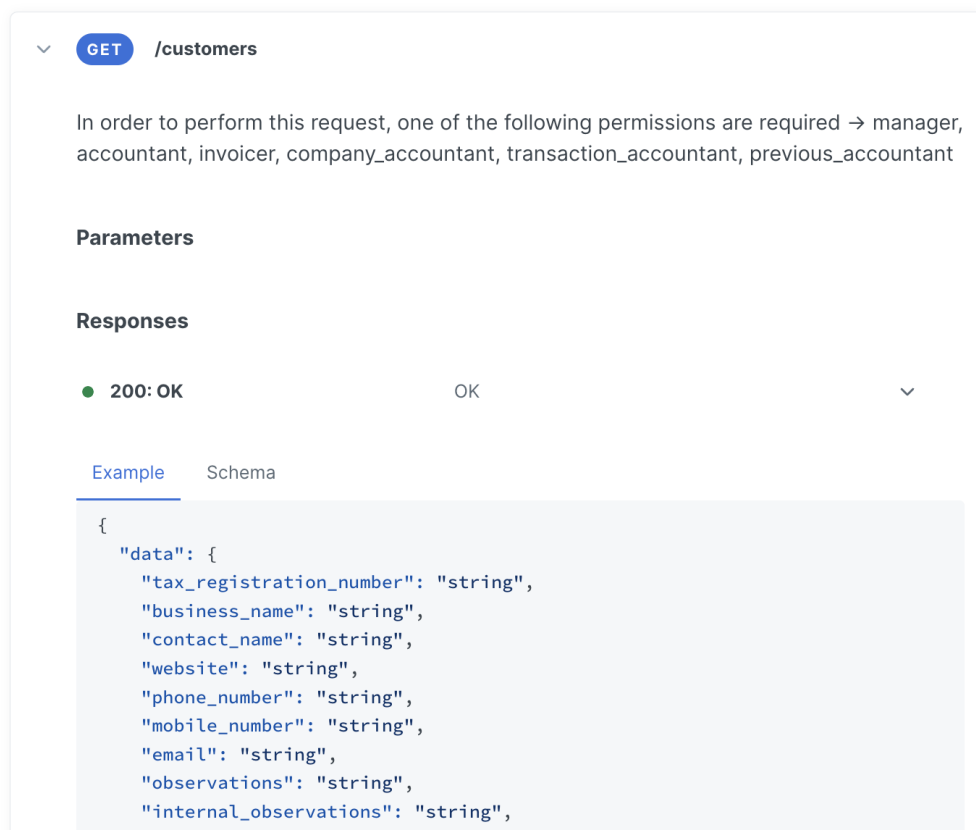
Listing 5.4: JSONAPI SQL code

All of this information is in the request itself, and does not need parsing. Should any additional handling be done, it can still be performed in the code directly before, or after this SQL function is called, or by the use of triggers. In the case of the TOCOnline API, all of these methods are used. There is additional handling being performed before and after this function executes, in the Ruby code that executes it, and there are triggers in the database, running before the insertion/updating of the tables involved, and afterwards as well. Despite these advantages, the payload necessary for the

Clientes e Moradas

As rotas definidas no presente capítulo permitem gerir toda a informação relativa aos clientes associados a uma dada empresa: obter informação sobre clientes, editá-la, eliminar, e criar novos

Neste primeiro exemplo, ao realizar um pedido para a rota `https://apiv1.toconline.com/customers` irá receber uma resposta semelhante à descrita de seguida, com uma lista de elementos 'data', onde cada elemento corresponde a um cliente associado à sua empresa. Esta rota não requer qualquer tipo de parâmetros.



The screenshot shows a documentation page for the `GET /customers` endpoint. It includes a dropdown menu with 'GET' selected. Below the endpoint name, it states: 'In order to perform this request, one of the following permissions are required → manager, accountant, invoicer, company_accountant, transaction_accountant, previous_accountant'. There are sections for 'Parameters' and 'Responses'. Under 'Responses', there is a status '200: OK' with an 'OK' label and a dropdown arrow. Below this, there are tabs for 'Example' and 'Schema'. The 'Example' tab is active, showing a JSON response structure:

```
{
  "data": {
    "tax_registration_number": "string",
    "business_name": "string",
    "contact_name": "string",
    "website": "string",
    "phone_number": "string",
    "mobile_number": "string",
    "email": "string",
    "observations": "string",
    "internal_observations": "string",
    "..."
  }
}
```

Figure 5.5: Snippet of a documentation page, describing the customers route

client to create and send is still overly complicated. The JSONAPI payload structure as described in the official documentation is described in Listing 5.5.

```
1 {
2   data: {
3     type: route_name,
4     attributes: {
5       attributeA: valueA,
```

```
6     attributeB: valueB,  
7     attributeC: valueC,  
8     attributeD: valueD  
9 },  
10    relationships: {  
11      relationshipA: {  
12        data: {  
13          type: relationshipA,  
14          id: idA  
15        }  
16      },  
17      relationshipB: {  
18        data: {  
19          type: relationshipB,  
20          id: idB  
21        }  
22      },  
23      relationshipC: {  
24        data: {  
25          type: relationshipC,  
26          id: idC  
27        }  
28      }  
29    }  
30  }  
31 }
```

Listing 5.5: Original JSONAPI payload structure

In order to simplify this payload, there will be two major improvements: remove excessive nestings, and remove redundant attributes. There will no longer exist attributes with a high level of nesting, as was the case with `idA`, `idB`, and `idC`, in Listing 5.5. Additionally, there will no longer exist redundant attributes, such as the `'type'` attributes, in lines 13, 19, and 25, in the same Listing.

With this new payload structure, shown in Listing 5.6, I believe it will be much easier for clients to read, and create their own payloads, which will hopefully make the API easier to use, and thus diminish the help requests received.

```
1 {  
2   attributeA: valueA,  
3   attributeB: valueB,
```

```
4     attributeC: valueC,  
5     attributeD: valueD,  
6     relationshipAid: idA,  
7     relationshipBid: idB,  
8     relationshipCid: idC  
9 }
```

Listing 5.6: New payload structure

The second improvement in terms of usability will come by reducing the number of requests necessary to create a document. As it was detailed in Section 2.1, to create any sort of document, being a bill, payment document, purchase document, or purchase payment, at least three requests are necessary: the first type, to create the document header; the second type, to send each of the document lines; and the last type, to finalize the document. The new structure, which will henceforth be referred to as API v1, allows for any type of document to be created with a single request. By joining these two improvements together, in order to create a sales document, for example, only one request will have to be performed, to the `v1_commercial_sales_documents` route, using the following payload, in Listing 5.7.

```
1 {  
2     "date": "15-7-2022",  
3     "document_type": "FT",  
4     "due_date": "2020-02-01",  
5     "finalize": true,  
6     "return_pdf": true,  
7     "customer_tax_registration_number": 999999990,  
8     "settlement_expression": "7.5",  
9     "vat_included_prices": false,  
10    "currency_iso_code": "USD",  
11    "currency_conversion_rate": "1.21",  
12    "notes": "Document notes",  
13    "external_reference": "External document reference",  
14    "payment_mechanism": "MO",  
15    "commercial_document_series_id": 0,  
16    "bank_account_id": 0,  
17    "tax_exemption_reasons_id": 0,  
18    "lines": [  
19        {  
20            "item_code": "string",  
21            "item_type": "string",
```



```
22         "quantity": 0,  
23         "unit_price": 0,  
24         "description": "string",  
25         "settlement_expression": "string",  
26         "unit_of_measure_id": 0  
27     }  
28 ]  
29 }
```

Listing 5.7: New sales document payload

Now, with only one request, and a much simpler payload, it is possible to create any document. The data that before was sent in the header request, is present in the beginning of the payload, with the new structure. The information in each of the previously necessary line requests is now present in the "lines" array, in line 18. The "finalize" attribute, in line 5, replaces the finalization route, meaning the document will be created, and instantly finalized, if specified. Lastly, the `url_for_print` route, which was previously necessary to obtain the PDF of a generated document was also replaced, in this case by the `return_pdf` attribute, in line 6, which if set will return a link to the document PDF, as a return to the API request.

Thus, what was before comprised of at least 4 requests (create header, create each line, finalize, return_pdf), is now condensed into a single request. This largely helps its usability from the user's perspective, and removes a lot of strain from the servers responsible for handling these requests, when it comes to the number of requests that are received, and the number of responses calculated and sent. The link to the PDF is now direct and can be immediately accessed, and no string operations are necessary on several fields, as previously. This principle was applied to every single document route. Thus, I developed four new routes : `v1_commercial_sales_documents`, `v1_commercial_sales_receipts`, `v1_commercial_purchases_documents`, and at last `v1_commercial_purchases_payments`. For each, I also developed a `v1_finalize` and a `v1_void` route, if the user does not wish to finalize or void the document at the time of its creation, and wishes to do so later. For each of these four types of documents, I also developed some helper routes for the remaining CRUD operations: POST line, DELETE line, GET lines, DELETE document, and PATCH document. This resulted in a total of 26 new routes. Although some of these routes already existed, they are now independent of the JSONAPI used in the old requests. The possibility of creating a document with a single request is much more powerful, but comes with its limitations. It should be possible to perform CRUD operations in the newly created document. This was already possible in the previous version of the API, but in order to maintain consistency between the requests, I developed new helper requests. The first request allows a user to add new lines to an existing document, by performing a POST request to `API_URL/v1_commercial_sales_document_lines/id`, with the payload of Listing 5.8.

```
1 {
2   "lines": [
3     {
4       "payable_type": "Purchases::Document",
5       "payable_id": 940972,
6       "paid_value": 10.21,
7       "settlement_percentage": 5
8     },
9     {
10      "payable_type": "Purchases::Document",
11      "payable_id": 940973,
12      "paid_value": 1.34,
13      "settlement_percentage": 5
14    }
15  ]
16 }
```

Listing 5.8: New add lines request

This request follows the same structure as the request described in Listing 5.7, although this request only contains the `lines` section of the request.

The next request allows a user to edit all of the information about an existing document, as well as all of the information of one of its lines, in a single request. This request should be performed as a `PATCH` to `API_URL/v1_commercial_purchases_payments/id`, using the same payload as the `POST` request for the document creation. All of the parameters are optional, except the identification parameters, and when specified, will substitute all of the information stored at the moment of the request. In order to retrieve information regarding a specific document, a `GET` request can be performed to `API_URL/v1_commercial_purchases_payments/id`. Lastly, in order to delete an entire document, or a specific line in a document, a `DELETE` request can be performed to `API_URL/v1_commercial_purchases_payments/id`, or

`API_URL/v1_commercial_purchases_payment_lines/id`, respectively.

5.3 Implementating the new API

In summary, with much simpler payloads, and workflows, I developed 28 new routes. The main difficulties encountered by the clients when first using this API, as described in Chapter 2 were all mitigated. The authentication process was simplified by the usage of the automatically generated Postman file, the payloads are much simpler, and for each action the API enables, a smaller number of requests is needed. An adapter for the existing routes which were not yet implemented was also developed. This adapter transforms the new structure in requests to the old API in order to standardize the API and make the new version as complete as the old one. The implementation of

these routes was not trivial, at all. The creation of this new API served a second purpose, which was to extend the usage of a new technological stack. As it is detailed in Chapter 8, this decision had a slight effect in the application's performance. This however, was expected, and was a company decision, outside the scope of this work.

The requests to the original API follow a path which is different from the new version. The first step is common to both versions, which is the gatekeeper. This is an ERB file, which is essentially a Ruby templating language. This file acts, as the name implies, as a gatekeeper. It receives a request and checks if the route is valid, if the user has permissions to use said route, and decides if it should return a 403 forbidden code, or if it should send the request to the appropriate handler.

```
1 {  
2   "methods": ["GET"],  
3   "expr": "^/v1/commercial_sales_documents(.*?)?",  
4   "module_mask": "0x0",  
5   "role_mask": "0x4041e",  
6   "job": {  
7     "tube": "v1-document-ops",  
8     "methods": ["GET"]  
9   }  
10 }
```

Listing 5.9: Example of gatekeeper entry

The example in Listing 5.9 shows the entry for the GET request to the sales documents route. This allows for a user to perform a GET request to the `/v1/commercial_sales_documents` route, which is only available to users with the `role_mask` specified, and will forward the request to the `v1-document-ops` tube. A tube is merely a file responsible for handling back-end requests. In the old version this request would redirect directly to the JSONAPI, written in C++, which would then reply. In the new version, Ruby handles the request, in this case, in the `v1-document-ops.rb` file. This document then parses, and validates the data sent in the request. If everything is valid, it will then communicate with the back-end. In order to do this, a database API was developed: `CDB_API`, in which CDB stands for Central Database, one of the various databases in the project. This API then communicates with a C++ gatekeeper, which redirects the request to a Ruby file which handles the communication to the database. This process allows for a standardized and controlled access to the CDB, which is very important and delicate, while facilitating the accesses to the database.

Chapter 6

Studying and Improving the Application's Security

At this point, the application is built, and is completely functional. It is now important to assure its security. This section will describe the security analysis performed on the new API when it comes to fuzzing, malformed payload injection, and malicious content injection, among others. These all depend on the data sent to the API. To test all the possible scenarios, unit tests and automatically generated tests were performed. Using the information obtained in Chapter 3, a list of potential technologies, frameworks, and methodologies was created, and described in Section 3.2. Using this information, the most appropriate technologies were selected in order to test the current API, and the new API version, in terms of security and performance. This study is performed in Section 4.1, and concluded that Postman would be used for the overall management of the API, as well as performance monitoring. To complement this, RESTler will be used to perform pseudo-random requests to the API, in order to test it for the first four types of attacks in this section. Both of these technologies will complement the tool already in place, in the company's technological stack, Rspec. This is a unit testing tool, which was used for the initial unit tests.

6.1 Unit Testing

To begin the security testing, some unit tests were devised. As the research performed in Section 2.2 shows, this is one of the most fundamental and important steps in API testing. As unit tests are rather simple to execute, the tool used for this step was Ruby On Rails' Rspec. As it was mentioned previously, the software in which this API is encompassed is based on this framework, and thus Rspec has been set up previously and has already been used to test several components of this software. Specs were developed for each of the new routes, and these attempt to cover every possible way each route can be executed. As TDD is one of the most widely used and agreed-upon as ideal for software development, these tests were devised prior to the implementation of the routes. As the TDD methodology suggests, each spec, representing a unit test, was detailed with a specific feature of each route in mind. Afterwards, the implementation of the routes started, and after each

feature was implemented, its respective test was added to the pool of tests to run, and the entire pool was reran.

In the end, there was a large number of tests performed. These follow the structure detailed in Listing 6.1.

```
1      it "POST    - EXPECTED: '200 OK' do
2        json, document_id = expect_post(
3          'commercial_sales_documents',
4          attributes: {
5            document_type: 'FT',
6            customer_tax_registration_number: '8888888880',
7            customer_business_name: 'Service partner',
8            notes: 'RC test',
9            vat_included_prices: false,
10           settlement_expression: '10'
11          }
12        )
13        new_customer_id = get_relationship_id(json, :customer)
14      end
```

Listing 6.1: Ruby rspec example

I devised a collection of about 50 unit tests, written manually, for each document type. As it was mentioned earlier, these were developed alongside the development of the routes themselves, to ensure their behaviour was as intended. For example, for the receipts, initially some tests were devised, which simply created the header with different types of information: different `document_type`, `settlement_expression`, etc., or even missing attributes, either being mandatory (in which case we expect an error to be returned), or optional attributes. In order to verify their correct creation, the tests used the GET route, which returns all of the available information about the document. Afterwards, in order to test the `lines` route, more tests were devised, which added lines after a document was created, and later verified if the line was added correctly. In order to test the alteration of the document itself, tests focused on the PATCH routes were written, which check if a document was correctly altered after these routes have been executed. Lastly, in order to test the DELETE routes, the last tests were written, which verify if the routes delete either the lines, or the entire document correctly.

6.2 Fuzzing

These unit tests were then fed to RESTler in order for it to fuzz the inputs and find new possible bugs and security flaws. API fuzzing essentially consists of sending randomized parameters to

a specific program. The principle of these attacks come from the fact that the program may not be prepared to handle every type of input it can receive, and can thus be exploited, when the appropriate input is found. Fuzzing can be divided in three different categories, depending on how the program generates the inputs to send [41].

- Generation-based or mutation-based depending on whether inputs are generated from scratch or by modifying existing inputs.
- dumb (unstructured) or smart (structured) depending on whether it is aware of input structure.
- white-, grey-, or black-box, depending on whether it is aware of program structure.

As there are some API fuzzing attacks that learn from the output given for previous attempts, an API should not reveal any information that is not essential for its usage, when it encounters an error. Such information includes data structure, database query, file system information, etc.

The tool chosen, RESTler, is characterized by being a mutation and generation-based, smart, black-box program. As described above, it is mutation-based since it devises its inputs by either mutating given inputs, or by generating them from scratch; it is smart, as it knows the supposed structure for the inputs; and it is black-box as it has no information regarding the program's structure.

The first step was to provide RESTler with the Swagger file, generated previously, detailing all of the routes of the API. Using this, RESTler compiles a `grammar.py` file, which will then be used to generate the tests. In order to adapt RESTler, and give it all the tools needed to provide the best possible tests for this API, some modifications to the program had to be done, namely in this grammar file. As it was previously detailed, RESTler has a mutation-based input generation component. Thus, it is necessary to provide it with the information regarding the characteristics of the parameters. The first information to provide it is whether a parameter is "fuzzable" and can be altered, or if it shouldn't be altered, and should instead be considered as static. A good example of a static argument is a predetermined header. The API expects a header to indicate a `Content-Type`, and it always expects it to be of the value `application/json`. This distinction is performed according to the example below, in Listing 6.2.

```
1     primitives.restler_static_string("Content-Type: application/  
    json"),  
2     primitives.restler_fuzzable_string("fuzzstring", quoted=True,  
    examples=["2020-02-01"])
```

Listing 6.2: Defining RESTler strings

For the static strings, the value to input should be defined. For the "fuzzable" strings, some parameters should be detailed, such as an initial seed for the fuzzing, an option to have the parameter value quoted, and some examples to feed the fuzzer. Aside from these two, RESTler also

recognizes authentication routes and parameters. It will automatically perform a request to obtain an authentication token, and later provide it to the API, before each request.

With this step completed, it is possible to test the grammar, in order to verify if RESTler can connect to the API, and perform the requests correctly. This is referred to as a smoke test. During the setup of the tool, I was in this loop for a few hours, trying to fine-tune the tool. I experimented with the different options for the parameters, ran the smoke test, analyzed the output logs, and looped back to the parameter tuning. Once the results were as expected, and the parameters were set up as needed, the actual fuzzing process began. RESTler, intelligently, has another quick initial test. Instead of immediately running all of the different possibilities, and find all of the bugs and security flaws at once, it provides a `fuzz-lean` mode, which runs each method for each route in the grammar once, using a default set of checkers to see if it can find a bug quickly. During this first phase, various bugs and security flaws were discovered. Lastly, I used the `fuzz` mode, which performs a smart breadth-first search, trying to find more hidden, and specific edge-cases, and also find resource leaks, back-end corruptions, among others. All the problems found were related to the handling of unexpected inputs, such as customer details relative to customers, or suppliers, who did not yet exist, wrong currencies, payments referencing nonexistent receipts, wrong types of inputs, such as integers instead of strings, etc. All these problems were found thanks to RESTler being able to generate inputs that were not thought of in the previous testing phase. Having found these, a solution was devised, and after running RESTler once more, there were new, similar problems, and some that were not completely corrected. Thus, afterwards, my work consisted of this loop of fixing vulnerabilities, and finding new ones to fix. Some of these were also added to the previously developed unit test pool, as they are faster and easier to run with the remainder of the company's stack. The solution to these vulnerabilities consisted in two distinct adjustments: assuring the correct type of input is provided for each field, and verifying if all of the fields provided reference existing data. In both cases, when there is a wrong input, a clear error message, and the appropriate error code are provided, according to the standard error messages, of the HyperText Transfer Protocol [19].

6.3 Malformed Payload Injection

The next type of attack to study is the Malformed Payload Injection. As the name implies, this attack consists of sending a payload, which is different from the correct payload for the given request, with the hope that the API is not prepared to deal with it. The most common malformations can be as follows.

- Very large payloads
- Very large parameter names, or values
- Incorrect payload structure
- Large number of nested elements

This type of attack was also studied using the methods described in Section 6.2. The tool mentioned also sent malformed payloads to the API, which helped find some vulnerabilities, similar to the ones found during fuzzing, which ended up being fixed using the same testing and fixing loop mentioned.

6.4 Malicious Content Injection

Similar to the previous attack, in the sense that it is also done through the sent payload, comes Malicious Content Injection. In this attack, a script such as a SQL, JavaScript or shell is sent in the payload, with the hope that there is a system flaw, which directly executes this code without any sort of security verification, or input sanitization. If successful, the script may be able to modify, delete, or retrieve sensible data [23]. As it was previously mentioned, a security study was performed by an exterior company, which focused their study in malicious content injection. At the time of the study, some vulnerabilities were found in the application, which included SQL Injections. In this study, some guidelines and suggestions were given in order to mitigate this problem. These were taken into account, and applied throughout the system, and stored as guidelines for future software development, including the application developed in this work. After the security measures were applied, a new study was performed which showed that these vulnerabilities had been resolved.

6.5 DDoS

Denial of service attacks started to become widely used in the beginning of the 21st century, with the attacks on Yahoo!, Amazon, etc. This attack has the objective of denying the service the resource is trying to provide its clients. This is usually done by overloading the service with a much larger number of requests than it is prepared to handle, making it impossible for it to provide service to real clients. DoS attack is considered the biggest threat to IT industry, and intensity, size and frequency of the attack are observed to be increasing every year [22].

Distributed Denial of Service is still one of the most common attacks today. It originates from the Denial of Service Attack, which is easily defended against by limiting the number of requests sent by a single origin [29]. For the software at hand, there was never either a DoS or DDoS defence mechanism devised. The Open Systems Interconnection (OSI) model is an officially accepted standard by the ISO (International Organization for Standardization). This model provides a basis for the conceptualization and development of systems interconnection. This model is divided into seven abstraction layers, commonly described in a vertical list manner, which, from the bottom, are: Physical, Data Link, Network, Transport, Session, Presentation, and Application [7]. Although (D)DoS attacks are so important and preoccupying, in today's day and age, the large majority of (D)DoS attacks are performed in the lower layers of the OSI model. Regardless, attacks to the higher layers of this model, in which the work of this API is inserted, are still present, and need to be accounted for.

Attacks on layer 6 and layer 7 (Presentation and Application layers, respectively), are often categorized as application layer attacks. These are usually less common, but more sophisticated. The first defense mechanism is 'Reduce Attack Surface Area', and consists of ensuring the application is not exposed to ports, protocols, or applications from where communication is not expected [4]. In the context of this work, traffic can come from essentially any IP address, as clients may be located anywhere in the world. Every route must be accessible to all clients, so this first measure does not apply to the API.

The second defense mechanism is 'Plan for Scale'. This strategy consists of adapting the resources made available to the clients in a proportional manner to the number of clients. This allows for only the necessary resources to be used, but should have a good margin, to ensure that unusual high traffic times are not problematic. For this specific application, as it consists of an accounting software, there are times of the year in which companies use more resources. Every month, companies finish their accounting closer to the end of the month, and every year they finish their yearly accounting closer to the end of the year as well. Thus, these inconsistencies must be accounted for [4].

To start the protection following the two aforementioned strategies, a study was performed to determine typical usage. Using the results of this study, a rate limit was defined, in relation to a time frame, and is specific to some entities. As a developer of this software, I already know that a specific entity sends an unusually large number of requests on a specific day of the month, without malicious intent. This exception must be handled. And because many others like it may also exist, the solution to this problem defines different limits for different clients of the software. The difficult component of this defense is finding a limit which avoids attacks, but does not affect the real end user.

In order to perform this study, the raw data from the application's logs will be analyzed. These logs contain all the information of the requests performed to the application's backed, and will have to be filtered. Each line of the logs follows the structure shown in Listing 6.3.

```
1 Date (UTC), PID, client IP, module name, ID1, ID2, status, data
```

Listing 6.3: Log information structure

In this structure, the status indicates the state of the request being processed. It is known that, for an API request, the first status is `ST:MODULE`, and the last is `ST:CLEANUP`, so this is used to limit the start and end of the request, in the entire log. Thus, the entire log for a single request can be seen in Listing 6.4.

```
1 00:58,123,1.2.3.4,cdn,123456,0x570c,ST :MODULE ,name=cdn,
  version=0.3.22b9
2 00:58,123,1.2.3.4,cdn,123456,0x570c,IN :USER-AGENT ,Mozilla
  /5.0
```

```

3 00:58,123,1.2.3.4,cdn,123456,0x570c,IN :HOST ,app10.
   toonline.pt
4 00:58,123,1.2.3.4,cdn,123456,0x570c,IN :URI ,
   wbWTAhs0xbli
5 00:58,123,1.2.3.4,cdn,123456,0x570c,IN :METHOD ,GET
6 00:58,123,1.2.3.4,cdn,123456,0x570c,IN :CONNECTION ,close
7 00:58,123,1.2.3.4,cdn,123456,0x570c,IN :ACCEPT ,text/html,
8 00:58,123,1.2.3.4,cdn,123456,0x570c,IN :HEADER ,accept-
   encoding=gzip
9 00:58,123,1.2.3.4,cdn,123456,0x570c,IN :HEADER ,accept-
   language=pt-PT
10 00:58,123,1.2.3.4,cdn,123456,0x570c,IN :HEADER ,job-params
   =fefwfwq
11 00:58,123,1.2.3.4,cdn,123456,0x570c,ST :CONSTRUCTED ,count=1
12 00:58,123,1.2.3.4,cdn,123456,0x570c,JB :ID ,
   sequential_id ~> 53285
13 00:58,123,1.2.3.4,cdn,123456,0x570c,JB :TUBE ,render-
   public-document
14 00:58,123,1.2.3.4,cdn,123456,0x570c,JB :TTR ,180 second
   (s)
15 00:58,123,1.2.3.4,cdn,123456,0x570c,JB :EXPIRES IN ,480 second
   (s)
16 00:58,123,1.2.3.4,cdn,123456,0x570c,JB :KEY ,public-
   document:53285824
17 00:58,123,1.2.3.4,cdn,123456,0x570c,JB :CHANNEL ,public-
   document:53285824
18 00:58,123,1.2.3.4,cdn,123456,0x570c,JB :PAYLOAD ,{}
19 00:58,123,1.2.3.4,cdn,123456,0x570c,JB :TIMEOUT IN ,480 second
   (s)
20 00:59,123,1.2.3.4,cdn,123456,0x570c,OUT:CONTENT-TYPE ,text/html
21 00:59,123,1.2.3.4,cdn,123456,0x570c,OUT:CONTENT-LENGTH,44091
22 00:59,123,1.2.3.4,cdn,123456,0x570c,OUT:BODY-SIZE ,44091
23 00:59,123,1.2.3.4,cdn,123456,0x570c,OUT:BODY ,<filtered
   - binary data>
24 00:59,123,1.2.3.4,cdn,123456,0x570c,OUT:STATUS-CODE ,200
25 00:59,123,1.2.3.4,cdn,123456,0x570c,ST :CLEAN UP ,

```

Listing 6.4: Request example in log

While a request is being processed, another can enter the same machine, so the logs will be interleaved. This was taken into account, using the unique identifiers (IP, ID1, ID2). As this is a

data analysis problem, the chosen language for the development of the analysis script was Python, as it is widely recognized as the best language for problems of this nature, thanks to its ecosystem of packages, many of which developed specifically for these problems [43]. I then developed the aforementioned Python script, which performs several tasks: initially, it removes the first few lines of the log file, which contain some metadata, usually in the following format.

```
1 --- --- ---
2 LOG FILE      : /var/log/nginx-broker/cc-modules.log
3 OWNERSHIP     : 664
4 - USER      : 1000 - ooc
5 - GROUP     : 1000 - ooc
6 PERMISSIONS:
7 - MODE      : 664
8 RECYCLED AT: 2022-11-19T00:01:04+00:00
9 --- --- ---
```

Listing 6.5: API request log metadata

After this is removed, the script uses the knowledge described previously regarding the request format and creates a dictionary in which the key is the IP address of the entity that performed the request, and the value is a list of all the requests this entity has performed. Using the UTC ISO format date time in the request data, a new dictionary is created, in which the key is the time frame, and the value is the maximum number of requests performed in that time frame. Lastly, this dictionary is iterated, and the maximum number of requests performed in all the time frames is attained.

When protecting an application against DOS attacks, rate limiting is the most common defense. This limiting is usually applicable for two time windows, one for burst attacks, and one for rate limiting [38]. I devised the Python script with this in mind, and thus, it is able to get the maximum number of requests performed for any time window. For this purpose, the script was ran for a 1-hour time frame, and for a 1-minute time frame. For the 1-hour time frame, the maximum number of requests performed was 1128, while the maximum for a 1-minute time frame was 204. Using this information, I imposed two rate limits in the server responsible for the management of these requests, which is using nginx. This rate limit is based on the typical usage we studied earlier, and allows for 1500 requests in an hour, and 250 in a minute. Using the HTTP error code convention, the application will respond with an error code 429 when either of these limits is reached.

6.6 Man in the middle

The man-in-the-middle attack takes advantage of the fact that the HTTPS server provides the Web browser a certificate containing its public key. The entire communication channel is exposed if this

certificate is not reliable. A changed certificate is used in place of the authenticating HTTPS server's original certificate in such an attack. If the user disregards the browser's warning to double-check the certificate, the attack is effective. This happens too frequently, especially among individuals who routinely visit intranet sites using self-signed certificates [12]. Despite this, it is generally recommended [12, 39] that HTTPS being used, and with the correct headers, listed next, it is considered to be safe, and enough to avoid man-in-the-middle attacks [30].

- Cache-Control.
- Content-Security-Policy.
- Disable X-Powered-By.
- Strict-Transport-Security
- X-Content-Type-Protection.
- X-frame-Options
- X-XSS-Protection: helps prevent XSS (Cross-Site Scripting) attacks.

Using this information, a set of headers was determined to be essential for the security of the application, and made mandatory for the usage of the API.

6.7 Social engineering

Social engineering is one of the most common types of attacks, and yet it is often overlooked, and not properly defended against [32]. This type of attacks are usually characterized by a lower level of technical development, while focusing on the social aspect of the situation. For example, instead of trying to find security vulnerabilities and exploit them to access confidential data, the attacker deceives someone with access to the data in a way that grants him access.

As described in Section 2.1, in order for a user to connect to the API, he/she needs to consult some information, available in the TOCOnline interface. This information includes, but is not limited to, two critical pieces of information: `client_id`, and the `client_secret`. It is considered a bad practice to have this information constantly available in the interface [39]. One obvious reason is that anyone with access to an open device with the interface open, would immediately be granted access to the API.

To avoid this problem, the enrollment process in the API will be completely redone, and this sensitive information will only be provided once to the user. When a user wants to get access to the API, they will have to access the same menu as before, but a new page will be shown. In this new page, the user will provide the e-mail that should receive the API credentials, which should be the e-mail of the developer that will use the API. The developer will then receive an e-mail with a temporary link to access the information. This link leads to what was previously the API Data page, in the TOCOnline interface, with a frontend upgrade. The developer can consult all

the information needed, as well as access the old features: regenerate `client_secret`, and alter the `redirect_uri`. A new functionality allows the user to download a Postman file example, which allows him to instantly make requests to the API. After the developer has accessed the page, it will be automatically made unavailable. This happens 72h after the first access. Both the user who asked for the credentials, and the developer who received them, are warned multiple times that this information is only available temporarily, and should be stored by them safely.

During the usage of the API, should there be a necessity to alter data relative to the API access, it can be done via the new API data page. This page allows the user to void the current credentials, resend the credentials to the original developer, and setup new credentials for a new developer.

This follows what is considered to be good API security practices, and will hopefully prevent users to fall victim to a social engineering attack.

Chapter 7

Evaluating the Security of the Application

At this stage, the application's development is finalized. The security has been thoroughly tested, and the application has been improved after this security study. In order to evaluate the application objectively, the application will be tested for its security.

7.1 Setup and Design of the Evaluation

All of the security measures are in place, and the software is protected by what the literature suggests. In order to have a more objective evaluation of the security of the tool, a quantitative analysis will be performed. This analysis will be based, and using a tool previously developed in another work for this exact purpose [28]. The Security Assurance Evaluation (SAE) process consists of three main types of metrics: vulnerability metrics, security requirement metrics and assurance metrics; while the process itself consists of five main activities: application modeling, metric selection and test case definition, test case execution and measurement collection, assurance metrics and level calculation, evaluation and monitoring. The security evaluation process will be followed as described in the literature for the application at hand, and each step will be detailed and explained along the way.

7.1.1 Application Modeling

This first step consists of modeling the application and identifying what are the security requirements and which vulnerabilities may exist. As suggested by the literature, the security requirements for this work will be compiled from the OWASP ASVS [36]. Out of all of the requirements listed, the following were selected as vital for a REST API [28].

- Authentication
- JWT security

- Access Control
- Input Sanitization
- Error Handling
- Data Protection
- Communication Security
- HTTP Security
- Web Services

The next step is to select the main vulnerability types crucial for a REST API. According to the same work, the following list was gathered [28].

- Injection
- Broken Authentication
- Sensitive Data Exposure
- Broken Access Control
- Elevation of Privilege
- Cross-Site Scripting
- Cross-Site Request Forgery
- Parameter Tampering
- Man-in-the-middle-attack

7.1.2 Metric Selection, Test Case Definition, Test Case Execution, and Measurement Collection

The next two steps were combined, as they had already been previously performed. As detailed in Section 5, the necessary tests were already selected, defined, executed, and measured. Aside from the tests developed in this work, some independent security tests were also performed on the API. This last set of independent security tests was performed in two stages: the first, which identified a set of different vulnerabilities present in the API, and the second, after a few weeks of modifications and improvements, which found that these vulnerabilities had been fixed. During this process, some vulnerabilities were found, but all of these were fixed using the advice given by the testers, as well as the theoretically correct fixes.

7.2 Results

7.2.1 Assurance Metrics

At this stage, it is required to evaluate the application's performance when it comes to the security requirements and crucial vulnerabilities described previously, in the light of the tests performed. This evaluation will be performed using the GQM approach, also suggested in the work aforementioned [28], and exemplified in Table 7.1. This approach, as the name implies, involves a goal, a question, and a metric. For each goal described in Subsection 7.1.1, a set of questions will be listed, for which a metric (answer) will be given. In order to protect the company, the exact questions are not shown in this work, as they can facilitate a future attack. Using these answers, a value will be attributed to describe if the goal was achieved. Using a weighted average of all the goals, in which the weight is attributed by me, considering the importance of the goal, a final score will be given, which will evaluate the security of the application.

Sub-goal	Question/Test case	Answer/Metric
Use input sanitization	Do server side input validation failures result in request rejection and are logged?	1(Full)
	Are input validation routines enforced on the server side?	1(Full)
	Are prepared statements used for protecting SQL queries, and others?	1(Full)
	Are security controls preventing LDAP Injection enabled?	0(Weak)
	Is client side validation used?	0(Weak)
	Is positive validation (whitelisting) used for input data, like REST calls and HTTP headers?	0(Weak)
	Is JSON.parse used to parse JSON on the client.?	0(Weak)

Table 7.1: Example of applying GQM approach to quantify the fulfillment factor for the security requirement (sub-goal): Use input sanitization.[28]

After this process, each requirement and vulnerability was evaluated, resulting in Table 7.2. The details of the questions used to evaluate each vulnerability and security requirement were omitted as to protect the integrity of the company's system. It should only be noted that an evaluation of weak indicates that no question had a positive answer, average indicates that some questions had a positive answer, and full indicates that all questions had a positive answer.

Injection, Cross-Site Scripting, Cross-Site Request Forgery, were tested using the fuzzing technique described in Subsection 6.2, and in the independent security tests. Although none of these vulnerabilities were found in the application, not all of the principles required to protect against these types of attacks were followed, such as using a Web Application Firewall.

Broken Authentication, or Identification and Authentication Failures, as it is currently known, was also given an average evaluation. Although this type of vulnerability was not accounted for in this work, there were already some protections in place for this. There are no deployed credentials,

Category	Name	Evaluation
Vulnerabilities	Injection	0.5 (Average)
	Cross-Site Scripting	0.5 (Average)
	Cross-Site Request Forgery	0.5 (Average)
	Broken Authentication	0.5(Average)
	Sensitive Data Exposure	0.5(Average)
	Broken Access Control	0.5(Average)
	Elevation of Privilege	0(Full)
	Parameter Tampering	0(Full)
	Man-in-the-middle-attack	0(Full)
Security Requirements	Authentication	1(Full)
	JWT security	0.5(Average)
	Access Control	0.5(Average)
	Input Sanitization	1(Full)
	Error Handling	0.5(Average)
	Data Protection	1(Full)
	Communication Security	1(Full)
	HTTP Security	1(Full)
	Web Services	0.5(Average)

Table 7.2: Evaluation of the Vulnerabilites and Security Requirements for the application

there is a rate limit for the number of authentication requests, and the API uses Oauth 2.0 for its authentication, which already comes with a large number of security protections.

Sensitive Data Exposure, or Cryptographic Failures, as it is currently known, was evaluated with average. This comes due to the usage of several different security measures, such as encrypting all sensitive information with secure and modern hashes, using HTTPS for communications, ensuring cryptographic randomness, among others.

Broken Access Control was evaluated with an average result. This vulnerability is recommended to be protected against by denying resources by default, re-using common access control mechanisms, provide rate limits, remove metadata and backup files from production, and log access control failures. Although not all of these are present, there is a majority implemented.

Elevation of Privilege was evaluated with the lowest score, which in the case of vulnerabilities, for this method, represents the safest score. This vulnerability is recommended to be protected against by using proper authentication and authorization, using the least privilege principle, validate inputs, use HTTPS, use rate limiting, using content security policies, and using security headers. All of these mechanisms have been put to use in the current application.

Parameter Tampering was also evaluated with the best score. This was the most thoroughly tested vulnerability of the application, and as it was mentioned in Subsection 6.2 and Subsection 6.1, a large number of fuzzing and unit tests were performed, with the aim of breaching the application by parameter tampering. All of the found problems were fixed, and the application passed all final tests.

Lastly, for Man-In-The-Middle attacks, the application was also awarded the best score. The

recommended protections are using HTTPS, API keys, and authentication tokens, which are all present in the measures taken in the development of this application.

All of the information regarding the types of vulnerabilities, how to protect against them, and the common mistakes made were attained from OWASP's official website [36]. The same source was used for the security requirements, described next.

Regarding authentication, a full score was attributed. User ID's are being used, sensitive accounts are not allowed to log in, there is a secure password recovery mechanism, passwords are stored securely, safe functions are used to compare password hashes, there is a change password mechanism, and error messages are generic.

For JWT security, an average score was given. There has been a convergence to use these as security tokens, as they are currently considered the most secure to use. For the use-case at hand, they are used for the credentials page, as it does not require a session to be accessed. Although it is used, and follows most security principles, it lacks the issuer and audience claims, which are considered essential. This method is also used in a number of other cases in the application, such as password redefinition page, and joining invitation links.

Access Control, now referred to as Authorization, was given an average score. There is a large number of requirements in this section, most of which are appropriate to this part of the application, and are enforced. Once again, the Least Privilege principle is applied, as well as Deny by Default. Permissions are validated on every request, and safely exit on authorization fails.

Input Sanitization was given a full score, as all of the user inputs are previously validated, and only executed as a database query as a prepared statement.

Error Handling was given an average score. Although a large number of cases was considered, and all of the considered errors are properly handled, there are still some cases that come up sometimes in which a generic error message is provided, which may not be sufficiently clear to some end-users.

Data Protection was given a full score. Only the strictly necessary data is provided to a user, for any CRUD operation. The most sensitive data is not stored in plain text in the database, and is hashed with a secure algorithm. The database itself is structured in a sharded manner, which guarantees that a given user only has access to the information of the company he is enlisted in.

Communication Security and HTTP Security were also given a full score. The API communicates using HTTPS, with the necessary security headers.

Web Services was given an average score, as not all of the requirements are met. Although server and user authentication, transport encoding, message integrity and confidentiality, content-validation, and availability are present, some mechanisms such as JSON DoS protection are not put in place.

7.2.2 Level Calculation

At this point, all of the security requirements and vulnerabilities were evaluated. Using the same formula described in the literature, it is now possible to evaluate the tool developed. It was considered that any vulnerability or security requirement could lead to a fatal flaw in the system,

and as such, the weights of the different components were considered to be identical. The current step will yield an objective value of security to the application. This result will be on a 0-10 scale, and each value represents a degree of security as described in Table 7.3

Score	Degree of Security
0.0 - 0.99	None Security
1 - 3.9	Low security
4.0 - 6.9	Moderate Security
7.0 - 8.9	Very Good Security
9.0 - 10.0	Excellent Security

Table 7.3: Degree of Security [28]

According to the literature, the degree of security of the application, or the security Assurance Metric (AM) should be calculated with Equation 7.1.

$$AM = RM - VM \quad (7.1)$$

In this equation, RM represents the security Requirement Metric, and VM represents the Vulnerability Metric. These can be calculated using Equation 7.2 and Equation 7.3, respectively.

$$RM = \sum_{i=1}^m Rm_i \quad (7.2)$$

$$VM = \sum_{k=1}^n Vm_k \quad (7.3)$$

For each of these, Rm_i and Vm_k represent the requirement metric, for a given security requirement (i), or vulnerability metric, for a given vulnerability (k), respectively. With the different vulnerabilities and security requirements being the ones specified in Table 7.2. In order to calculate each Rm_i and Vm_k , the following Equation 7.4 and Equation 7.5 should be used, respectively.

$$Rm_i = (w_i * \sum_{j=1}^k f_{ij}) \quad (7.4)$$

In this equation, Rm_i is evaluated by multiplying w_i , which is the weight of a specific requirement, multiplied by the sum of the evaluation of the fulfillment of each test case j, for the requirement i: f_{ij} .

$$Vm_k = (w_k * \sum_{l=1}^n f_{kl}) \quad (7.5)$$

In this equation, Vm_k is evaluated by multiplying w_k , which is the weight of a specific requirement, multiplied by the sum of the evaluation of the fulfillment of each test case l, for the requirement k: f_{kl} .

Lastly, given the number of vulnerabilities, security requirements chosen, and the weight of each, there is a necessity to normalize the results to the security degree table, displayed in Table 7.3, which goes from 0-10.

In order to do so, Equation 7.6 was used. In this equation, the value in the new scale (v') is calculated using the value in the original scale (v), the minimum and maximum values of the old scale (min_A and max_A respectively), and using the minimum and maximum values of the new scale ($newmin_A$ and $newmax_A$ respectively). The method used here is the min-max normalization [11].

$$v' = (v - min_A) / (max_A - min_A) * (newmax_A - newmin_A) + newmin_A \quad (7.6)$$

Using these metrics, the evaluation resulted in the application obtaining a degree of 'Very Good Security', with an objective evaluation of 7.27.

7.3 Result Analysis

After this security study, I can say the application developed is fairly well protected. More objectively, about a 7/10 in the security scale used, which places the application in a moderate/very good evaluation. The different types of potential attacks were studied, their prevention was analyzed, and implemented. Any application can always be more secure, there is no such thing as a 100% secure application. A good balance between the theoretical practices and the company's requirements was found: there was a robust security study performed, while maintaining the company's schedule for the application release.

Chapter 8

Evaluating the Performance of the Application

At this stage, the first and most important evaluation step is concluded. Security has been thoroughly tested and evaluated. In order to continue evaluating the application it is necessary to test and evaluate its performance and usability. This work is described in the present chapter.

8.1 Performance Evaluation

8.1.1 Setup and Design of the Evaluation

Performance testing is done over several different categories, which will be described in the listing below.

- **Baseline testing:** The goal of this testing is to determine how well the system operates under a load that is typically anticipated. Analysis of the average, peak, and error rates for the API should be done using the test's data. To get rid of any resource bottlenecks, the platform's CPU and memory use should also be examined [16].
- **Load testing:** Testing under increased load, the performance of the API is examined. To evaluate the performance under load, performance parameters like response time and throughput of the APIs should be examined. This testing's objective is to comprehend the expected system behavior and capacity to handle anticipated peak loads, not to locate the system's breaking point. To assess the condition of the platform and its capacity to manage heavy demand, server performance measures such as CPU usage, heap memory use, and network port utilization should be examined [16].
- **Stress testing:** The purpose of stress testing is to determine the platform's breaking point. It is used to calculate the system's maximum throughput capacity. In this type of testing, the demand on the API is steadily increased until a threshold is reached when performance starts to suffer or API call failures start to rise [16].

- **Soak testing:** In long-term testing, soak testing is used to identify system instabilities. To discover any undesirable behaviors that could emerge when the system is utilized for a long period, the baseline test may be carried out over the course of several days or weeks. The objective is to identify any problems with freeing up system resources so they may be used for the subsequent cycle of execution. System crashes under prolonged high loads are quite likely if system resources are not being released on a regular basis. Soak testing becomes more crucial when baseline testing or load testing are unable to detect such issues [16].

Although this is the theoretically correct way of performing this type of tests, the performance of the API was not the most important factor for the company, it was only necessary that the performance was similar to the previous application, as it was proven to be sufficient for the number of clients at hand. Thus, I performed several performance tests in which all of the possible requests were performed to the API, with their response times measured. In order to obtain these results in a standardized and repeatable manner, Postman was used. This decision is explained in Section 4.1. Using Postman, namely its 'Runner' feature, it is possible to automatically run requests in a collection for a given number of iterations, specifying the delay between them.

8.1.2 Results

Several different performance tests were performed. In the first test, different use cases were taken into account, such as performing a GET to a document that existed, and one that did not, performing POST requests with errors in the body, or with correct payloads to test the actual insertion of the document, among others. In summary, this is a baseline testing, which details how the system performs under typical loads, and covers the large majority of the possible requests the API is prepared to respond to. After these results were obtained, the average was calculated and grouped into request types. The results are displayed in Figure 8.1.

As it was previously mentioned, the existing version of the API already has a satisfactory performance and thus it is being used as the benchmark for the new version. As we can see by the results in Figure 8.1, the new version behaves similarly to the previous version, with response times in the same order of magnitude. Although they are similar, there is a small difference between the average times of the two versions. The older version is about 11% faster than the new one when it comes to single-action requests. This was to be expected, as it was a company decision, and the results are still in the acceptable range for the company. This difference in performance is due to the fact that I took the opportunity of creating this new application to start the implementation of a new technological stack for the company's software. This had already been decided by the company before the beginning of this project, and although it comes with a small performance downgrade, the stack is now easier to use, more programmer-friendly, more updated, and future-proof. This was exactly the goal defined initially, and thus I can conclude that the new version of the API has a good performance to be used by the company's clients when it comes to single requests.

The next type of test uses a more specific set of requests. The new API allows for a great number of operations to be performed in a single request, but is also prepared to perform the single

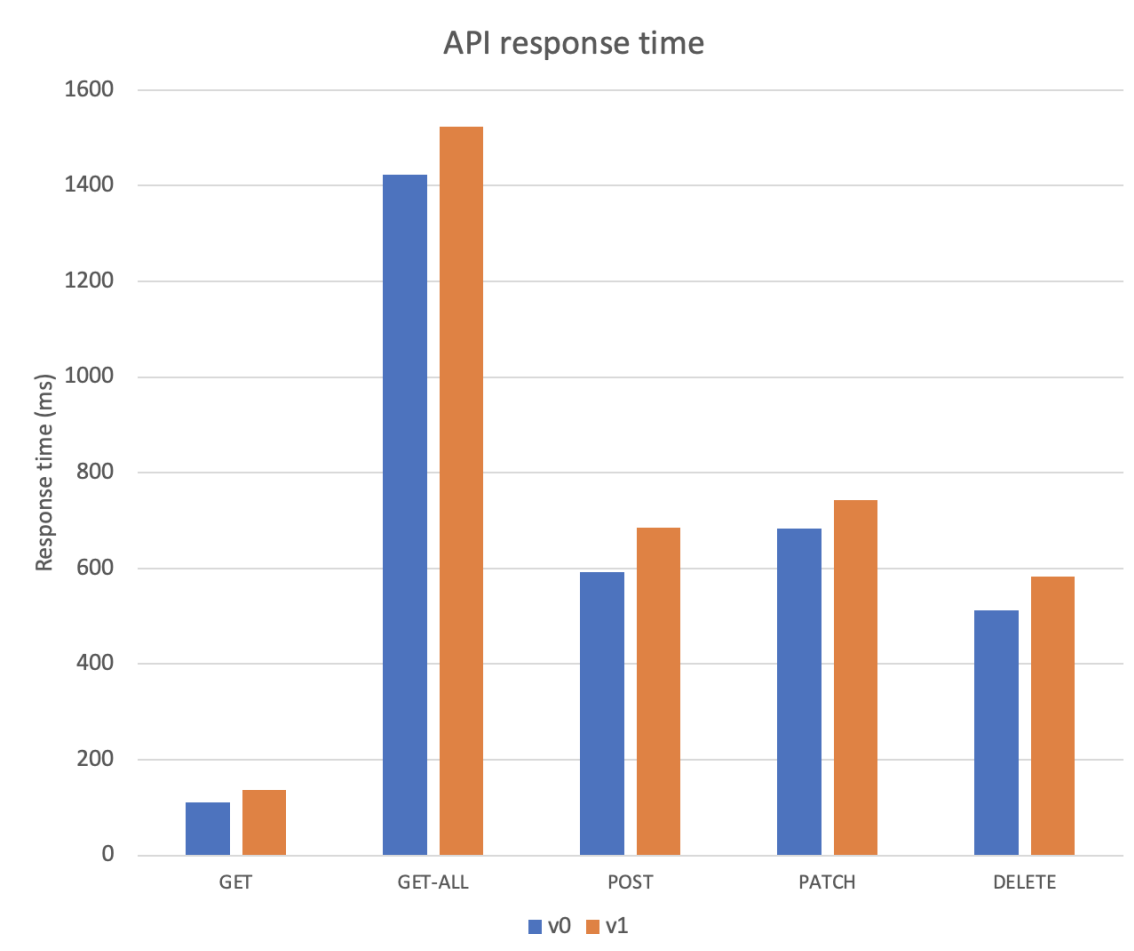


Figure 8.1: Performance results of the new API, in comparison to the old version

actions present in the older version, as they may be necessary. Thus, it is expected that some requests take longer than the previous version. For example, a POST request in the previous API, which merely created the header for a bill, is faster than a request in the new API, which creates a header, several lines, generates a bill PDF, and finalizes the document. This data can be seen in Figure 8.2.

In this chart, the first four categories represent the time taken by each version of the API to perform each of the actions: create header, create line, generate PDF, and finalize the document. The fifth column represents the sum of the times of each operation, for both versions. For both of the versions, an average of 5 line requests was used, as this is the average number of lines present in a bill in the system. Obviously, the more lines a user needs to introduce in their bill, the longer the whole process will take, as a request for each line must be performed, unless the new version is being used. The last column represents the time it takes for the new API to perform the single request that executes all of these actions.

From the data presented in Figure 8.2, it is clear that both versions of the API offer similar results in each individual operation, visible in the first four columns. As it was previously mentioned, there is a slight loss of performance in the new version, which is expected, and is still in the acceptable

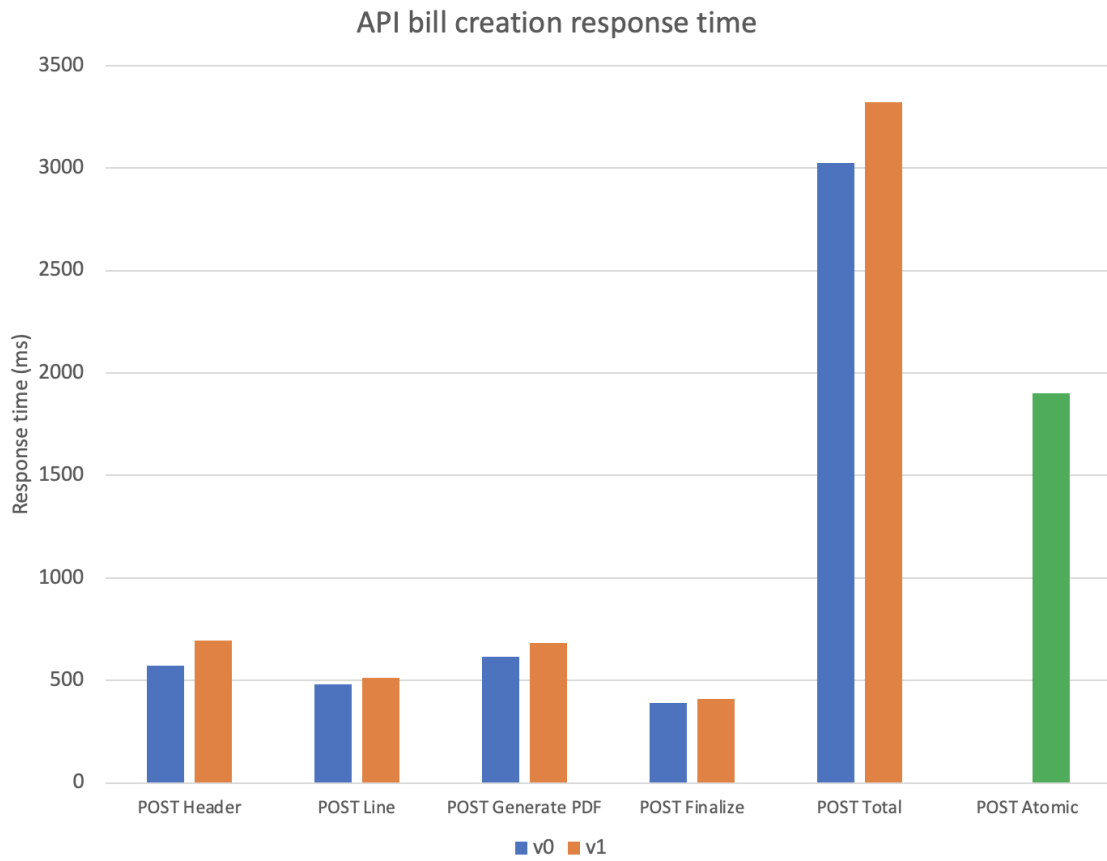


Figure 8.2: Performance results of the new API, in comparison to the old version

parameters for the company. From the fifth column, the total of these operations are also similar, which is to be expected, given that this column is merely the sum of all previous 4 columns. The big improvement given by the new API is visible on the last column. This is the average time recorded for the new API to perform a single request to execute all of the previously mentioned actions. This value is significantly lower than its counterpart, column four. It should be also noted that the times recorded only refer to the time of the requests themselves. In a real world application, in order to perform all of these actions using the previous version, there would also be an overhead caused by the program actually sending and receiving the multiple different requests, which will cause an even larger gap between the two versions of the application. Thus, even though these are already great results, it is worth mentioning that the real practical improvement is even larger.

8.1.3 Result Analysis

Following the results attained in this performance study, I concluded that the new application developed has a good performance. The response times are similar to the previous version, which was the benchmark. These response times are satisfactory for our clients, and are in line with the response times of other commercially used API's. This is all in relation to the single-action requests. The major improvement in the application is the ability to perform a single request which

fulfills a large number of actions, in which case there was a large improvement relatively to the older version.

Chapter 9

Deployment and release of the API

At this point, all of the conceptualized features for the new API version are implemented, and the entire software has been tested, fixed, and properly evaluated with an objective metric in regards to its security. Thus, the next step is to deploy the software, make it available to the end-user, provide it to them, and collect feedback on how to improve it. As this is a major update on the API, the existing must still be maintained and kept alive, a large number of clients still depend on it.

9.1 Documentation

Similarly to the previously developed documentation, it was necessary to document the newly developed API version. This documentation was developed in a similar manner to the previous documentation. It was also written in Markdown, and deployed in gitbook. Despite this, it was written manually, instead of automatically generated, as there are no YAML files describing these new routes.

Some components are common to both versions and do not need to be described again. These components are the authentication process (both the manual and the Oauth 2.0), and all of the knowledge regarding the inputs of the API routes, such as currency identifiers, document referencing, etc. Thus, it is merely necessary to introduce the new alterations done, and detail the new routes. The first page of the new documentation provides a detailed explanation of the differences between this new version, and the older version of the API, further referred to as Legacy API. These are the differences explained in Chapter 5. The entirety of the documentation is available in the same link, and refers to the documentation for both versions: <https://cloudware.gitbook.io/documentacao-api>.

The remaining pages are grouped by document type: bills, receipts, purchases, and payments. Each of these detail all of the routes developed for each type of document. Similarly to the previously developed documentation, each page contains, for each request, a cURL command, a visual description of the request, using gitbook's API request block, and a textual explanation of the purpose and usage of the request. Once again, and for the same reasons as described earlier, this documentation is also written in Portuguese. As an example, a section of the bills page of the new documentation can be seen in Figure 9.1

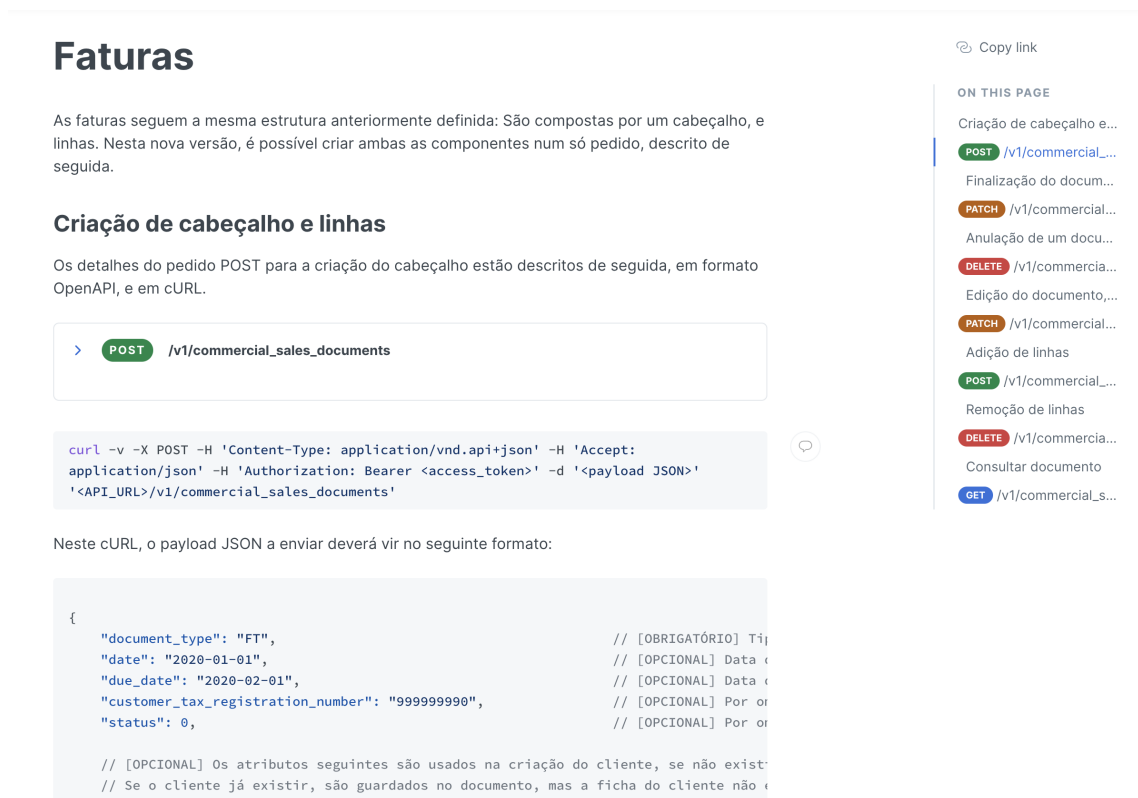


Figure 9.1: Bill page of the new documentation

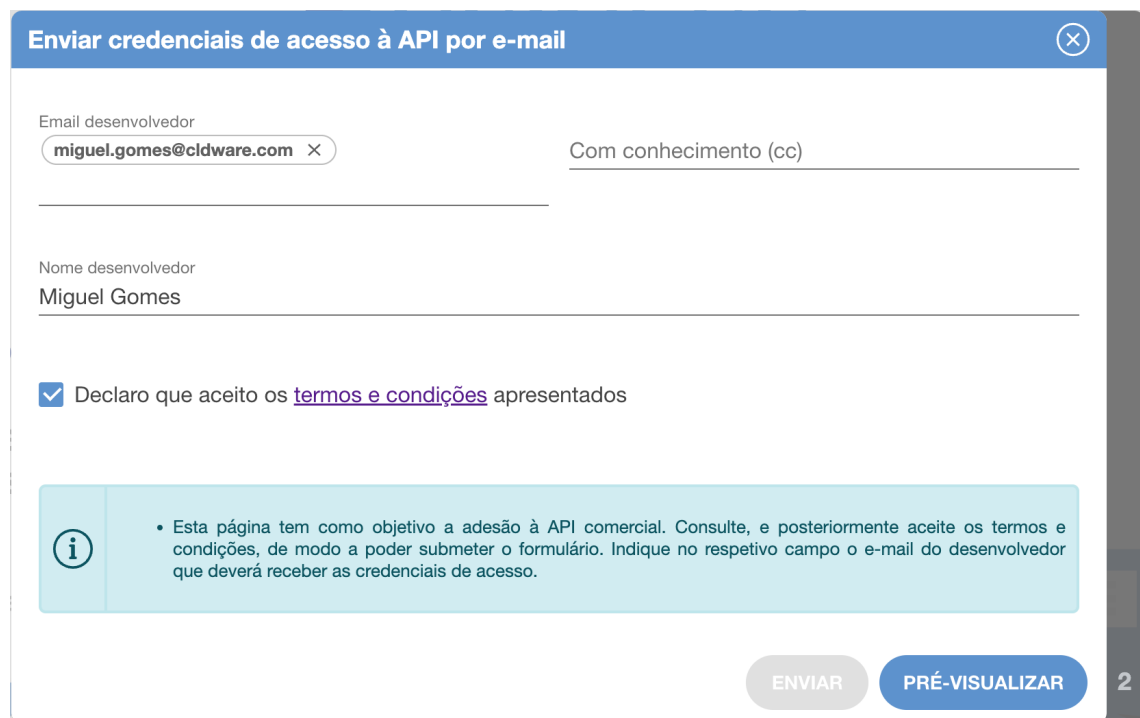
9.2 Enrollment Process

As described in Chapter 5, a new enrollment process was developed. This allows for greater security in terms of the API access, and provides an additional layer of protection against social engineering. This also allows the company to save the e-mail of the developers who will use the new API, which will be essential to obtain feedback of this new tool. The screen presented to the user, as the first step to enroll in the API process is shown in Figure 9.2. Like the documentation, the website is also written in Portuguese. This enrollment process is now used by any client that wishes to start using the API, or a client that already uses the API, but will now require new credentials, either to perform a new integration, or to delete the credentials given to a previous worker.

This first page, shown in Figure 9.2 allows the user to define the e-mail, and name of the developer. These will then be used to send an e-mail to the developer, containing the credentials necessary for them to interact with the API. The second page, shown in Figure 9.3, displays a preview of the e-mail, which will be sent to the developer.

When the developer receives the e-mail, they can click the blue button, which will give them the necessary credentials to access the API. In order to ensure another layer of security, the ability to consult and make alterations to the credentials will be temporary. From the moment the developer first opens the credentials page, a timer will start, which will make the page unavailable after the timer reaches 72h.

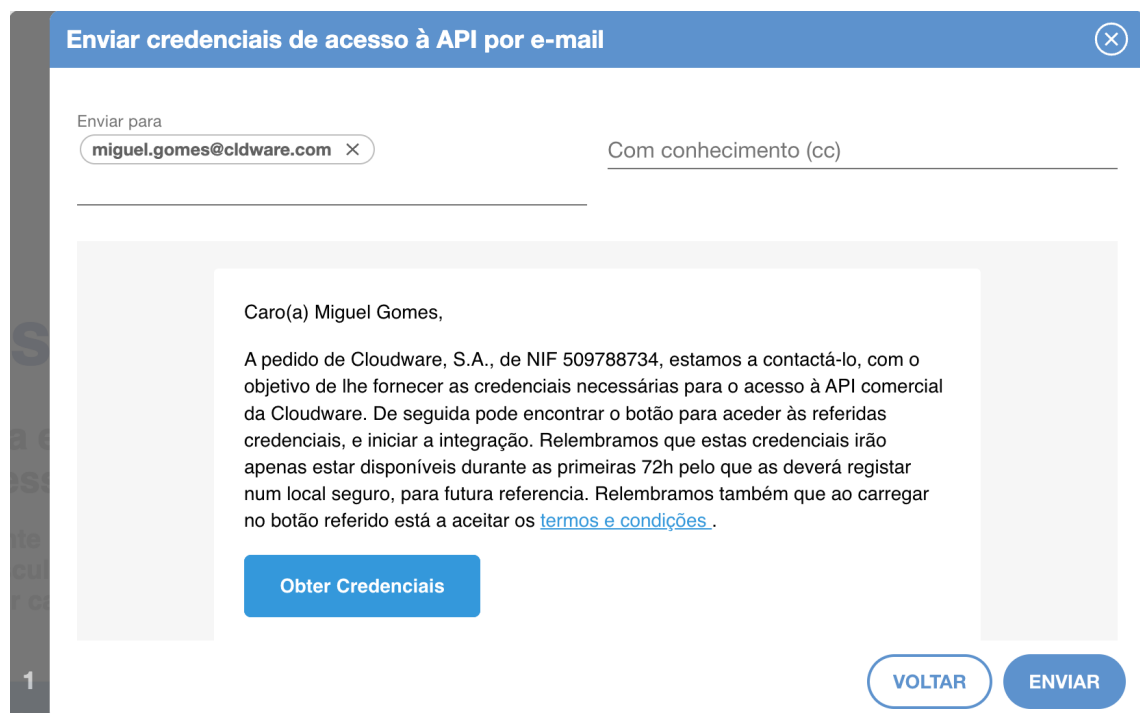
This new page contains the same functionalities present in the old credentials page, with two



The first screen of the enrollment process is titled "Enviar credenciais de acesso à API por e-mail". It features a form with the following elements:

- Email desenvolvedor:** A text input field containing "miguel.gomes@cldware.com" with a close button (X).
- Com conhecimento (cc):** A text input field for additional email addresses.
- Nome desenvolvedor:** A text input field containing "Miguel Gomes".
- Declaração:** A checkbox labeled "Declaro que aceito os [termos e condições](#) apresentados" is checked.
- Informação:** A light blue box with an information icon (i) and the text: "Esta página tem como objetivo a adesão à API comercial. Consulte, e posteriormente aceite os termos e condições, de modo a poder submeter o formulário. Indique no respetivo campo o e-mail do desenvolvedor que deverá receber as credenciais de acesso."
- Buttons:** "ENVIAR" (disabled) and "PRÉ-VISUALIZAR" (active).
- Page Indicator:** A small "2" in a grey box at the bottom right.

Figure 9.2: First screen of the enrollment process



The second screen of the enrollment process is titled "Enviar credenciais de acesso à API por e-mail". It features a form with the following elements:

- Enviar para:** A text input field containing "miguel.gomes@cldware.com" with a close button (X).
- Com conhecimento (cc):** A text input field for additional email addresses.
- Preview:** A large grey box containing a preview of the email content:
 - Greeting: "Caro(a) Miguel Gomes,"
 - Body: "A pedido de Cloudware, S.A., de NIF 509788734, estamos a contactá-lo, com o objetivo de lhe fornecer as credenciais necessárias para o acesso à API comercial da Cloudware. De seguida pode encontrar o botão para aceder às referidas credenciais, e iniciar a integração. Relembramos que estas credenciais irão apenas estar disponíveis durante as primeiras 72h pelo que as deverá registar num local seguro, para futura referência. Relembramos também que ao carregar no botão referido está a aceitar os [termos e condições](#)."
 - Button: "Obter Credenciais"
- Buttons:** "VOLTAR" and "ENVIAR".
- Page Indicator:** A small "1" in a grey box at the bottom left.

Figure 9.3: Second screen of the enrollment process

newly added features. The first new feature allows the user to download a JSON file, which can be imported into Postman, and instantly used to experiment with the API. This is similar to the Postman file described in Section 5.1, with a new fool-proof addition: the information of this file is

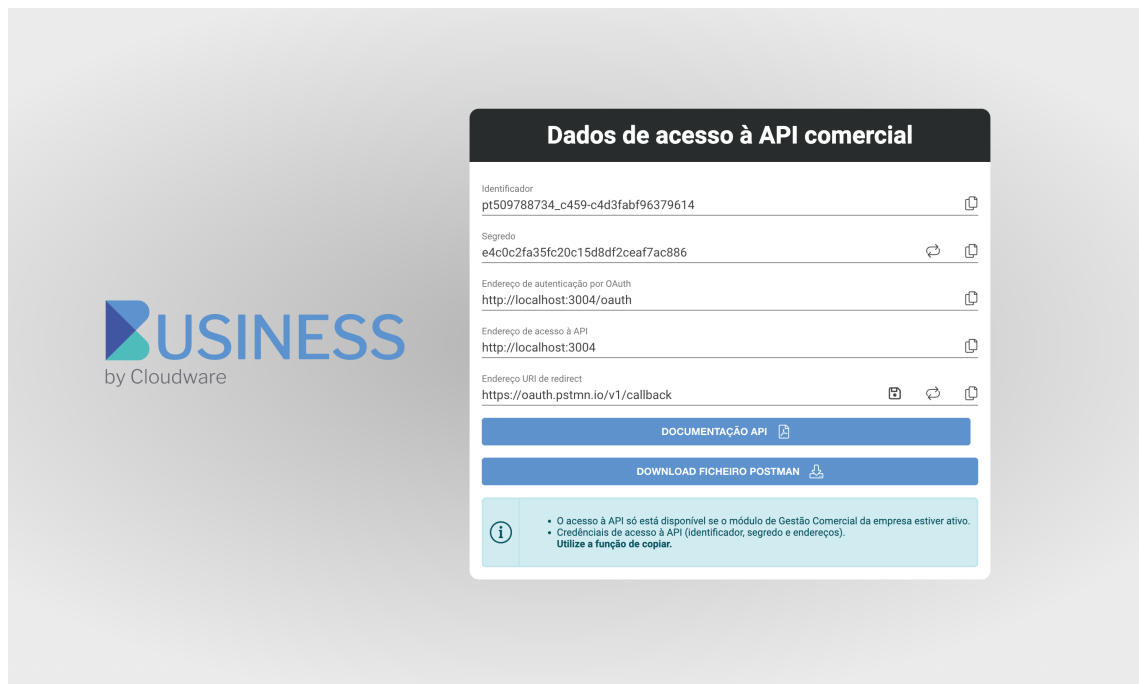


Figure 9.4: New API credentials page

specific to the user who downloads it and is thus already ready to be used. The user does not need to follow any instructions and edit variables in Postman, it is only needed to click a button to perform the authentication automatically, and start performing requests, which are already prepared in the file itself. The second new feature simply allows the user to access the documentation developed for the new API, detailed in Section 9.1. The remaining features were already present in the old API credentials page, and consist of: regenerating the client secret, altering the redirect URI, and restoring the original redirect URI. This new page is visible in Figure 9.4.

At this point, everything is ready to be presented to the client, and thus the deployment process began. As this process is very specific to the company and contains sensible information which should not be public, the details of the deployment process were omitted in this work. It is only important to know that all of the newly developed pages, processes, and features were deployed to production, and made available to the company's clients.

Chapter 10

Conclusions and Future Work

10.1 Goals met

This work was able to achieve most of the milestones initially proposed. As the objective was to help the clients use the API, there were two main groups of tasks that were developed: tasks that helped the users with the original API, and the development of the new API. In the first group of tasks, several scraping tools were developed which analyze the codebase and generate documentation for the users both in markdown and swagger; hand written documentation was also created from scratch, which guides the users throughout the whole enrollment and authentication process, followed by detailed instructions for each and every single API endpoint; lastly a program to generate a custom Postman file which contains all of the endpoints, ready to use, with automatic authentication, and request examples, which led to the customers first experimenting with the API to be able to make their first request with a single button click was developed. In the second group of tasks, an entire new API was developed, with a client-focused structure, which aimed to be more intuitive, and simple to use, which led to a decrease in the need for help interacting with the API. Much like the previous version, several pages of documentation were developed, once again guiding the user through the authentication process, enrollment process, as well as every single API request. A new enrollment process was developed, which has an entire set of full-stack features, and allowed for a more secure protection of the access credentials, as well as saving information about the clients using the API, and an acceptance of Terms and Conditions, vital for the company's protection. During the development of the API, a security study was performed, which shed light onto the correct procedures to use when developing the application, and allowed for a secure piece of software. This study also improved the application's security, and evaluated it objectively. In summary, I believe I was successful in developing a secure and easy to use API, with all of the necessary extra accessories. The API is currently deployed and being used by the clients, and has been well reviewed.

10.2 Future work

Although this work achieved a great number of milestones, there is always more work to be done. As it was mentioned in the previous section, a great number of API routes was developed. However, the original API contains an even larger number of endpoints. The adapter was a great solution, and allowed for all of the requests to be performed to the same url, but it is still necessary to recreate the remaining routes using the same procedure: study usability and security, and develop the remaining endpoints with these two characteristics in mind. These were not yet developed as they are less used routes, and had presented no problems at the point of the development of this work, but should still be considered for the near future. As it was mentioned during the security study, no piece of software is 100% secure. Thus, there is always more room for improvement when it comes to security. New vulnerabilities will exist, and new methods to test the API will be created, which leads to a need for a continuous security study for the application, as it is being commercially used.

Lastly, and most importantly, one of the main focus points of this work was to improve the usability of the company's API. Initially, there was a plan to evaluate the usability at the end of the work, by surveying the API's users, and there were also some thought of plans to survey some students or general people of this area in order to learn their thoughts on the improvements in usability. Unfortunately, at the end of the development of this work, there was not enough time to receive a significant and large enough number of responses to this survey. This was mainly due to the time of the deployment of the API. It was delayed to the end of the year, which is a difficult time for the company's software users. In the accounting world, the end of the year is a stressful and busy time due to the fact that the accounting of the entire year must be sorted, and there are tight deadlines to fulfill. The API was deployed at the end of the year, and so there was not a large number of users willing to experiment and move to a new application in such a complicated time. At the beginning of the next year after the release, a large increase in new users has been spotted, which corresponded to my expectations, and is a good indicator of the quality of the application made available to the customers. Despite this, I still believe there was a great improvement of the usability of the application, due to the feedback of other employees, and due to the fact that at the time of submission of this work, there has not been a single e-mail sent requiring support to use the new API structure. Every client currently using this new version has been able to use it properly only with the help of the documentations created and the auxiliary tools, namely the automatically generated Postman file. Thus, the need to objectively evaluate the API's usability, find possible improvements, and implement them is the most important step for the future work in this application.

References

- [1] Advanced rest client. <https://docs.advancedrestclient.com/using-arc/workspaces>. Accessed: 2022-09-22.
- [2] Chai assertion library. <https://www.chaijs.com/>. Accessed: 2022-09-21.
- [3] Curl. <https://curl.se/>. Accessed: 2022-07-10.
- [4] Ddos attack protection. <https://aws.amazon.com/shield/ddos-attack-protection/>. Accessed: 2022-07-10.
- [5] The fun, simple, flexible javascript test framework. <https://mochajs.org/>. Accessed: 2022-09-10.
- [6] The next generation java testing. <https://testng.org/doc/>. Accessed: 2022-09-10.
- [7] Mohammed M Alani. Osi model. In *Guide to OSI and TCP/IP Models*, pages 5–17. Springer, 2014.
- [8] Andrea Arcuri. Restful api automated test case generation. *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2017.
- [9] V. Atlidakis, P. Godefroid, and M. Polishchuk. Restler: Stateful rest api fuzzing. In *Proceedings of the 41st International Conference on Software Engineering, ICSE'19*, pages 748–758, Piscataway, NJ, USA, 2019. IEEE Press.
- [10] Stefan Bechtold. Junit 5 user guide. <https://junit.org/junit5/docs/current/user-guide/#writing-tests>.
- [11] Steve Borgatti. *Normalizing Variables (Handout)*. 2018.
- [12] Franco Callegati, Walter Cerroni, and Marco Ramilli. Man-in-the-middle attack to the https protocol. *IEEE Security and Privacy*, 7(1):78–81, 2009.
- [13] Antonio Carzaniga, Gian Pietro Picco, and Giovanni Vigna. Is code still moving around? looking back at a decade of code mobility. In *29th International Conference on Software Engineering (ICSE'07 Companion)*. IEEE, May 2007.
- [14] John Daughtry, Joseph Lawrance, Brad Myers, André Santos, and Chamila Wijayarathna. Api usability. <https://sites.google.com/site/apiusability>.
- [15] Kevin S. David B., Wendell S. Programmable web. <https://www.programmableweb.com>. Accessed: 2022-07-22.
- [16] Brajesh De. *API Testing Strategy*, pages 153–164. Apress, Berkeley, CA, 2017.

- [17] Adeel Ehsan, Mohammed Ahmad M. E. Abuhaliqa, Cagatay Catal, and Deepti Mishra. Restful api testing methodologies: Rationale, challenges, and solution directions. *Applied Sciences*, 12(9), 2022.
- [18] S. Fahl, M. Harbach, H. Perl, M. Koetter, and M. Smith. Rethinking ssl development in an appified world. 2013.
- [19] R. Fielding and J. Reschke. Hypertext transfer protocol (http/1.1): Semantics and content, Jun 1970. <https://www.rfc-editor.org/rfc/rfc7231>.
- [20] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000.
- [21] Gao Gao, Finn Voichick, Michelle Ichinco, and Caitlin Kelleher. Exploring programmers’ api learning processes: Collecting web resources as external memory. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–10, 2020.
- [22] Brij B Gupta and Omkar P Badve. Taxonomy of dos and ddos attacks and desirable defense mechanism in a cloud computing environment. *Neural Computing and Applications*, 28(12):3655–3682, 2017.
- [23] William G Halfond, Jeremy Viegas, Alessandro Orso, et al. A classification of sql-injection attacks and countermeasures. In *Proceedings of the IEEE international symposium on secure software engineering*, volume 1, pages 13–15. IEEE, 2006.
- [24] Dick Hardt. The oauth 2.0 authorization framework. Technical report, 2012.
- [25] Frederick Hirsch, John Kemp, and Jani Ilkka. *Mobile web services: architecture and implementation*. John Wiley and Sons, 2007.
- [26] Isha, Abhinav Sharma, and M. Revathi. Automated api testing. *2018 3rd International Conference on Inventive Computation Technologies (ICICT)*, 2018.
- [27] Stefan Karlsson, Adnan Causevic, and Daniel Sundmark. Quickrest: Property-based test generation of openapi-described restful apis. *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, 2020.
- [28] Basel Katt and Nishu Prasher. Quantitative security assurance metrics: Rest api case studies. In *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*, pages 1–7, 2018.
- [29] F. Lau, S.H. Rubin, M.H. Smith, and L. Trajkovic. Distributed denial of service attacks. In *Smc 2000 conference proceedings. 2000 ieee international conference on systems, man and cybernetics. 'cybernetics evolving to systems, humans, organizations, and their complex interactions' (cat. no.0, volume 3, pages 2275–2280 vol.3, 2000*.
- [30] Arturs Lavrenovs and F Jesús Rubio Melón. Http security headers analysis of top one million websites. In *2018 10th International Conference on Cyber Conflict (CyCon)*, pages 345–370. IEEE, 2018.
- [31] Donald Lewine. *POSIX Programmer’s Guide*. O’Reilly Media, Sebastopol, CA, April 1991.
- [32] I. Mann. *Hacking the human: Social engineering techniques and security countermeasures* (1st ed.). 2008.

- [33] Matt Morley. Rpc 2.0 specification.
- [34] Susan Murillo. What you should know about standards, apis, interfaces and bindings, 2022.
- [35] Brad A. Myers and Jeffrey Stylos. Improving api usability. *Commun. ACM*, 59(6), may 2016.
- [36] E. Oftedal, A. Stock, T. Chih, J. Peeters, J. Wolff, and R. Granitz. Rest security cheat sheet, 2017.
- [37] M. Reddy. *API Design for C++*. Elsevier Science, 2011.
- [38] Salah Sharieh and Alexander Ferworn. Securing apis and chaos engineering. In *2021 IEEE Conference on Communications and Network Security (CNS)*, pages 290–294. IEEE, 2021.
- [39] Prabath Siriwardena. *Advanced API Security*. Apress, 2020.
- [40] Anshu Soni and Virender Ranga. Api features individualizing of web services: Rest and soap. *International Journal of Innovative Technology and Exploring Engineering*, 8(9):664–671, 2019.
- [41] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing*. Addison-Wesley Educational, Boston, MA, June 2007.
- [42] Robert Van Engelen, Kyle Gallivan, Gunjan Gupta, and George Cybenko. Xml-rpc agents for distributed scientific computing. In *IMACS'2000 Conference*, 2000.
- [43] Jake VanderPlas. *Python data science handbook: Essential tools for working with data*. "O'Reilly Media, Inc.", 2016.
- [44] Ervin Varga. *The Core JSON API*, pages 229–247. Apress, Berkeley, CA, 2016.