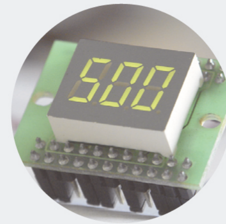
INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

MESTRADO EM ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES

isep



APIbuster Testing Framework

PEDRO FERREIRA DE SOUSA

novembro de 2022

POLITÉCNICO
DO PORTO

Politécnico do Porto

Instituto Superior de Engenharia do Porto

APIbuster Testing Framework

Pedro Ferreira de Sousa

Master in Electrical and Computer Engineering
Major in Telecommunications

Departamento de Engenharia Eletrotécnica
Instituto Superior de Engenharia do Porto

November, 2022

This dissertation partially satisfies the requirements of the Thesis/Dissertation course of the program Master in Electrical and Computer Engineering, Major in Telecommunications.

Candidate: Pedro Ferreira de Sousa, No. 1120641, `1120641@isep.ipp.pt`

Scientific Guidance: Maria Benedita Campos Neves Malheiro, `mbm@isep.ipp.pt`

Company: INESC TEC

Advisor: Gonçalo Campos Gonçalves, `gfcg@inesctec.pt`

Departamento de Engenharia Eletrotécnica
Instituto Superior de Engenharia do Porto
Rua Dr. António Bernardino de Almeida, 431, 4200–072 Porto

November, 2022

Acknowledgements

I would like to thank Prof. Benedita Malheiro for her constant sharing of knowledge, experience and prompt availability throughout the development of the project. I would like to express my gratitude to Gonçalo Gonçalves for his support and advisory. I would also like acknowledge José Ornelas for proposing the project. Special thanks to INESC TEC for believing in me and receiving me so welcomingly and for providing me with the resources to execute this project.

I could not have undertaken this journey without my mom and my girlfriend. A special thanks to both, Ilídia and Maria João, for the patient and understanding which you showed through the years, for always believing in me, and, at the same time, for always pushing me to be a better version of myself.

Pedro Ferreira

Abstract

In recent years, not only the Service-Oriented Architecture (SOA) became a popular paradigm for the development of distributed systems, but there has been significant progress in terms of their testing. Nonetheless, the multiple testing platforms available fail to fulfil the specific requirements of the Moodbuster platform from Instituto de Engenharia de Sistemas e Computadores, Tecnologia e Ciência (INESC TEC) – provide a systematic process to update the test knowledge, configure and test several Representational State Transfer (REST) Application Programming Interface (API) instances. Moreover, the solution should be implemented as another REST API.

The goal is to design, implement and test a platform dedicated to the testing of REST API instances. This new testing platform should allow the addition of new instances to test, the configuration and execution of sets of dedicated tests, as well as, collect and store the results. Furthermore, it should support the updating of the testing knowledge with new test categories and properties on a needs basis.

This dissertation describes the design, development and testing of APIbuster, a platform dedicated to the testing of REST API instances, such as Moodbuster. The approach relies on the creation and conversion of the test knowledge ontology into the persistent data model followed by the deployment of the platform (REST API and user dashboard) through a data modelling pipeline.

The APIbuster prototype was thoroughly and successfully tested considering the functional, performance, load and usability dimensions. To validate the implementation, functional and performance tests were performed regarding each API call. To ascertain the scalability of the platform, the load tests focused on the most demanding functionality. Finally, a standard usability questionnaire was distributed among users to establish the usability score of the platform.

The results show that the data modelling pipeline supports the creation and subsequent updating of the testing platform with new test attributes and classes. The pipeline not only converts the testing knowledge ontology into the corresponding persistent data model, but generates a fully operational testing platform instance.

Keywords: APIbuster, data modelling pipeline, REST API, SOA, testing platform.

Resumo

Nos últimos anos, o desenvolvimento de sistemas distribuídos do tipo *Service-Oriented Architecture* (SOA) popularizou-se, tendo ocorrido significativos progressos em termos de testagem. Contudo, as múltiplas plataformas de testagem existentes não satisfazem as necessidades específicas de testagem de projetos *Application Programming Interfaces* (API) do tipo *Representational State Transfer* (REST) como o Moodbuster do Instituto de Engenharia de Sistemas e Computadores, Tecnologia e Ciência (INESC TEC). O INESC TEC necessita de um processo sistemático de atualização, configuração e testagem de múltiplas instâncias API REST. Adicionalmente, esta solução deverá ser implementada como mais uma API REST.

O objetivo é conceber, implementar e testar uma plataforma de testagem de instâncias API REST. Esta nova plataforma deverá permitir a adição de instâncias de teste, configuração e execução de grupos de testes, assim como, obter e salvaguardar os resultados. Deverá ainda viabilizar a atualização do conhecimento do domínio mediante a especificação de novas categorias e atributos de teste.

Esta dissertação descreve a conceção, desenvolvimento e testagem da plataforma APIbuster dedicada à testagem de instâncias API REST, como as do projecto Moodbuster. A abordagem baseia-se na definição e conversão da ontologia de representação do conhecimento sobre a testagem de API REST no correspondente modelo persistente de dados, seguida da criação da plataforma (REST API e portal do utilizador) através de um processamento sequencial dedicado.

O protótipo da APIbuster foi testado detalhadamente com sucesso em relação à funcionalidade, desempenho, carga e usabilidade. Foram efetuados testes funcionais e de desempenho a cada chamada da API para validar a implementação. Para determinar a escalabilidade da plataforma, os testes de carga focaram-se na funcionalidade mais exigente. Finalmente, o questionário de usabilidade foi distribuído entre os utilizadores para definir a usabilidade da plataforma desenvolvida.

Os resultados mostram que o processamento sequencial desenvolvido suporta a criação e a subsequente atualização, com novos atributos e categorias, da plataforma de testagem. Este processo não converte apenas a ontologia no modelo de dados

persistente, mas gera uma instância atualizada e operacional da plataforma.

Palavras-Chave: APIbuster, data modelling pipeline, REST API, SOA, testing platform.

Contents

List of Figures vii

List of Tables ix

Listings xi

List of Acronyms xiii

1 Introduction 1

1.1 Context 1
1.2 Motivation 2
1.3 Goals 2
1.4 Document Structure 2

2 Distributed Tests 3

2.1 Service-Oriented Architectures 3
2.2 Software Tests 4
2.3 Automated Test Frameworks 5
2.3.1 Testing Framework Categories 6
2.3.2 Popular Testing Frameworks 7
2.3.3 Framework Comparison 10
2.4 Summary 10

3 Problem Statement and Proposed Solution 13

3.1 Problem Statement 13
3.2 Requirements Specification 14
3.2.1 Actors 14
3.2.2 Functionalities 14
3.2.3 Technical Requirements 15
3.3 Proposed Solution 16
3.3.1 Architecture 16
3.3.2 Adopted Technologies 17
3.4 Development Methodology 17
3.5 Summary 18

4 APIbuster Testing Platform **19**
4.1 Data Modelling Pipeline 19
4.1.1 APIbuster OWL 19
Class Properties 21
Class Associations 22
4.1.2 APIbuster UML 22
Protégé — Export to UML 23
Visual Paradigm — UML import 23
4.1.3 APIbuster ERD 25
4.1.4 APIbuster ERD — DB 26
4.2 API 27
4.2.1 LoopBack 4 27
Datasource 27
Models 28
Repositories 28
Controllers 29
4.3 Portal — User Dashboard 32
4.3.1 Vue.js 32
4.4 Test Definition and Execution 33
4.5 Summary 34

5 APIbuster Testing **35**
5.1 Pipeline Testing 35
5.2 API Testing 36
5.2.1 Functional and Performance Tests 36
5.2.2 Load Tests 37
5.2.3 Usability Tests 37
5.3 Summary 38

6 Conclusion **39**
6.1 Achievements 39
6.2 Future Work 40

References **41**

Appendix A umlImport.sh **45**

Appendix B dbImport.sh **49**

Appendix C setupAPI.sh **53**

Appendix D User Dashboard **57**

List of Figures

2.1 Steps for a better Test Automation Approach [6] 5
2.2 Most popular JavaScript back-end testing frameworks [14] 7

3.1 UML diagrams of the data modelling pipeline 16
3.2 APIbuster back-end and front-end 16
3.3 Gantt chart 18

4.1 Ontology representation 20
4.2 UML class diagram representation of the ontology 24
4.3 ERD representation of the ontology 26

D.1 APIbuster user dashboard: Home page 57
D.2 APIbuster user dashboard: API page 58
D.3 APIbuster user dashboard: Adding an API 58
D.4 APIbuster user dashboard: Updating an API 59
D.5 APIbuster user dashboard: Methods page 59
D.6 APIbuster user dashboard: Adding a Method 60
D.7 APIbuster user dashboard: Updating a Method 60
D.8 APIbuster user dashboard: Test Groups page 61
D.9 APIbuster user dashboard: Adding a Test Group 61
D.10 APIbuster user dashboard: Updating a Test Group 62
D.11 APIbuster user dashboard: Tests page 62
D.12 APIbuster user dashboard: Adding a Test 63
D.13 APIbuster user dashboard: Updating a Test 63
D.14 APIbuster user dashboard: Results of the latest Test 64

List of Tables

3.1 Functionalities 14

4.1 Ontology class definitions 21
4.2 API methods 30

5.1 Pipeline: functional and run-time results 35
5.2 API: functional and performance results 36
5.3 API: load results 37

Listings

4.1 datasourceConfig.json 28
4.2 apiRepositoryConfig.json 29
4.3 apiControllerConfig.json 29
4.4 <testGroupName>_id<testGroupId>.test.js 33
A.1 umlImport.sh 45
B.1 dbImport.sh 49
C.1 setupAPI.sh 53

List of Acronyms

API Application Programming Interface

BDD Behaviour-Driven Development

CLI Command Line Interface

CPU Central Processing Unit

CRUD Create, Read, Update and Delete

CSS Cascading Style Sheets

CSV Comma-Separated Values

DB Database

DDL Data Definition Language

DOM Document Object Model

E2E End-to-End

ECMA European Computer Manufacturers Association

EER Enhanced Entity Relationship Model

ERD Entity Relationship Diagram

ESM European Computer Manufacturers Association Script Modules

HTML HyperText Markup Language

INESC TEC Instituto de Engenharia de Sistemas e Computadores, Tecnologia e Ciência

OWL Ontology Web Language

PDF Portable Document Format

QA Quality Assurance

RAM Random Access Memory

REST	Representational State Transfer
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SPA	Single Page Applications
SUS	System Usability Scale
TDD	Test-Driven Development
UI	User Interface
UML	Unified Modelling Language
XMI	eXtensible Markup Language Metadata Interchange
XML	eXtensible Markup Language

Chapter 1

Introduction

This chapter provides a context to this dissertation along with the motivations that led to its execution, expresses the goals that are expected to be achieved and a brief description of this document structure.

1.1 Context

Software testing assesses whether the developed software meets client requirements. Distributed systems are considerably more challenging to test than their standalone counterparts since problems may occur due to code and network malfunctioning. This is the case of Service-Oriented Architecture (SOA) projects, which rely on the interaction between multiple nodes, called services, potentially located across the network, to provide complex functionalities transparently to the end user.

The Instituto de Engenharia de Sistemas e Computadores, Tecnologia e Ciência (INESC TEC) develops multiple Representational State Transfer (REST) SOA projects that require ample testing. The number, diversity and complexity of these projects at INESC TEC require the constant addition, (re)configuration and execution of multiple and different types of tests. Development and support teams are normally small and team members change frequently, which makes development and testing hard to coordinate. Since existing software testing tools do not provide a solution to this problem, this project aims to design, develop and test a software testing platform that supports the systematic definition, reconfiguration and execution of REST API project tests.

1.2 Motivation

Firstly, there is the personal challenge to design an evolving testing platform to test the multiple REST API of undergoing and future projects at INESC TEC.

Secondly, the design and implementation of such a large scale project constitutes a unique learning and personal growth experience. Moreover, since experienced software testers are highly sought professionals, this project provides an opportunity to acquire knowledge and master a key technology.

Lastly, there is the personal contribution to the increase in software quality and productivity of the development teams, both for active and new members, at INESC TEC.

1.3 Goals

This dissertation has two main goals. The first goal is to create a data modelling pipeline to easily create and update the test API. This allows the platform to accommodate new test categories and attributes seamlessly adapting itself to the evolving needs of any particular project.

The second goal is to develop the corresponding Web testing platform prototype, composed of a REST API and a user dashboard. This allows platform users to test in real-time the REST API of undergoing projects and the development team to adapt the platform for the testing of future projects.

1.4 Document Structure

The remaining five chapters of this dissertation are structured as follows:

- Chapter 2 provides a brief introduction to the context of this dissertation, namely by addressing service-oriented architectures, software testing and automated test frameworks.
- Chapter 3 details the problem, the multiple requirements, the proposed solution, as well as, the work plan for the framework.
- Chapter 4 documents the solution development steps.
- Chapter 5 presents the multiple tests to which the platform was submitted as well as the results.
- Chapter 6 summarises the accomplishments and proposes new features that might be implemented in the future.

Chapter 2

Distributed Tests

This chapter introduces the concept and categories of distributed software tests along with a brief comparison of multiple test frameworks.

2.1 Service-Oriented Architectures

Although there is no clear definition for a SOA, it can be described as a communication mediation paradigm for service consumers and providers in dynamic settings, supporting the access to multiple services and functionalities [1, 2, 3]. This allows a transparent data exchange between parties.

A distributed SOA architecture is an architectural model to deploy, combine and use application components distributed through the Internet, as required. The keyword is "service", which means a precisely-defined, self-contained and independent function from any other service ("black-box like structure") [2, 4].

By offering different heterogeneous interfaces, interoperable distributed systems provide connectivity between multiple services. The usage of distributed SOA architectures provides to companies a more flexible method to build systems, helping ensure the scalability of their businesses by creating a strong foundation and allowing the expansion of their business logic in the future. A company is able to simply create a new service and intertwine it with its existing platform [2] — a smooth operation which, for any company, is crucial for building a strong reputation. For any company, a smooth operation is a must to build a strong reputation. To make

sure everything runs as intended, all services must be thoroughly tested beforehand. There are several techniques and frameworks to perform software testing.

2.2 Software Tests

Software testing is achieved through dynamic verification and validation of the behaviour of a program based on a predetermined set of test cases, against an expected behaviour [5].

Tests are performed throughout the whole software development and maintenance phases, therefore software testing is regarded as a process embedded in the software development process [5].

Testing activities are performed at different levels. Unit testing focuses on testing specific units or components that were developed. Integration testing happens when such units are integrated into a system. Last but not least, system testing is done once the whole system is tested [5].

A testing technique can be introduced as a way of providing systematic guidelines of designing test cases. The main purpose is to be as systematic as possible in identifying the software behaviours that are being subjected to test.

Black-box tests are a kind of testing mechanism focused on the end-to-end system using inputs and outputs.

White-box tests are tests which evaluate the internal sub-components of a system. In other words, this is a technique based on information about how the software was designed and implemented.

Defect-based tests are meant to uncover categories of likely or predefined faults.

Model-based tests are based on models such as Statecharts, Finite State Machines and others.

Acceptance tests are meant to emulate user experience. These tests give the costumers confidence that the application has the required features and they are functioning correctly.

Integration tests are based on a different unit which is responsible for supplying the test cases, passing the parameters to the unit being tested, collecting results and presenting them to the tester.

Unit tests focus on a single unit of a whole application in total isolation, usually, a single class or function. Ideally, the tested component is free of side effects so it is as easy to isolate and test as possible.

This dissertation focuses mainly on Acceptance tests, Integration tests and Unit tests. These are the tests most used to assess REST API instances. While Acceptance tests are mostly oriented towards customers, Integration and Unit tests are essential for developers to have confidence on their software.

2.3 Automated Test Frameworks

An automated testing framework integrates a set of assumptions, concepts, practices and libraries that provide support for automated software testing. The framework helps developers and testers to write test code more efficiently while finding bugs quicker. Moreover, they provide an improved test automation flow, ensuring that all essential steps are included, as shown in Figure 2.1.

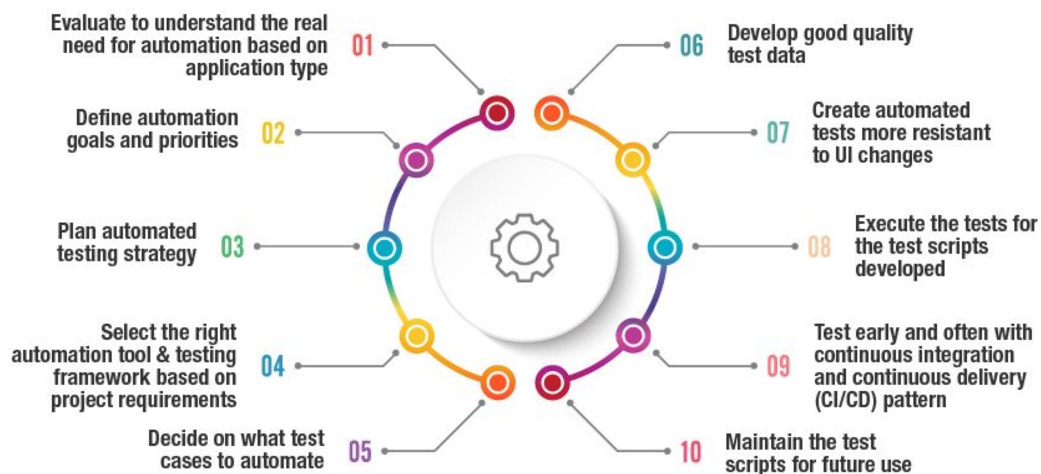


Figure 2.1: Steps for a better Test Automation Approach [6]

It is possible to divide Figure 2.1 into two main stages:

- Preparation — Evaluate the need to automate testing, define goals and priorities, plan a strategy, select the right testing tools based on the project requirements and decide which test cases to automate.
- Automation — Develop good quality test data, create automated tests resistant to user interface changes, execute the developed automated tests, test early and often with continuous integration and store the tests for future use.

This dissertation addresses the automation stage since it aims to design, develop and deploy a testing platform for users to easily test, debug and maintain their REST API.

2.3.1 Testing Framework Categories

The most commonly used automated testing frameworks types include modular testing, test libraries, keyword-driven testing, data-driven testing, hybrid automated testing [7], test-driven [8], behaviour-driven [9] and ontology-based test modelling [10], [11], [12], [13].

Modular Testing frameworks normally create small, independent and reusable scripts for the applications being tested. The main drive is the modular packaging of programming, where small scripts are combined together to form a much bigger test case. This is the easiest framework to use and master, while increasing the scalability of the automated tests [7].

Test Library frameworks are application-dependent. They break down the application into procedures and functions and create dedicated modules and function library files for each application. Unlike modular testing, they do not rely on scripts [7].

Keyword-Driven Testing frameworks are application-dependent. They require data tables and keywords to manipulate the testing scripts in order for the application and data to be tested. The functionality of the application is written to a data table along with the execution steps for each intended test. This sort of framework minimises the code volume required to implement a large number of test cases, as the same tests can be reused with different data tables, corresponding to different individual test cases [7].

Data-Driven Testing frameworks read the input and output data from data files and then load them into variables. These variables are not only used to store the input values but also output validation values. Throughout the testing program, the script reads the data files and records the test status and information. This type of framework is meant to reduce the number of test scripts required to complete all test cases. Similarly to keyword-driven testing, very little code is required to contemplate a large number of test cases [7].

Hybrid Automated Testing frameworks combine all or part of the aforementioned types, reinforcing their strengths and cancelling their weaknesses [7].

Test-Driven Development (TDD) frameworks are frequently used with Agile Software Development and imply the usage of automatic testing tools. This testing framework supports the creation of tests and provides a reliable pass/-fail indication and repeated test running [8].

Behaviour-Driven Development (BDD) frameworks were derived from test-driven development and are mainly focused on the system behaviour. The idea

is to describe how the application should behave in a very simple user/business-focused language [9]. By doing so, it helps even the non-technical users to easily analyse and understand the tests.

Ontology-based Test Modelling frameworks explore ontologies to represent the structure and relationships within the test knowledge [10, 11, 12, 13]. By default, ontologies are reusable within the same application domain. Considering the testing of distributed Web services, the performed literature survey did not find any reusable ontology.

The most popular testing paradigms nowadays are test-driven and behaviour-driven development [8]. Ontology-based test modelling, although less popular, allows for application domain re-usability which will prove useful when creating, or updating, domain knowledge. This work adopts both ontology-based test modelling and behaviour-driven development frameworks.

2.3.2 Popular Testing Frameworks

Choosing a testing framework is an essential step when developing new Web projects. Figure 2.2 compares eleven of the most popular back-end testing frameworks according to usage since 2016 [14]. These include[1]:

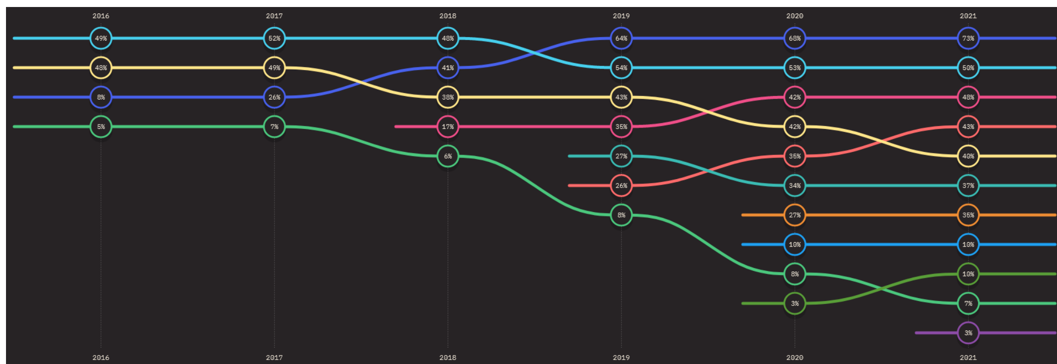


Figure 2.2: Most popular JavaScript back-end testing frameworks [14]

- **Jest** [2] is used and recommended by Facebook and is officially supported by React. Since it is a framework supported by a tech giant, it is inevitable that when programmers run into problems, aside from official documentation, there is a lot of support from fellow developers [15]. Jest has good performance, great for continuous deployments; is compatible with other applications developed in Angular, Node, Vue.js and other babel-based projects; has an extended

[1] The frameworks are identified by the corresponding colours in the list below.
[2] Jest — `https://jestjs.io/`

API which will likely have most of the functionalities a developer requires; and has an active development and community, which updates the framework regularly. Unfortunately, Jest is a slow runner [15].

- **Mocha** [3] provides developers with just the base test structure. Designed originally to work with Node, it is now capable of working with a larger range of frameworks (Angular, React, Vue) with some configuration. The most popular assertion library used with Mocha is Chai[4], but Assert, Should.js and Better-assert can also be used [15]. Mocha is very lightweight and simple; has a flexible configuration since there are a lot of libraries that can be used; and is compatible with European Computer Manufacturers Association Script Modules (ESM). Developers using Mocha have a harder time setting up the framework since multiple libraries need to be configured. Some inconsistencies can potentially happen due to the utilised plugins. Mocha's documentation is known to be weak [15].

- **Storybook** [5] is more of a UI testing tool. It provides an isolated testing environment to test various components. Although Storybook is listed as one of the most popular back-end testing frameworks it is very hard to compare it to other frameworks.

- **Cypress** [6] runs entirely through a browser (Chrome, Firefox and Edge). It is commonly know for its End-to-End (E2E) testing capabilities. Pre-defined user behaviours can be followed through this tool and reports are given for potential differences upon each deployment [15]. Cypress uses E2E testing; has timeline snapshot testing allowing developers or Quality Assurance (QA) testers to review specific timelines; is steady and dependable, compared to other frameworks; has a rich documentation; and is fast, normally around 20 ms. Unfortunately, Cypress can only run tests in a single browser [15].

- **Jasmine** [7] is a famous BDD tool. It is not exclusively a JavaScript testing tool since it can also be used with other programming languages such as Ruby or Python [15]. Jasmine has a straightforward API, with clean and easy to understand syntax, and has internal assertion libraries. Jasmine creates a lot of test globals, which might be a downside on specific scenarios. Async testing is hard to achieve with Jasmine [15].

[3] Mocha — `https://mochajs.org/`
[4] Chai — `https://www.chaijs.com/`
[5] Storybook — `https://storybook.js.org/`
[6] Cypress — `https://www.cypress.io/`
[7] Jasmine — `https://jasmine.github.io/index.html`

- **Puppeteer** [8] is a Node library created by Chrome's Development Team. It is a testing framework that enables users to test chrome pages. It allows users to simulate actions done in a Chrome browser. Some of the most popular usage of this application is to: generate screenshots and Portable Document Format (PDF) from web pages, keyboard input simulation and form submissions and testing chrome extensions [15]. Being a headless but full-fledged browser, it's an ideal choice when testing Single Page Applications (SPA) [15].

- **React Testing Library** [9] is not a test runner like Jest. Actually, they can be used simultaneously. React Testing Library is a set of tools and functions which help with Document Object Model (DOM) access and various actions performed on them. Rendering components into Virtual DOM, searching and interacting with it [15]. React Testing Library is recommended by React Team and provides great documentation; is lightweight and written to test React applications; and has seen an increase in popularity, so it is bound to have community support. React Testing Library provides no shallow testing, when rendering components with an increased amount of children it does not allow to skip rendering portions of those children, which may increase testing times [15].

- **WebdriverIO** [10] is an automation framework for Web and mobile applications. It allows users to create scalable and stable test suites. The ability to allow hybrid testing, native mobile apps, native desktop apps and Web apps is what distinguishes this framework from the others mentioned [15]. WebdriverIO is multi-platform testing framework compatible with other assertion libraries and frameworks, such as, Mocha, Jasmine and Cucumber, and is simple and fast. Being hard to debug and its poor documentation are the only disadvantages of using WebdriverIO [15].

- **Playwright** [11] is another automation framework best used for E2E testing. It was built and maintained by Microsoft and is able to run across multiple browser engines - Chromium, Webkit and Firefox [15]. Playwright is backed by a trusted company; supports multiple languages, such as JavaScript, Java, Python, and .NET C#; can be used by other testing frameworks such as Mocha, Jest and Jasmine; allows multiple browser testing; and can emulate mobile devices, geolocation and permissions. It is, unfortunately, in its early stages, which means community support is still lacking, and does not support real devices when doing mobile devices testing, only virtual mobile devices [15].

[8] Puppeteer — `https://pptr.dev/`
[9] React Testing Library — `https://testing-library.com`
[10] WebdriverIO — `https://webdriver.io/`
[11] Playwright — `https://playwright.dev/`

- **AVA** [12] is a test runner and takes advantage of JavaScript's async nature to run tests concurrently, increasing performance. It has a very simple API and snapshot testing, which allows users to know when their application's UI changes unexpectedly. AVA does not allow users to group tests and has no built-in mocking, but it can be added using Sinnon.js [15].

- **Vitest** [13] was made by the same team that developed Vite. It is a native test runner that provides a compatible DOM that allows developers to use it as a replacement to Jest in most projects. Vitest is optimised for performance and uses Worker threads to run as much as possible in parallel. Vitest's intends to become the best Test Runner choice for Vite projects [15]. Vitest is multi-threaded, has native ESM and TypeScript support, and in-source testing, since it provides a way to run tests within user's source code along with the implementation, similar to Rust's module tests. Vitest is still in an early adoption phase, which means community support is still reduced [15].

2.3.3 Framework Comparison

After analysing multiple test frameworks, it is not possible to clearly choose the best one in this group.

Each framework has its strong and weak points. Most provide the essential for testing, which is a testing environment paired together with mechanisms that ensure that given x, y values are always returned.

Jest, or other framework with big communities, are a strong choice for a first approach to software testing. Having help from other testers through forums will fasten the testing development. But if a user requires a broader API that might contain unique features, then Mocha is a smart choice, since it is largely extensible. For E2E testing, the best frameworks should be Cypress, Puppeteer or Playwright.

For this project, Mocha will be the chosen framework along with Chai, which is one of the supported assertion libraries. Being the most extensible framework is a big factor since it provides more options for developing. Furthermore, since it is already used in the API development framework of some existing INESC TEC projects, it allows for a cleaner integration with these projects.

2.4 Summary

This dissertation focuses on the automation stage and considers Acceptance tests, Integration tests and Unit tests. Moreover, it adopts the ontology-based test modelling technique and behaviour-driven testing frameworks. Ontologies specify the

[12] AVA — `https://github.com/avajs/ava`
[13] Vitest — `https://vitest.dev/`

structure of a knowledge domain and, as such, are reusable within the same application domain. Behaviour-driven testing ensures the widest compatability between frameworks. For this project, the test knowledge ontology will be created and the selected testing framework was Mocha.

The next chapter will present the problem as well as the proposed solution.

Chapter 3

Problem Statement and Proposed Solution

This chapter presents the problem and solution requirements followed by the proposed design and development solution and the work plan.

3.1 Problem Statement

The stable operation of a distributed environment/network requires that all components work faultlessly. As such, proper testing needs to be made on a regular basis. Although multiple testing frameworks exist, none provide a systematic and automated way to test several REST Web service API instances that can be implemented as a service of a working SOA project.

This problem was identified within the Moodbuster INESC TEC project [16]. Moodbuster is a research platform that allows researchers and practitioners to create and conduct online interventions regarding different psychological problems, such as depression or anxiety disorders. On one hand, it allows patients to communicate and provide live mood updates to therapists. On the other hand, therapists can, based on the incoming data, manage patient treatments and chat with patients. To do so, Moodbuster combines multiple Web portals and mobile applications (iOS and Android) into a multiplatform project, exposing multiple REST API instances.

Considering the size and complexity of Moodbuster, multiple and constant tests need to be performed. Since the assigned development team is small, it becomes

hard to coordinate development with testing deadlines. The implementation of a testing platform capable of reliably building tests for multiple API would improve considerably the work flow while maintaining the high quality standard associated with INESC TEC projects.

The goal of this dissertation is then to design, develop and deploy a platform prototype, composed of a test API and corresponding user dashboard portal, to test multiple REST API instances as services.

3.2 Requirements Specification

The solution needs to meet functional and technical requirements. The actors that will be using the framework, the required framework functionalities and the set of technologies already in use in Moodbuster were identified.

3.2.1 Actors

This project encompasses two different types of users:

- End User — represents the project partners that will employ the application;
- Support Team Member — represents the members of the platform development/support team, normally supported by INESC TEC developers.

The support team member integrates the end user functionalities as well the implementation, modification and stability testing of new functionalities regarding the contemplated REST API.

3.2.2 Functionalities

The new framework should be implement the functionalities listed in Table 3.1 considering both actors.

Table 3.1: Functionalities

#	**Functionality**	**Actor**
1	Add, modify, delete a new API instance.	End User Support Team Member
2	Add, modify, delete a new Method instance for an API.	End User Support Team Member
3	Add all Methods of a REST API using a ".json" OpenAPI format file.	End User Support Team Member
4	Add, modify, delete a new Test Group instance for a Method.	End User Support Team Member
5	Add, modify, delete a new Test instance for a Test Group.	End User Support Team Member

...continued

#	Functionality	Actor
6	Run Tests from a Test Group and show Result.	End User Support Team Member
7	Run Test and show Result.	End User Support Team Member
8	Show latest Results from Tests of a Test Group.	End User Support Team Member
9	Show latest Results from a Test.	End User Support Team Member
10	Create/Update the local copy of the APIbuster platform.	Support Team Member

Both the End User and the Support Team Member interact with the framework to add, modify or delete REST API, methods, test groups or tests at will, as well as running and collecting the results of the created tests. The Support Team Member needs, in addition, to be able to create, update and deploy a local copy of the framework, allowing changes to both the testing API and portal.

3.2.3 Technical Requirements

Since Moodbuster uses a specific set of software and hardware technologies, the proposed solution should reuse them to reduce the learning curve and ensure the control over the development, deployment and usage of the new platform.

Moodbuster adopts the following Web development frameworks:

- LoopBack 4 — an open-source framework for creating dynamic REST API [17];
- Vue.js — an open-source JavaScript framework for front-end Web design development [18];
- PostgreSQL — an open-source relational database management system [19].

The platform should allow 10 to 15 connections to be made concurrently. Development teams are small, two to three programmers, and partners are also, normally, comprised of the same number of users. As such, by assuring that the platform holds about double the amount of maximum expected connections, a proper and fluid usability of the platform is preserved.

Considering these requirements, the physical platform should have a standard 6 core processor with 8 GB of Random Access Memory (RAM).

The next section takes into consideration all the project requirements and provides a solution to the problem capable of fully answering the different functionalities.

3.3 Proposed Solution

The proposed testing framework, called APIbuster, supports the refinement of the test knowledge, the addition of new REST API instances to test, the creation and execution of multiple test entries as well as the collection and storage of the results. This chapter provides an overview of the solution. Chapter 4 specifies to great extent each phase of development.

3.3.1 Architecture

The designed APIbuster testing platform integrates three main components (Figures 3.1 and 3.2):

1. Data modelling pipeline — the processing pipeline that generates the entity relationship data model from the test domain ontology;

2. APIbuster — the testing API (also implemented as REST Web service);

3. Portal — the APIbuster user dashboard.

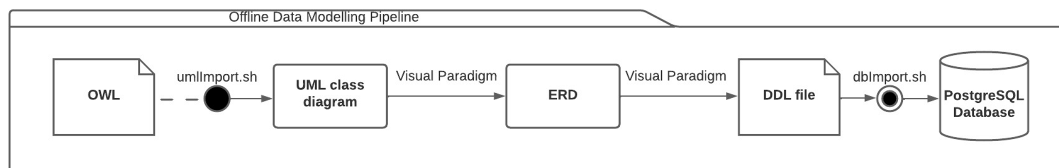


Figure 3.1: UML diagrams of the data modelling pipeline

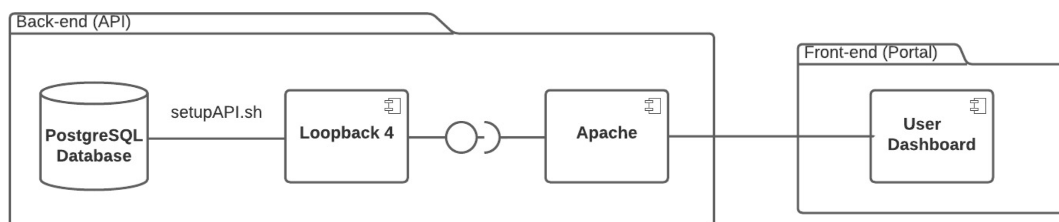


Figure 3.2: APIbuster back-end and front-end

The input of the data modelling pipeline is the test domain knowledge ontology and the output is the data model of the testing platform. To this end, the created Ontology Web Language (OWL) file, which models the test domain knowledge, follows a set of sequential transformations to obtain the Enhanced Entity Relationship Model (EER) data diagram. This is achieved, as shown in Figure 3.1, by converting from OWL into Unified Modelling Language (UML), UML into an Entity Relationship Diagram (ERD) and, lastly, ERD into an usable EER. This semi-automated

pipeline allows the refinement of the test knowledge ontology and, consequently, of the data model, by adding new classes, e.g., new test categories, and new class properties for existing classes, e.g., new test attributes, but will not support changes to class properties previously created.

The persistent data model of the APIbuster platform corresponds to the resulting EER data model. Once the APIbuster component is deployed, the users can interact with the platform, i.e., add new API instances to test, define the corresponding test suite and collect the results, using the APIbuster user dashboard portal component.

3.3.2 Adopted Technologies

Taking the project requirements in consideration, it is possible to choose an appropriate set of development technologies. The creation of the ontology adopts Protégé, an open source ontology editor and framework for building systems [20]. The data modelling pipeline partially relies on the Visual Paradigm modelling tool [21].

The selected Web service development technologies are based on those already in use in the Moodbuster platform. These include Loopback 4 as back-end (API) and Vue.js as front-end (portal) development frameworks. The chosen relational database server to store the platform's database is PostgreSQL database management system.

3.4 Development Methodology

Since the implementation of the whole project relies solely on one person no standard development methodology, such as SCRUM, Agile or DevOps, was followed.

In this case, a "trial-by-error" approach was adopted regarding the creation of the pipeline. It's already planned that various iterations of the OWL file will be made since there might be errors and updates needed. Creating a pipeline will make changes faster, stabler and, at the same time, allow for a more structured architecture.

Using this method will also allow for an API creation process based on an ontology to be made capable of creating any type of API that follows the same methodologies and utilises the same technologies as the proposed solution.

The Gantt chart of the project development is presented in Figure 3.3. It comprises the following tasks:

T1 Creation of the test domain ontology [1st to 8th week];

T2 Development of the data modelling pipeline and creation of the persistent database compliant with the test domain ontology [8th to 14th week];

T3 Development of the testing API (APIbuster) which interacts with the persistent test database [14th to 18th week];

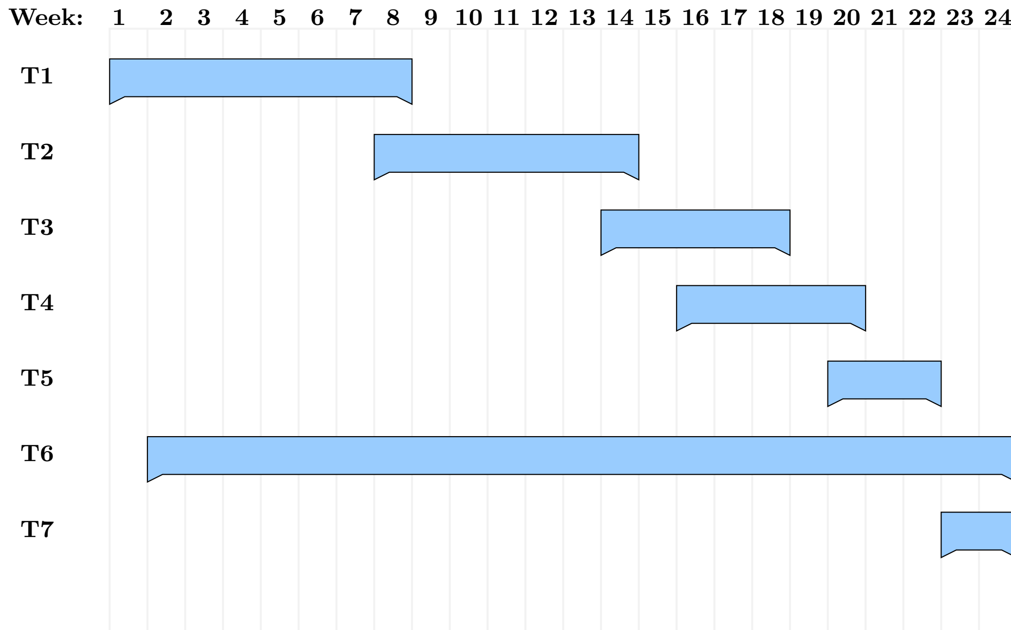


Figure 3.3: Gantt chart

T4 Development of the APIbuster Web portal for users to interact with the platform [16$^{\text{th}}$ to 20$^{\text{th}}$ week];

T5 APIbuster testing [20$^{\text{th}}$ to 22$^{\text{th}}$ week];

T6 Writing of the dissertation [1$^{\text{st}}$ to 24$^{\text{th}}$ week];

T7 Development of the dissertation PowerPoint presentation [23$^{\text{th}}$ to 24$^{\text{th}}$ week].

3.5 Summary

The designed testing platform is compliant with the adopted test domain ontology and supports multiple API testing. Moreover, end users and support team members can interact with the platform through a Web dashboard.

The following chapter details the development of the APIbuster testing platform.

Chapter 4

APIbuster Testing Platform

This chapter documents and details the development of the APIbuster platform prototype.

4.1 Data Modelling Pipeline

As seen in Chapter 3, the APIbuster platform is divided into three major components. The data modelling pipeline is the first and most complex one since its role is to create the database model for the whole framework.

The data modelling pipeline implements a series of data transformations that converts the created OWL file, through UML and ERD, to the desired EER model. These transformations rely on two tools and two dedicated scripts:

1. OWL file — generated by Protégé;
2. UML class diagram file — produced by “umlImport.sh” (Appendix A) and Visual Paradigm;
3. ERD — produced by Visual Paradigm;
4. EER — produced by Visual Paradigm and “dbImport.sh” (Appendix B).

4.1.1 APIbuster OWL

Unlike existing test frameworks, the APIbuster test framework revolves around an ontology to support the systematic knowledge update process. This allows for a

better scalability and provides users with a simpler and robust development and update pipeline.

The framework's ontology was created with Protégé. Protégé is an ontology editor and framework for building intelligent systems created by Stanford [20]. The ontologies can be generated using the OWL, which is a Semantic Web language designed to represent rich and complex knowledge about things, groups of things, and relations between things[1].

The APIbuster OWL ontology defines a tree comprising a set of nodes and leafs corresponding to classes with and without sub-classes, as shown in Figure 4.1. On the one hand, the conversion of OWL nodes into the UML class diagram, creates all child classes as generalisations of the parent class, i.e., a single UML class component is created for the parent class with each property being the fusion of all its child class properties. On the other hand, the conversion of OWL leafs result in UML classes without child relationships.

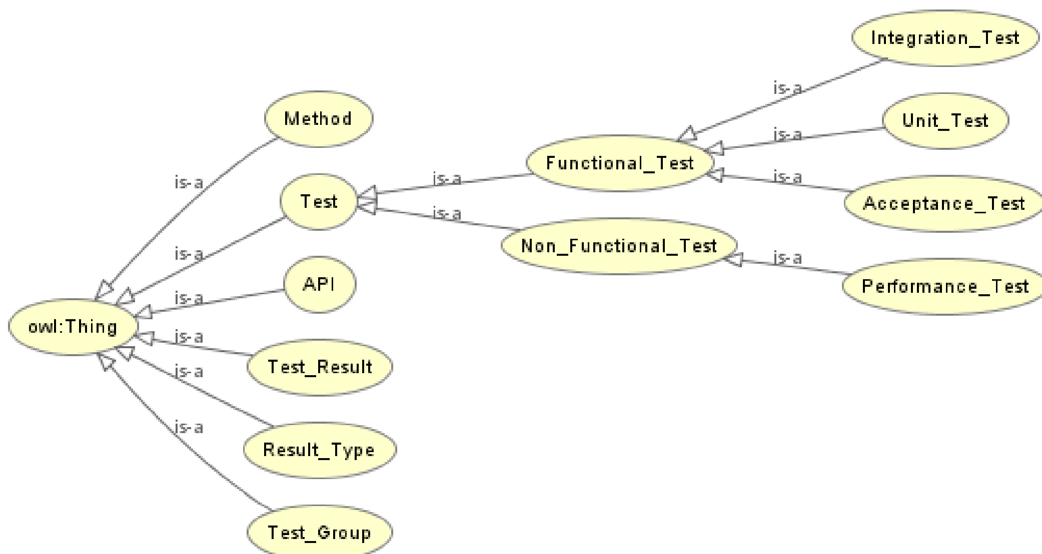


Figure 4.1: Ontology representation

By analysing Figure 4.1 it is easy to identify the classes representation category. The leaf classes are:

- API — representing a REST API web app instance to be tested;
- Method — representing a REST API method used to consult, create, modify or delete instances from the DB (GET, POST, PATCH, PUT and DEL);
- Test_Group — representing a test group instance for a specific method;

[1]OWL W3C — `https://www.w3.org/OWL/`

- Test — representing a test instance for a specific test group;
- Test_Result — representing a test result instance for a specific test;
- Result_Type — representing one of the three result types (pass, failure or pending).

The following classes have sub-classes (are created as generalisations of the class "Test"):

- Functional_Test — representing a functional test instance;
  - Acceptance_Test — representing an acceptance test instance;
  - Integration_Test — representing an integration test instance;
  - Unit_Test — representing a unit test instance;
- Non_Functional_Test — representing a non-functional test instance;
  - Performance_Test — representing a performance test instance.

Each class has its own class properties and associations. Both components are important to establish a proper UML class representation. They later translate into UML attributes and relationships.

Class Properties

Creating class properties is achieved through the usage of "Data Properties". A property is created and then assigned to a class by simply designating the domain and range. Domain represents the class to which the created data property belongs and range is the data type: int, number, date time, string, etc. A list of each class's properties can be analysed in Table 4.1.

Table 4.1: Ontology class definitions

Class	**Data Properties**	**Object Properties**
API	api_name: string api_domain: string api_type: string	hasMethods
Method	method_body: string method_header: string method_name: string method_route: string method_type: string	hasTestGroups

...continued

Class	Data Properties	Object Properties
Test_Group	test_group_name: string test_group_duration: int test_group_start: string test_group_end: string	hasTests
Test	test_name: string test_body: string test_expect: string test_type: string	hasTestResults
Test_Result	result_title: string result_date: string result_duration: string result_error: string result_speed: string	
Test_Type	type_name: string	hasResultTypes
Functional_Test		
Non_Functional_Test		
Acceptance_Test		
Integration_Test		
Unit_Test		
Performance_Test		

Class Associations

Protégé allows the creation of associations between classes through the usage of "Object Properties". The associations between classes was created using this approach. The list of class associations is displayed in Table 4.1.

For example, the object property "hasMethods" has "API" as domain and "Method" as range. This means that the "API" class has, as object properties, a group "hasMethods" of "Method" class.

To complete the association, this object property must be added as a sub-class of the domain class and assigned a quantitative declaration (some, min, max) as well as the connective range, in this case "Method". This later assures the one to many (with 0..* multiplicity) associations and relationships between classes in the UML and ERD, respectively.

4.1.2 APIbuster UML

According to Figure 3.1, the first conversion transforms the OWL ontology file into a UML class diagram. This transformation is, unfortunately, impossible to perform with the available tools. This section elaborates on the steps necessary to obtain a faithful UML class diagram representation of the original OWL file.

Protégé — Export to UML

The Protégé wiki refers a couple of export[2] and back-end[3] plugins. From the available list, only three candidates meet the requirements of the project's pipeline:

- OWL2RDB[4] — transforms an OWL2 ontology into a relational database;
- XMI Backend[5] — exports OWL files into XMI files;
- UML Backend[6] — exports OWL files into UML files.

Although with some limitations, only the back-end solutions worked after ample tests. Moreover, since the XMI and UML back-end plugins are only compatible with Protégé 3.4, the project adopted this older version. Using the installed plugins, it is then possible to export the ontology file into an UML class representation.

Visual Paradigm — UML import

Although it is then possible to import the file to Visual Paradigm, if no changes are made, some errors occur.

The provided file is indeed a converted UML representation of the ontology, but three different errors need to be prevented.

In the first place, the file, even though it is a converted UML file, is exported without an extension. If the UML extension is not added to the file, Visual Paradigm won't even recognise the file as an acceptable UML file.

Secondly, since there is no property in Protégé that directly correlates to an association's navigability, the UML back-end plugin automatically sets it to false. This later prevents the pipeline from generating relationships properly on the ERD.

Lastly, there is no option to forcefully omit the full name of each class, object or data property. Protégé automatically gives each project property a full name[7] instead of a relative one. If no changes are done, when importing the UML file, property names won't be properly interpreted in Visual Paradigm.

All three issues are solved through the usage of the implemented bash script, "umlImport.sh" (Appendix A), as shown in Figure 3.1. This script appends the UML extension to the ontology exported file, substitutes the full property names for their relative names and changes all navigability values to true, since all created associations allow it.

[2] Protégé Export — `https://protegewiki.stanford.edu/wiki/Category:Export`
[3] Protégé Backend — `https://protegewiki.stanford.edu/wiki/Category:Backend`
[4] OWL2RDB — `https://protegewiki.stanford.edu/wiki/OWL2ToRDB`
[5] XMI Backend — `https://protegewiki.stanford.edu/wiki/XMI-Backend`
[6] UML Backend — `https://protegewiki.stanford.edu/wiki/UML-Backend`
[7] `http://semanticweb.org/{user}/ontologies/{projectname}#{propertyname}`

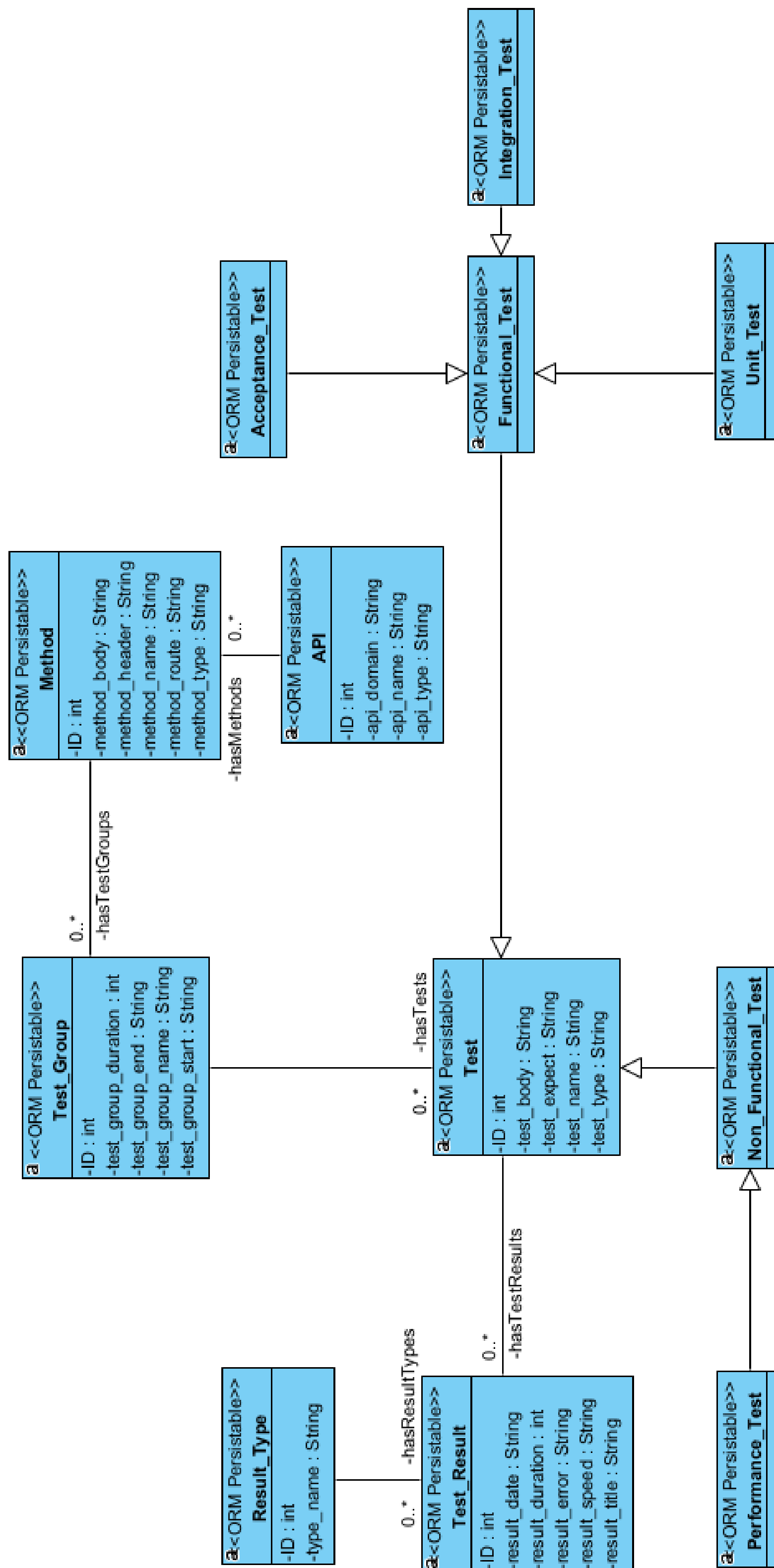


Figure 4.2: UML class diagram representation of the ontology

The UML file is then imported to Visual Paradigm, which is also achieved through the script since Visual Paradigm has a Command Line Interface (CLI) that allows the import and export of files to a project[8]. This last step can only be achieved if the project is previously created in Visual Paradigm and is not an open project.

Unfortunately, some steps cannot be fully automated since Visual Paradigm CLI doesn't have a function that allows automatic diagram creation. A new class diagram is then created, manually, as shown in Figure 4.2, by simply dragging all the imported classes to the working area of a "Class Diagram". Each block represents a class, associations are all one to many (with 0..* multiplicity), represented by the lines linking each class, and arrows represent the generalisations.

The script still creates an empty DB in PostegreSQL server, which needs to be previously installed, so that it is later possible to import the DB file created by Visual Paradigm.

4.1.3 APIbuster ERD

Following Figure 3.1 data modelling pipeline representation, the next conversion is made from UML to ERD.

Unfortunately, this entire process needs to be fully manual since there are no available CLI functions to help with this step's automation.

Assigning "ORM Persistable" stereotype to all classes, in the UML diagram, is a requirement for this conversion. Visual Paradigm has a function called "Synchronize to Entity Relationship Diagram" under "Tools" -> "Hibernate" window option which only works if correctly set up. Without the "ORM Persistable" stereotype this synchronisation would fail as it would not recognise the different classes to convert.

After synchronising the UML into ERD, a new diagram is created, as shown in Figure 4.3. This synchronisation will create all foreign keys and relationships between entities according to the associations previously created. That is why all "ID" properties were automatically added to each entity.

In this case, several tables were grouped into a single table since their respective UML classes all generalise to the same superclass. All their properties are condensed in Test table and a "Discriminator" property is automatically created.

The class properties, which are represented as a tree in the ontology class (Table 4.1), had no object or data properties. Moreover, during the ontology creation phase, all classes are used to create the ERD.

[8]Visual Paradigm CLI import — https://www.visual-paradigm.com/support/documents/vpuserguide/124/255/7348_exportingand.html

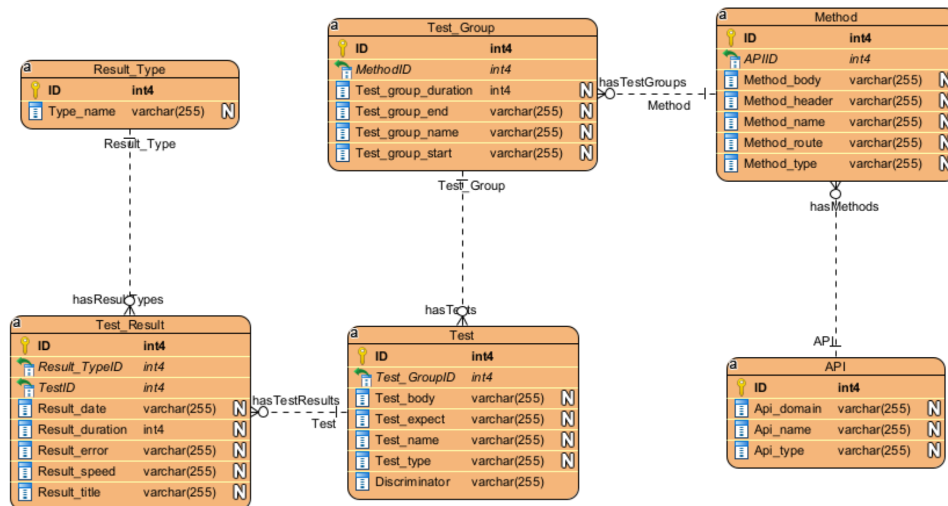


Figure 4.3: ERD representation of the ontology

4.1.4 APIbuster ERD — DB

The last conversion of the pipeline is made after achieving a correct ERD representation.

Visual Paradigm also allows DB creation based on ERD, which, unfortunately, is also a semi-automated process. Database configurations cannot be automated, but Visual Paradigm also allows database export through the CLI[9].

The first step is the “Database Configuration” under “Tools” -> “DB” window option. This is the step that Visual Paradigm CLI doesn’t allow to do automatically. Setting up the configuration for the type of DB the user is utilising can be made through a driver file or manually. To establish a connection, both cases require:

- hostname;
- port;
- database name;
- user;
- password.

[9] Visual Paradigm CLI database export — `https://www.visual-paradigm.com/support/documents/vpuserguide/124/255/7350_generatingor.html`

Using a second script, “dbImport.sh” (Appendix B), represented in Figure 3.1, a Data Definition Language (DDL) file containing the database schema is generated and the previously created PostgreSQL database is now populated with all tables by importing it to the PostgreSQL server. This can be done using the CLI functionalities that PostgreSQL provides.

Before importing the DDL file, the script still updates the table creation commands so it allows every table to cascade delete simply by adding “ON DELETE CASCADE” to the end of the each table creation query. This means that when deleting a parent instance, all child instances will also be removed.

After finishing this procedure, all the conversions are completed and a correct PostgreSQL database is built from the initial ontology file.

4.2 API

With a working pipeline capable of creating and updating the database, an API capable of managing that information can be created. For this outcome, the used framework to quickly create a Web REST API was LoopBack 4.

4.2.1 LoopBack 4

LoopBack 4 is a highly extensible, open-source Node.js and TypeScript framework based on Express. It enables developers to quickly create API and microservices composed from back-end systems such as databases and Simple Object Access Protocol (SOAP) or REST services [17]. This framework was chosen since it was required for the new platform to be produced on a similar web developing framework as other projects at INESC TEC, such as the Moodbuster project. There was also extensive knowledge on how to build a project through it and the fact that it uses Mocha internally as a test framework was also an advantage. This expedites the development process allowing for a more reliable product.

Loopback 4 has some CLI functionalities that allow users to quickly create projects, add datasources, models, repositories, relations and create controllers. Some of these commands were utilised in the third script, “setupAPI.sh” (Appendix C), in order to further automate the process. Unfortunately, there are some inputs that require manual input since Loopback 4 is still being updated nowadays and it’s functionalities are still being renewed. Even though some CLI options are documented, they are partially usable or not usable at all.

Datasource

The script creates the datasource that will be used by the API. In this case, creates a connection between the API and the DB. To automate this, using a config file as

stated in the documentation[10,11], a “datasourceConfig.json” file was developed.

```
{
    "name": "postgres",
    "connector": "postgresql",
    "url": "",
    "host": "localhost",
    "port": 5432,
    "user": "postgres",
    "password": "<PASSWORD>",
    "database": "APIbusterDB"
}
```

Listing 4.1: datasourceConfig.json

In this case, the previously created and populated PostgresSQL database, called “APIbusterDB” is connected to the API.

Loopback 4 will interpret this new datasource as “postgres” and will establish a connection to it using the “postgresql” connector. It has no url since the DB is hosted on the same machine as the created API, which means the host name is “localhost” and the port 5432. To finish the connection a user connection needs to be established to the database named “APIbusterDB”. In this case, the user “postgres” and password “<PASSWORD>” will be used. “<PASSWORD>” is only a placeholder for the real password.

Models

The Loopback 4 CLI also allows model creation based on the “discover” command[12]. This command, using “postgres” datasource connection, automatically creates models based on available database tables. It creates a model file, with all data properties, foreign keys and relations for each table.

This option requires the user to manually input the “camelCase” option since it can’t be skipped through command options.

Repositories

Contrary to models creation, this step is fully automated and requires no manual input from the user. A configuration file was made according to documentation[13]. All repository config files are inside the “configFiles/repositories” folder.

[10]LoopBack4 datasource — https://loopback.io/doc/en/lb4/DataSource-generator.html
[11]LoopBack4 datasource configuration — https://loopback.io/doc/en/lb4/PostgreSQL-connector.html#connection-pool-settings
[12]LoopBack4 model discover — https://loopback.io/doc/en/lb4/Discovering-models.html
[13]LoopBack4 repository — https://loopback.io/doc/en/lb4/Repository-generator.html

```
1     {
2         "name": "Api",
3         "datasource": "postgres",
4         "model": "Api",
5         "id": "id",
6         "repositoryBaseClass": "DefaultCrudRepository"
7     }
```

Listing 4.2: apiRepositoryConfig.json

Each repository has it's own configuration file. In this example, Loopback 4 creates a new repository named "Api" that uses the previously created "postgres" datasource and model "Api". It is also provided the id attribute, "id", and it is also defined the type of repository to be created. In this case, a "DefaultCrudRepository".

Controllers

This last step also requires no manual input as Loopback 4 proceeds as intended during controller creation. An example of a controller configuration file isn't provided in the controller page, but a configuration representation was found in Loopback 4 GitHub forums[14].

```
1     {
2         "name": "api",
3         "modelName": "Api",
4         "controllerType": "REST Controller with CRUD functions",
5         "repositoryName": "ApiRepository",
6         "idType": "number",
7         "httpPathName": "/apis"
8     }
```

Listing 4.3: apiControllerConfig.json

Each controller has it's own configuration file. In this example, Loopback 4 creates a new controller named "api" and based on model "Api". It is a "REST Controller with CRUD functions", meaning it is a controller with basic GET, POST, PUT, PATCH, DEL functions. It uses the "ApiRepository" previously created, its idType is a "number" and the base httpPathName is set to "/apis".

Loopback 4 automatic controller creation only implements standard Create, Read, Update and Delete (CRUD) methods as shown in Table 4.2.

[14] LoopBack4 GitHub forum controllers — https://github.com/loopbackio/loopback-next/issues/1844

Table 4.2: API methods

Controller	Method	Request	Input	Output
API	GET	'/apis'		Array of API instances
API	GET	'/apis/count'		Number of API instances
API	GET	'/apis/id'		Object representing a specific API instance
API	POST	'/apis'	Object representing API instance to create	Object representing the created API instance
API	PUT	'/apis/id'	Object representing API instance properties to update	Object representing the modified API instance
API	PATCH	'/apis/id'	Object representing API instance properties to update	
API	DELETE	'/apis/id'		
Methods	GET	'/methods'		Array of Methods instances
Methods	GET	'/methods/count'		Number of Methods instances
Methods	GET	'/methods/id'		Object representing a specific Methods instance
Methods	POST	'/methods'	Object representing Methods instance to create instance	
Methods	PUT	'/methods/id'	Object representing Methods instance properties to update	Object representing the modified Methods instance
Methods	PATCH	'/methods/id'	Object representing Methods instance properties to update	
Methods	DELETE	'/methods/id'		
Test Groups	GET	'/test-groups'		Array of Test Groups instances
Test Groups	GET	'/test-groups/count'		Number of Test Groups instances
Test Groups	GET	'/test-groups/id'		Object representing a specific Test Groups instance
Test Groups	POST	'/test-groups'	Object representing Test Groups instance to create	Object representing the created Test Groups instance
Test Groups	PUT	'/test-groups/id'	Object representing Test Group instance properties to update	Object representing the modified Test Groups instance

. . . continued

Controller	Method	Request	Input	Output
Test Groups	PATCH	'/test-groups/id'	Object representing Test Group instance properties to update	
Test Groups	DELETE	'/test-groups/id'		
Tests	GET	'/tests'		Array of Tests instances
Tests	GET	'/tests/count'		Number of Tests instances
Tests	GET	'/tests/id'		Object representing a specific Tests instance
Tests	POST	'/tests'	Object representing Tests instance to create	Object representing the created Tests instance
Tests	PUT	'/tests/id'	Object representing Tests instance properties to update	Object representing the modified Tests instance
Tests	PATCH	'/tests/id'	Object representing Tests instance properties to update	
Tests	DELETE	'/tests/id'		
Test Results	GET	'/test-results'		Array of Test Results instances
Test Results	GET	'/test-results/count'		Number of Test Results instances
Test Results	GET	'/test-results/id'		Object representing a specific Test Results instance
Test Results	POST	'/test-results'	Object representing Test Results instance to create	Object representing the created Test Results instance
Test Results	PUT	'/test-results/id'	Object representing Test Results instance properties to update	Object representing the modified Test Results instance
Test Results	PATCH	'/test-results/id'	Object representing Test Results instance properties to update	
Test Results	DELETE	'/test-results/id'		
Result Types	GET	'/result-types'		Array of Result Types instances
Result Types	GET	'/result-types/count'		Number of Result Types instances
Result Types	GET	'/result-types/id'		Object representing a specific Result Types instance

. . . continued

Controller	Method	Request	Input	Output
Result Types	POST	'/result-types'	Object representing Result Types instance to create	Object representing the created Result Types instance
Result Types	PUT	'/result-types/id'	Object representing Result Type instance properties to update	Object representing the modified Result Types instance
Result Types	PATCH	'/result-types/id'	Object representing Result Type instance properties to update	
Result Types	DELETE	'/result-types/id'		

For this project some extra methods were developed:

- apiMethods — returns an array of methods created for a specific API instance through an uploaded "openapi.json" file;
- runTestgroup — returns an array of test results from a test group's tests after running all test group's tests;
- runTest — returns a test result for a specific test after running it;
- latest — returns the latest test result for a specific test;
- findLatestResults — returns an array of the latest test results for a test group's tests.

Since the logic behind the implementation can't be automatically generated, all these methods had to be manually implemented and added to the API.

4.3 Portal — User Dashboard

The portal was created as a way for the different actors to interact with the created API. The users require an UI that allows them to view, create, modify and delete new API, methods, test groups and tests.

Taking into consideration all the solution requirements mentioned in Chapter 3, the chosen framework to create this portal was Vue.js.

4.3.1 Vue.js

Vue was chosen for its approachability, performance and versatility. The platform is designed to work with both JavaScript and TypeScript[15]. Built on top of standard HTML, Cascading Style Sheets (CSS) and JavaScript and has an intuitive API and

[15] Vue.js languages — `https://vuejs.org/about/faq.html#should-i-use-javascript-or-typescript-with-vue`

world-class documentation, rarely requires manual optimisation and it is incredibly scalable [18].

Vue is also a framework previously and currently used by some INESC TEC's projects, such as Moodbuster. Re-utilising this technology when producing a platform would reduce the learning curve normally needed when engaging with a new front-end framework.

The portal (Appendix C) presents four inter-linked views:

- API page — visualisation, creation, modification and deletion of API instances;
- Methods page — visualisation, creation, modification and deletion of method instances of a given API;
- Test Groups page — visualisation, creation, modification and deletion of test groups instances. Also allows for test group testing and visualisation of the latest test results of a given test group;
- Tests page — visualisation, creation, modification and deletion of test instances of a given test group. Also allows for specific test instance testing and visualisation of its latest test result.

4.4 Test Definition and Execution

Each test result is obtained either through "runTestgroup" or "runTest" methods. Either method is divided in four separate steps:

Test file building - in this step a test file, based on the "API", "Methods", "Test Groups" and "Tests" instances, is built containing, two "describe" statements (which describe the test suite), and one, or multiple, "it" statements (which represent the various tests to be executed).

```
1 describe('<methodName> tests', async () => {
2   describe('<testGroupName> tests', async () => {
3     it('<testName>', async () => {
4     <testBody>
5     const res = await axios.<methodType>("<apiDomain><
          methodRoute>");
6     <testExpected>
7   });
8 });
```

Listing 4.4: <testGroupName>_id<testGroupId>.test.js

Running test file - both methods utilise Mocha's CLI to run the created test files. Results are output to a ".json" file unique to each test group/test run.

Test result instances creation - based on the output file, new test result instances are created.

File deletion and values returned - after each instance is created, ".json" files are deleted and the new created test result values are returned to the user.

4.5 Summary

Throughout this chapter a detailed explanation of the multiple components of the project was presented. It was possible to implement a trustworthy data modelling pipeline capable of transforming an ordinary ontology file into a enhanced entity relationship model, allowing users to create a DB based on their ontology files.

A detailed description of the creation process of the API was shown, where a full list of the API methods was provided, and a brief explanation was given about the main features of the dashboard.

Upon the completion of the APIbuster prototype, a series of tests are to be made. Functional tests on every used API call, load tests on the "heaviest" call and performance tests on the system usability. By testing the system it will be possible to unveil any problems that might be undiscovered so far, while showing how stable the prototype actually is.

Chapter 5

APIbuster Testing

This chapter documents and details the functional, load and performance test results for the APIbuster platform.

5.1 Pipeline Testing

The data modelling pipeline was tested in a computer with a 6 core Central Processing Unit (CPU) and 16 GB RAM. The experiment consisted on the updating of the data model of the API. Table 5.1 presents the functional and run-time results.

Table 5.1: Pipeline: functional and run-time results

Functionality	**Step**	**Result**	**Run-time** (ms)
Create/Update the local copy of the APIbuster platform	umlImport.sh	Pass	354
	VP Editing	Pass	22 000
	dbImport.sh	Pass	647

Each step of the pipeline was tested separately. The first step involves converting APIbuster's OWL file into a UML, using "umlImport.sh" (Appendix A), and took less than 0.4 s. The second step was the slowest step to be completed since it is a manual conversion, done through Visual Paradigm. It produces an ERD from the UML generated by the "umlImport.sh" script. This step took approximately 22 s, but this value may vary not only from user to user but depend on the dimension of the updated ontology. Lastly, the "dbImport.sh" (Appendix B) script, creates or updates the DB model and, consequently, the API. This automated step took around 0.7 s.

This functional test clearly shows the run-time difference between automated and manual steps, i.e., between the duration of steps 1 and 3 and step 2.

5.2 API Testing

In order to achieve a robust and trustworthy application, tests to the framework need to be made. Three different types of tests were made using different platforms and/or techniques. Functional and performance tests were implemented with the aid of Postman [22], load tests were made using Apache JMeter [23] and usability tests through the System Usability Scale (SUS) [24] questionnaire. This tests were made using Moodbuster as the test target and all tests were conducted using a computer with a 6-core CPU and 16 GB RAM.

The server hosting APIbuster has the following computational infrastructure:

- 6-core CPU;
- 8 GB RAM.

5.2.1 Functional and Performance Tests

These tests were repeated ten times using Postman to determine the average and standard deviation of the exchanged data (B) and time response (ms) of each functionality listed in Table 3.1. The results were calculated with Calculator.tech[1] and can be observed in Table 5.2.

Table 5.2: API: functional and performance results

Functionality	**Method**	**Result**	**Size** (B)	**Latency** (ms)	
				μ	σ
Obtain a list of all API instances	GET	Pass	590	22.40	1.26
Obtain a list of all Methods of an API	GET	Pass	515	25.60	1.17
Obtain a list of all Test Groups of a Method	GET	Pass	567	20.00	0.94
Obtain a list of all Tests of a Test Group	GET	Pass	574	23.70	3.30
Create an API	POST	Pass	465	21.30	4.08
Create a Method	POST	Pass	521	21.90	0.99
Create a Test Group	POST	Pass	450	22.20	1.29
Create a Test	POST	Pass	550	23.80	2.61
Update an API	PATCH	Pass	347	120.10	66.55
Update a Method	PATCH	Pass	347	96.70	30.62
Update a Test Group	PATCH	Pass	347	83.50	80.25
Update a Test	PATCH	Pass	347	94.70	26.45
Delete an API	DELETE	Pass	347	131.90	82.24
Delete a Method	DELETE	Pass	347	88.40	37.91
Delete a Test Group	DELETE	Pass	347	108.00	24.93

[1]Calculator — `https://www.calculators.tech/variance-calculator`

. . . continued

Functionality	**Method**	**Result**	**Size** (B)	**Latency** (ms)	
				μ	σ
Delete a Test	DELETE	Pass	347	103.90	16.94
Run a Test	GET	Pass	542	465.50	20.17
Run a Test Group	GET	Pass	748	470.60	18.24
Obtain latest test results	GET	Pass	748	23.60	2.95

The results show that "GET" and "POST" requests have the lowest latency, for both mean and variance. Exceptions are the API calls that execute and return test results. The latter present the highest latency, making them interesting targets for load testing.

5.2.2 Load Tests

Table 5.2 shows that the most demanding API call (largest data exchange and latency) occurs when the user runs a group of tests. To evaluate the impact of this worst case on the server load, Apache JMeter was configured to make 10, 100 and 1000 concurrent test group requests. Table 5.3 holds the results.

As expected, the average latency increases considerably with the number of concurrent requests. When compared with the average latency of a single request in Table 5.2 (470.60 ms), the results in Table 5.3 are 12 % higher with 10 requests, almost three times higher with 100 requests and forty times higher with 1000 requests.

Nonetheless, these latency results comply with the requirements of the project, which stated that it should allow between 10 to 15 concurrent users. In this case, the latency will be above 0.50 s and below 1.80 s, which is acceptable.

Table 5.3: API: load results

Requests	**Functionality**	**Method**	**Results**	**Size** (B)	**Latency** (ms)	
					μ	σ
10	Run a test group	GET	10	749	528.30	51.50
100	Run a test group	GET	100	749	1779.10	245.58
1000	Run a test group	GET	1000	749	10 810.58	4631.82

5.2.3 Usability Tests

To test the usability of the APIbuster platform, a Google form[2] was elaborated according to the SUS model [24]. This defines a usability scale that allows anyone to quickly evaluate a system's usability.

The form implements a ten-question template the user can adapt to suit the application under test. Participants will rank each question from one to five based on how much they agree with the statement they are reading. Five means they agree completely, one that they disagree vehemently [24].

[2]Google forms — `https://www.google.com/forms/about/`

1. “I think that I would like to use this system frequently”;
2. “I found the system unnecessarily complex”;
3. “I thought the system was easy to use”;
4. “I think that I would need the support of a technical person to be able to use this system”;
5. “I found the various functions in this system were well integrated”;
6. “I thought there was too much inconsistency in this system”;
7. “I would imagine that most people would learn to use this system very quickly”;
8. “I found the system very cumbersome to use”;
9. “I felt very confident using the system”;
10. “I needed to learn a lot of things before I could get going with this system”.

SUS is scored from 0 — 100 and the average score is around 68 points. An overview of how the scores are measured:

- 80.3 or higher is an A. Users enjoy the platform and will recommend it;
- 68 or thereabouts leads to a C. The platform corresponds to expectations but could improve;
- 51 or under represents an F. Usability should be the priority for now.

The scores of odd and even questions are calculated differently. One point is removed from the score of each odd question, whereas five points are subtracted from the score of each even question. The resulting values are then added and the sum multiplied by 2.5.

Only five participants completed the form. Nonetheless, based on the available data, APIbuster achieved a score of 85 — grade A. This means that the respondents consider the platform user-friendly and useful.

5.3 Summary

The platform is fully operational. Every functional test worked without issues and with acceptable performance. The server is able to answer multiple concurrent requests as required without losing functionality. Lastly, the users seem to appreciate the navigability and the functionalities of the platform.

Chapter 6

Conclusion

This chapter contains the closing arguments to the dissertation.

6.1 Achievements

The APIbuster prototype answers the problem stated in Chapter 3. The designed solution was successfully implemented and tested without major setbacks. The implementation adopts the Web development technologies already used at INESC TEC, such as Loopback 4, PostgreSQL and Vue.js, and state-of-the-art modelling tools like Protégé and Visual Paradigm.

The data modelling pipeline starts with the creation of the test knowledge ontology and ends, after multiple conversions, with the persistent data model of the APIbuster DB. This approach partially automates the creation and update of the persistent data model based on the domain's knowledge ontology. Since Protégé and Visual Paradigm lack the tools to fully automate this pipeline, two additional dedicated scripts, presented in Appendix A and Appendix B, were developed to provide the missing functionalities.

The automation of the API development process was straightforward. Loopback 4 has a CLI that allows users to quickly create a working REST API based on a persistent data model. These steps are described in Chapter 4. Additional API calls were created taking into consideration the project's aim and specifications. The

integrated testing framework relies on Loopback 4, Mocha[1] and Chai[2] to perform tests and collect results.

The user dashboard, created using Vue.js, provides all users with a UI to interact with the APIbuster platform. This simple portal implements the functionalities listed in Table 3.1.

Based on the test results shown in Chapter 5, APIbuster complies with the identified project requirements. Functional, performance, load and usability tests were successfully performed.

6.2 Future Work

APIbuster can be refined in multiple fronts and even additional research could probably be made in order to further automate the data modelling pipeline.

As it stands, APIbuster is mainly for developers since it assumes users already know how to create software tests, specially using Mocha and Chai assertion library. A login system capable of separating users and their API tests was not created. This would prove useful in the future to prevent unauthorised users from making changes to the created test cases. Since multiple partners could use the API, it is essential to guarantee the safety of their tests.

Another possible addition to the project is to provide code snippets to automatically define new tests. This would allow users not familiarised with the testing framework to learn quicker and create more complex tests faster.

The updating of the test knowledge ontology does not support modifying existing class properties. This could be solved by developing new scripts. These scripts would need to export the affected class instance data to a Comma-Separated Values (CSV) file, reuse the existing scripts to update the DB model, alter the contents of the CSV file according to the class property changes, and import the data back into the respective table.

Finally, to fully automate the data modelling pipeline there is the need to automatically convert from XMI, the format of the UML file, to DDL, the format of the ERD database model schema. This possibility is currently being investigated.

[1] Mocha — `https://mochajs.org/`
[2] Chai — `https://www.chaijs.com/`

References

[1] S.-W. Chen, Y.-T. Tseng, and T.-Y. Lai, "The design of an ontology-based service-oriented architecture framework for traditional chinese medicine healthcare," in *2012 IEEE 14th International Conference on e-Health Networking, Applications and Services (Healthcom)*, pp. 353–356, 2012. [Cited on page 3]

[2] N. Josuttis, *SOA in Practice: The Art of Distributed System Design.* O'Reilly Media, 2007. [Cited on page 3]

[3] A. Kumar, A. K. Pandey, and M. Singh, "A novel testing framework for soa based services," in *International Conference for Convergence for Technology-2014*, pp. 1–4, 2014. [Cited on page 3]

[4] OpenGroup, "SOA source book - what is SOA?." `https://collaboration.opengroup.org/projects/soa-book/pages.php?action=show&ggid=1314`. Last accessed: 2021-08-21. [Cited on page 3]

[5] E. Souza, R. Falbo, and N. Vijaykumar, "Using ontology patterns for building a reference software testing ontology," in *2013 17th IEEE International Enterprise Distributed Object Computing Conference Workshops*, pp. 21–30, 2013. [Cited on page 4]

[6] TestingXperts, "Trending." `https://www.testingxperts.com/blog/test-automation-frameworks`. Last accessed: 2021-08-21. [Cited on pages vii and 5]

[7] Z. Sun, Y. Zhang, and Y. Yan, "A web testing platform based on hybrid automated testing framework," in *2019 IEEE 4th Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)*, vol. 1, pp. 689–692, 2019. [Cited on page 6]

[8] X. Wang and P. Xu, "Build an auto testing framework based on selenium and fitnesse," in *2009 International Conference on Information Technology and Computer Science*, vol. 2, pp. 436–439, July 2009. [Cited on pages 6 and 7]

[9] Tricentis Staff, "What is BDD (Behavior-Driven Development)?." `https://www.tricentis.com/blog/bdd-behavior-driven-development`, 2022. Last accessed: 2022-10-12. [Cited on pages 6 and 7]

[10] A. Freitas and R. Vieira, "An ontology for guiding performance testing," in *2014 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT)*, vol. 1, pp. 400–407, Aug 2014. [Cited on pages 6 and 7]

[11] M. El Hassan Charaf and M. Bergannou, "An ontology-based approach for modeling the distributed test issues," in *2018 International Conference on Electronics, Control, Optimization and Computer Science (ICECOCS)*, pp. 1–6, Dec 2018. [Cited on pages 6 and 7]

[12] K. Jayashree, "Trapping runtime faults from web service fault ontology," *Journal of Computational and Theoretical Nanoscience*, vol. 17, no. 8, pp. 3759–3764, 2020. [Cited on pages 6 and 7]

[13] G. Tebes, L. Olsina, D. Peppino, and P. Becker, "Specifying and analyzing a software testing ontology at the top-domain ontological level," *Journal of Computer Science & Technology*, vol. 21, pp. 126–145, 2021. [Cited on pages 6 and 7]

[14] S. Greif, "Testing." `https://2021.stateofjs.com/en-US/libraries/testing/`, 2022. Last accessed: 2022-10-18. [Cited on pages vii and 7]

[15] M. Taleb, "JavaScript unit testing frameworks in 2022: A comparison." `https://raygun.com/blog/javascript-unit-testing-frameworks/`, 2022. Last accessed: 2022-10-18. [Cited on pages 7, 8, 9, and 10]

[16] INESC TEC, "Moodbuster 2.0." `https://moodbuster2.inesctec.pt/`, 2022. Last accessed: 2022-10-16. [Cited on page 13]

[17] LoopBack, "Loopback 4." `https://loopback.io/doc/en/lb4/`, 2022. Last accessed: 2022-06-18. [Cited on pages 15 and 27]

[18] E. You, "Vue.js - the progressive JavaScript framework." `https://vuejs.org/`, 2014. Last accessed: 2022-06-18. [Cited on pages 15 and 33]

[19] The PostgreSQL Global Development Group, "PostgreSQL: The world's most advanced open source relational database." `https://postgresql.org/`, 2022. Last accessed: 2022-06-18. [Cited on page 15]

[20] Protégé, "The protégé project: A look back and a look forward." `https://protege.stanford.edu/`, 2015. Last accessed: 2022-06-18. [Cited on pages 17 and 20]

[21] Visual Paradigm, "Visual paradigm." `https://www.visual-paradigm.com/`, 2022. Last accessed: 2022-06-18. [Cited on page 17]

[22] Postman, "Postman API platform | sign up for free." https://www.postman.com/, 2022. Last accessed: 2022-10-18. [Cited on page 36]

[23] Apache, "Apache JMeter™." https://jmeter.apache.org/, 2022. Last accessed: 2022-10-18. [Cited on page 36]

[24] N. Thomas, "How to use the system usability scale (SUS) to evaluate the usability of your website." https://usabilitygeek.com/how-to-use-the-system-usability-scale-sus-to-evaluate-the-usability-of-your-website/, 2022. Last accessed: 2022-10-18. [Cited on pages 36 and 37]

Appendix A

umlImport.sh

```
1  #!/usr/bin/env bash
2  source "config"
3
4  set -e
5
6  usage() {
7    echo -e "
8      $0 usage:
9        -c
10         Creates '.uml' file.
11       -u
12         Updates '.uml file'.
13       -h
14         Display help."
15 }
16
17 while getopts ":cuh" arg; do
18   case $arg in
19     c)
20       create=true
21       ;;
22     u)
23       update=true
24       ;;
25     h | *)
```

```
26        usage
27        ;;
28    esac
29  done
30
31  info() {
32    msg="\e\n[1;33;44m ===> $1 \e[0m"
33    [ "$2" != "nobreak" ] && msg="${msg}\n"
34    echo -n -e $msg
35  }
36
37  # check for single option usage
38  [ $# -eq 0 ] && usage
39  ([ -n "$create" ] || [ -n "$update" ]) && [ $# -gt 2 ] || \
40  ([ -z "$create" ] && [ -z "$update" ]) && [ $# -gt 1 ] && \
41  info "You must use a single option. Use -h for more information."
       && exit 0
42
43  # Create function
44  create() {
45    info "Creating file.."
46
47    # change file name
48    echo -e "File: "$file
49    mv $file $uml
50
51    echo -e "UML: "$uml
52
53    # change class paths to relative path
54    sed -i "s/$pathName/$empty/" $uml
55    sed -i "s/$navigableFalse/$navigableTrue/" $uml
56
57    info "Changes applied to $uml."
58
59    # imports file to Visual Paradigm
60    cd "C:\Program Files\Visual Paradigm 16.3\scripts"
61
62    info "Importing .uml to APIbuster project."
63
64    ImportXMI.bat -project $project -file $filePath
65
66    # creates PostgreSQL DB if doesn't exists
67    cd "C:\Program Files\PostgreSQL\14\bin"
68
69
70    # print time in nanoseconds
71    echo "In case DB already exists"
72
73    PGPASSWORD=admin createdb -U postgres APIbusterDB
```

```
74 }
75
76 # Update function
77 update() {
78   info "Updating files.."
79
80   # change file name
81   echo -e "File: "$file
82   mv $file $uml
83
84   echo -e "UML: "$uml
85
86   # change class paths to relative path
87   sed -i "s/$pathName/$empty/" $uml
88   sed -i "s/$navigableFalse/$navigableTrue/" $uml
89
90   info "Changes applied to $uml."
91
92   # imports file to Visual Paradigm
93   cd "C:\Program Files\Visual Paradigm 16.3\scripts"
94
95   info "Importing .uml to APIbuster project."
96
97   ImportXMI.bat -project $project -file $filePath
98 }
99
100 # Delete function
101 delete() {
102   info "Deleting files.."
103   echo -e "\e[1;31mIf directories exist, they will be removed and
        files will be permanently lost."
104   echo -e "Press any key to proceed\t(Ctrl+C to cancel)\n"
105   read -n 1 -s -r
106
107   deleteDirs
108 }
109
110 # Running the scripted functions
111 if [ $create ]; then create;
112 elif [ $update ]; then update; fi
```

Listing A.1: umlImport.sh

Appendix B

dbImport.sh

```
#!/usr/bin/env bash
source "config"

set -e

usage() {
  echo -e "
    $0 usage:
      -c
        Creates '.uml' file.
      -u
        Updates '.uml file'.
      -h
        Display help."
}

while getopts ":cuh" arg; do
  case $arg in
    c)
      create=true
      ;;
    u)
      update=true
      ;;
    h | *)
```

```
26        usage
27        ;;
28    esac
29  done
30
31  info() {
32    msg="\e\n[1;33;44m ===> $1 \e[0m"
33    [ "$2" != "nobreak" ] && msg="${msg}\n"
34    echo -n -e $msg
35  }
36
37  # check for single option usage
38  [ $# -eq 0 ] && usage
39  ([ -n "$create" ] || [ -n "$update" ]) && [ $# -gt 2 ] || \
40  ([ -z "$create" ] && [ -z "$update" ]) && [ $# -gt 1 ] && \
41  info "You must use a single option. Use -h for more information."
       && exit 0
42
43  changeddl() {
44    # add ON DELETE CASCADE
45    sed -i "/REF.*(ID);/ s/;$/ ON DELETE CASCADE;/" $ddlFile
46
47    info "Importing to DB.."
48
49    # creates tables
50    cd "C:\Program Files\PostgreSQL\14\bin"
51
52    PGPASSWORD=admin psql -U postgres -d APIbusterDB -f $ddl
53
54    echo -e "Done.."
55  }
56
57  # Create function
58  create() {
59    info "Creating DDL.."
60
61    # saves current folder
62    currentFolder=$(pwd)
63
64    # updates DDL file
65    cd "C:\Program Files\Visual Paradigm 16.3\scripts"
66    GenerateORM.bat -project $project -out $currentFolder -code -ddl
          Create
67    cd $currentFolder
68
69    echo -e "Done.."
70
71    changeddl
72  }
```

```
73
74 # Update function
75 update() {
76   info "Updating DDL.."
77
78   # saves current folder
79   currentFolder=$(pwd)
80
81   # updates DDL file
82   cd "C:\Program Files\Visual Paradigm 16.3\scripts"
83   GenerateORM.bat -project $project -out $currentFolder -code -ddl
         Update
84   cd $currentFolder
85
86   echo -e "Done.."
87
88   changeddl
89 }
90
91 # Running the scripted functions
92 if [ $create ]; then create;
93 elif [ $update ]; then update; fi
```

Listing B.1: dbImport.sh

Appendix C

setupAPI.sh

```
#!/usr/bin/env bash
source "config"

set -e

usage() {
  echo -e "
    $0 usage:
      -c
        Creates '.uml' file.
      -h
        Display help."
}

while getopts ":ch" arg; do
  case $arg in
    c)
      create=true
      ;;
    h | *)
      usage
      ;;
  esac
done

```

```
26 info() {
27   msg="\e\n[1;33;44m ===> $1 \e[0m"
28   [ "$2" != "nobreak" ] && msg="${msg}\n"
29   echo -n -e $msg
30 }
31
32 # check for single option usage
33 [ $# -eq 0 ] && usage
34 ([ -n "$create" ] || [ -n "$update" ]) && [ $# -gt 2 ] || \
35 ([ -z "$create" ] && [ -z "$update" ]) && [ $# -gt 1 ] && \
36 info "You must use a single option. Use -h for more information."
     && exit 0
37
38 API() {
39   # create datasource for API
40   info "Creating datasource.."
41
42   cd "/c/Users/Jackfr0stt/Desktop/APIbuster/APIbuster-API"
43
44   # lb4 datasources
45   cd configFiles/datasources
46   repositoryConfigFiles=$(ls)
47   cd ../..
48
49   for datasourceConfig in $datasourceConfigFiles; do
50     lb4 datasource --config stdin < configFiles/datasources/
         $datasourceConfig
51   done
52
53   npm run build
54
55   echo -e "Done.."
56
57   # discover models
58   info "Setting up models, repositories and controllers.."
59
60   # models
61   lb4 discover --dataSource $dataSource --schema public --all --
       optionalId --relations
62
63   # repositories
64   cd configFiles/repositories
65   repositoryConfigFiles=$(ls)
66   cd ../..
67
68   for repositoryConfig in $repositoryConfigFiles; do
69     lb4 repository --config stdin < configFiles/repositories/
         $repositoryConfig
70   done
```

```
71
72    # controllers
73    cd configFiles/controllers
74    controllerConfigFiles=$(ls)
75    cd ../..
76
77    # remove existent controller files
78    cd src/controllers
79    rm *
80    cd ../..
81
82    for controllerConfig in $controllerConfigFiles; do
83      lb4 controller --config stdin < configFiles/controllers/
          $controllerConfig
84    done
85
86    echo -e "Done.."
87
88    info "Starting API.."
89    npm start
90  }
91
92  # Running the scripted functions
93  if [ $create ]; then API; fi
```

Listing C.1: setupAPI.sh

Appendix D

User Dashboard

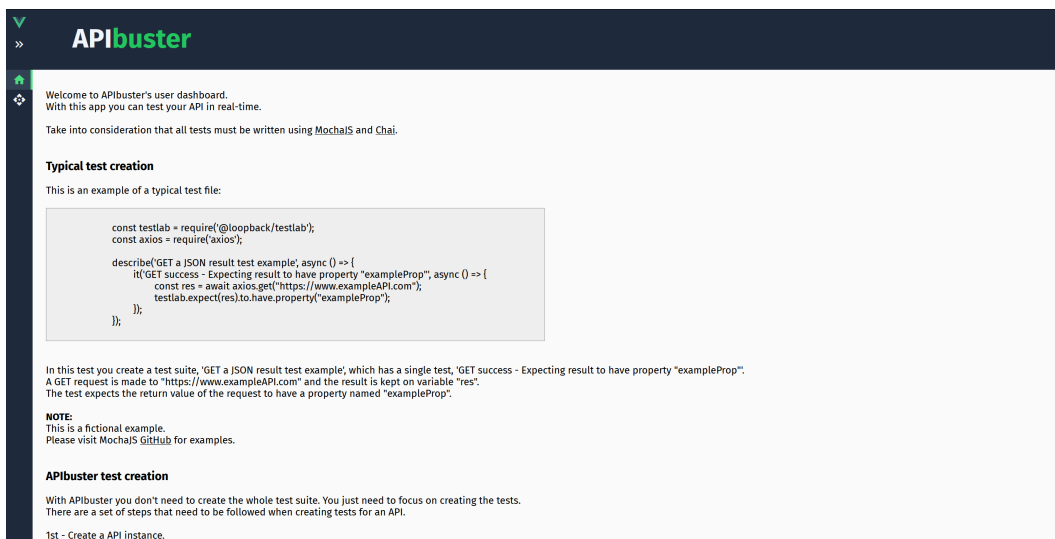


Figure D.1: APIbuster user dashboard: Home page

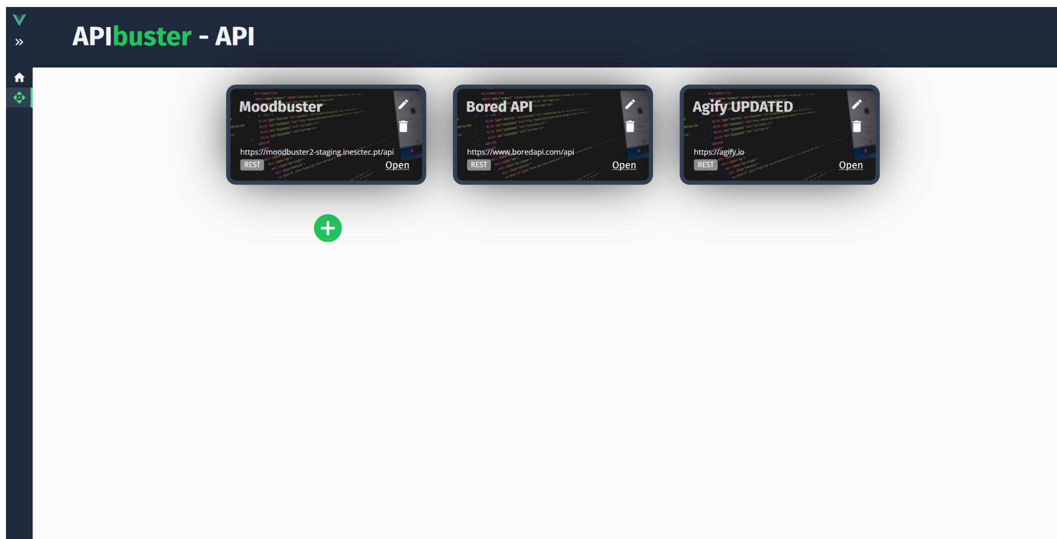


Figure D.2: APIbuster user dashboard: API page

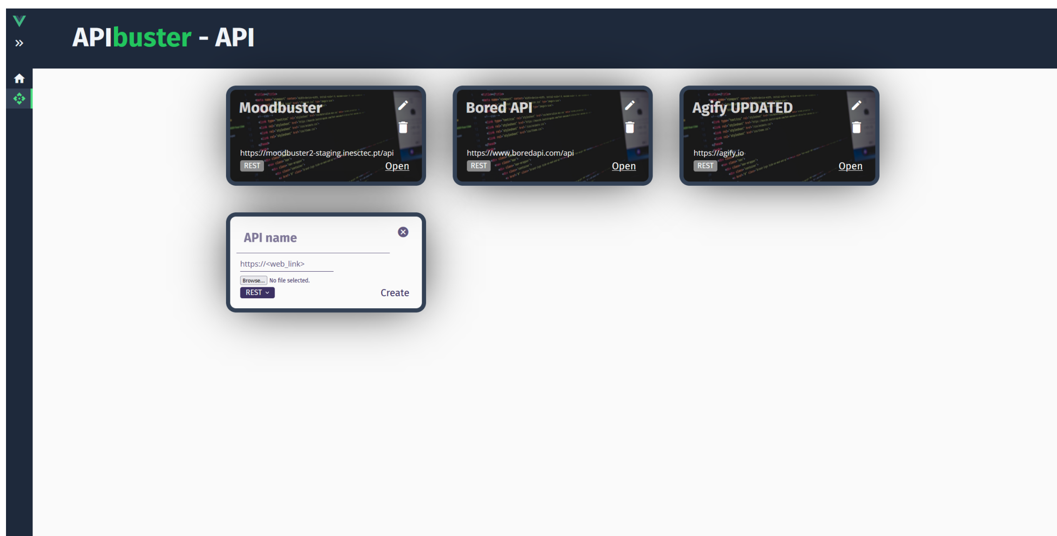


Figure D.3: APIbuster user dashboard: Adding an API

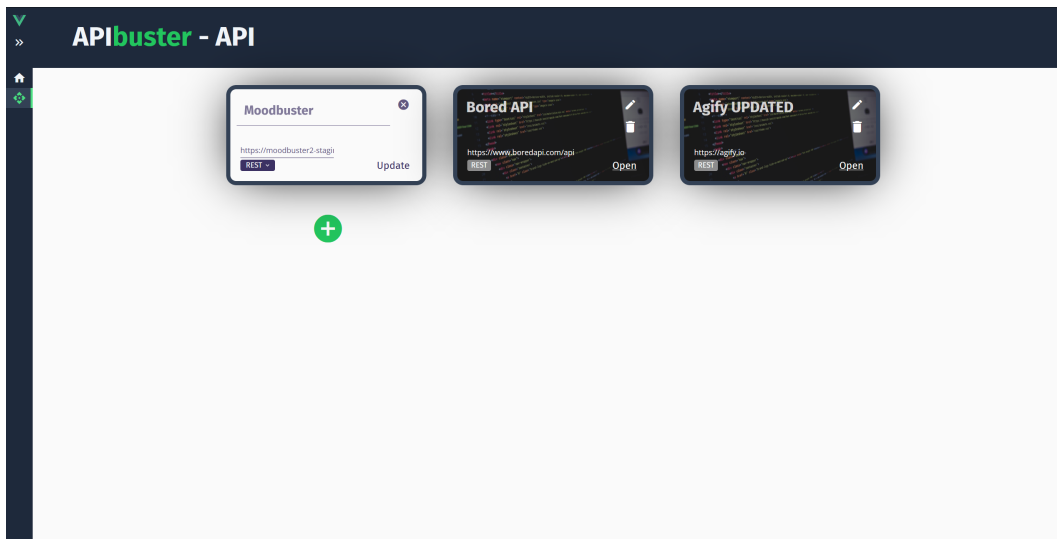


Figure D.4: APIbuster user dashboard: Updating an API

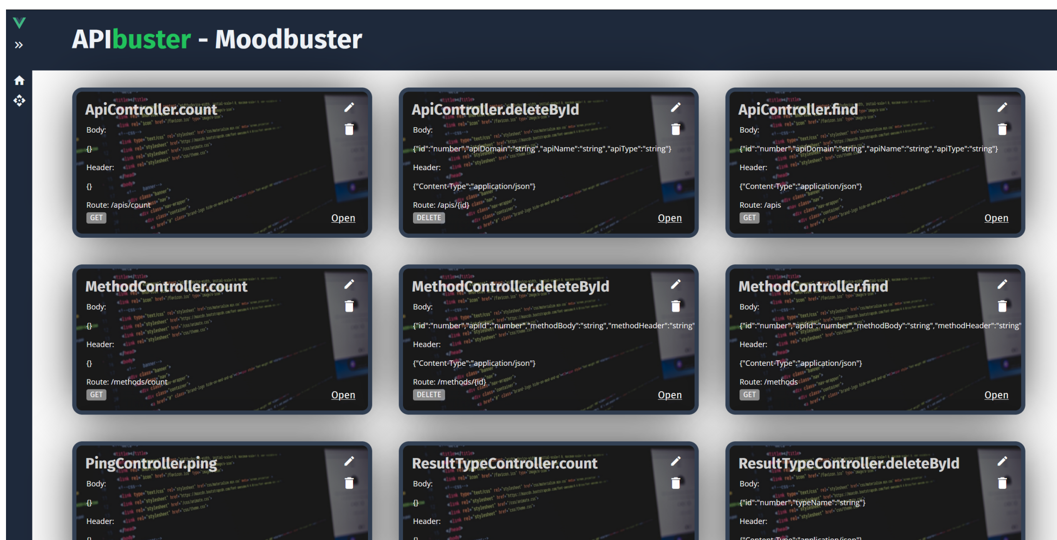


Figure D.5: APIbuster user dashboard: Methods page

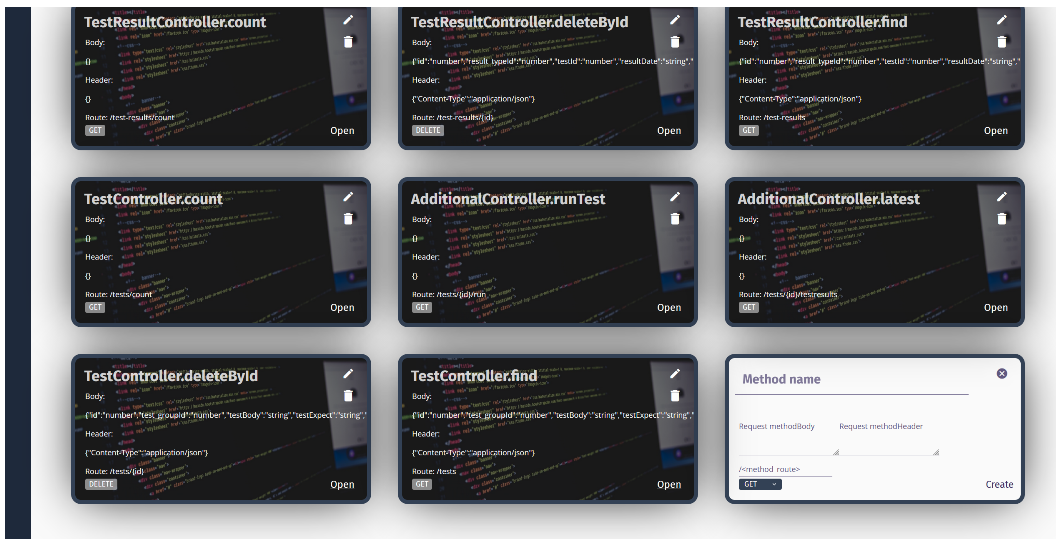


Figure D.6: APIbuster user dashboard: Adding a Method

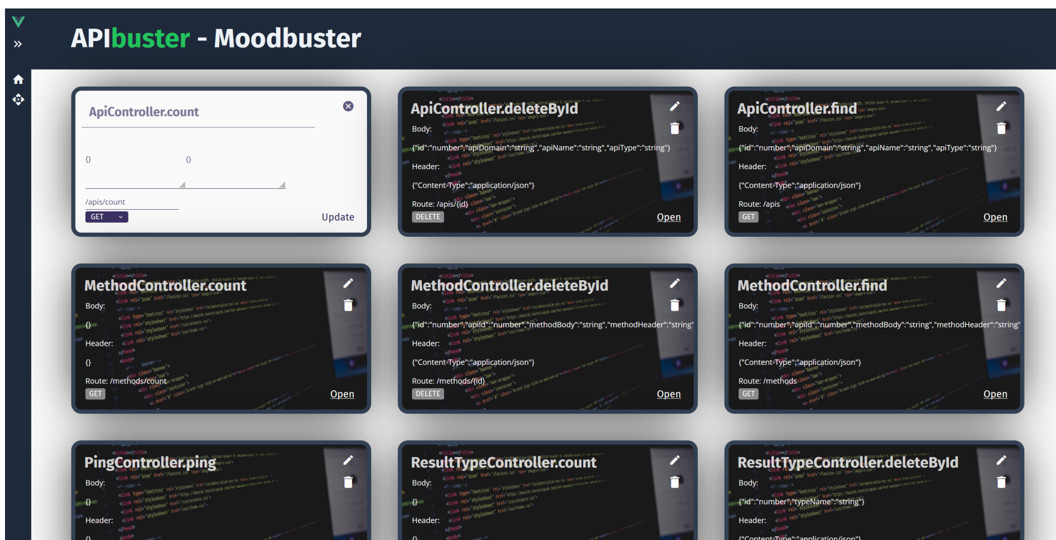


Figure D.7: APIbuster user dashboard: Updating a Method

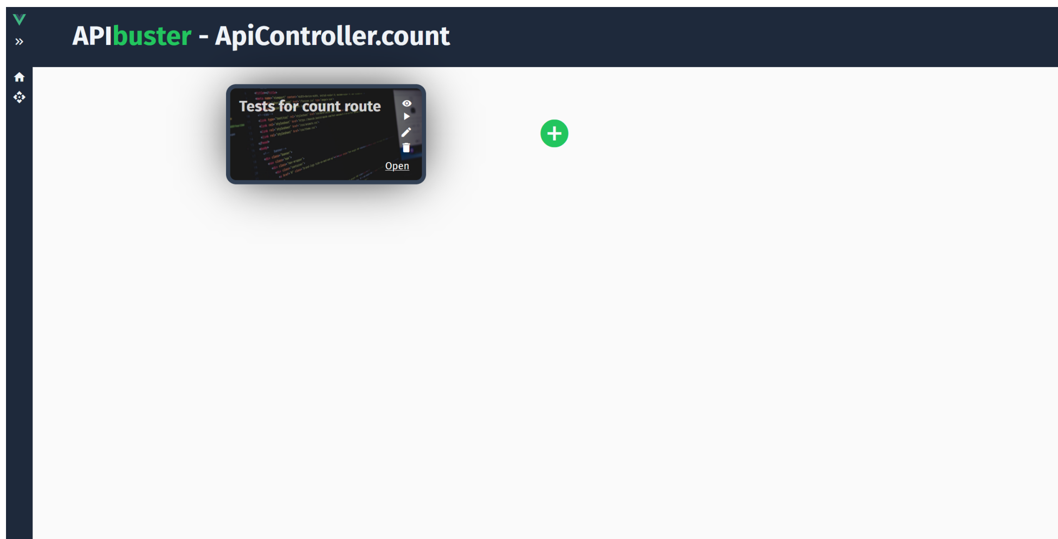


Figure D.8: APIbuster user dashboard: Test Groups page

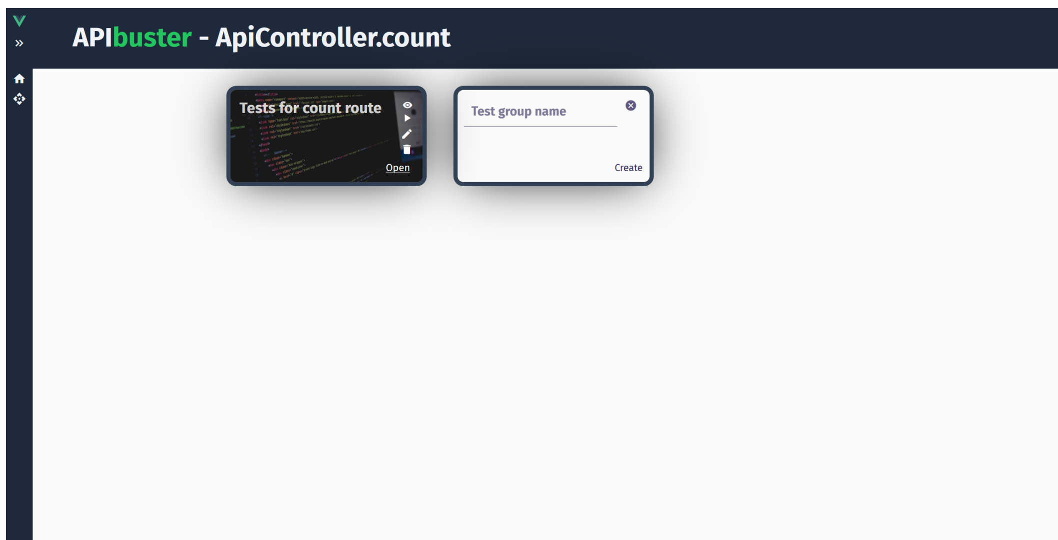


Figure D.9: APIbuster user dashboard: Adding a Test Group

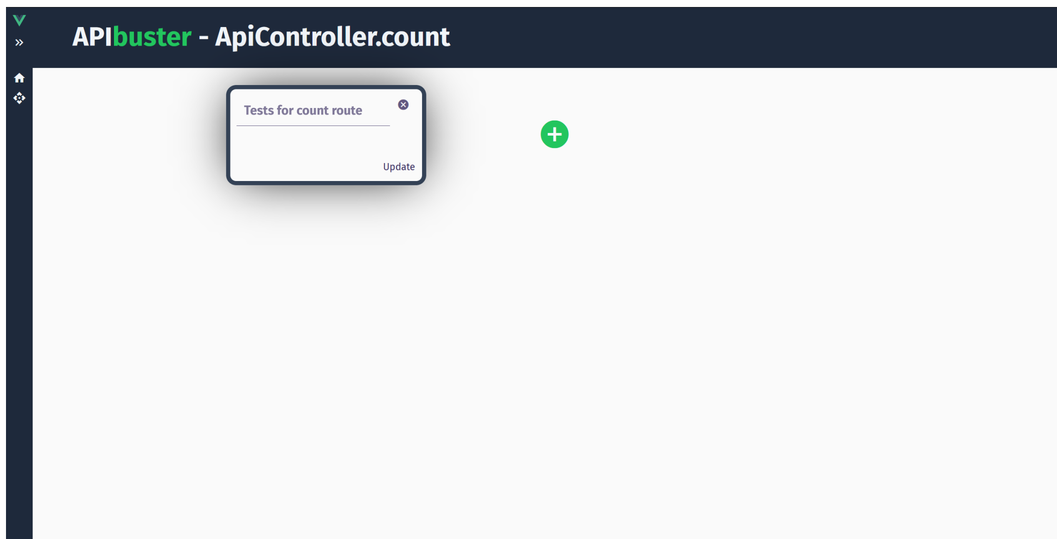


Figure D.10: APIbuster user dashboard: Updating a Test Group

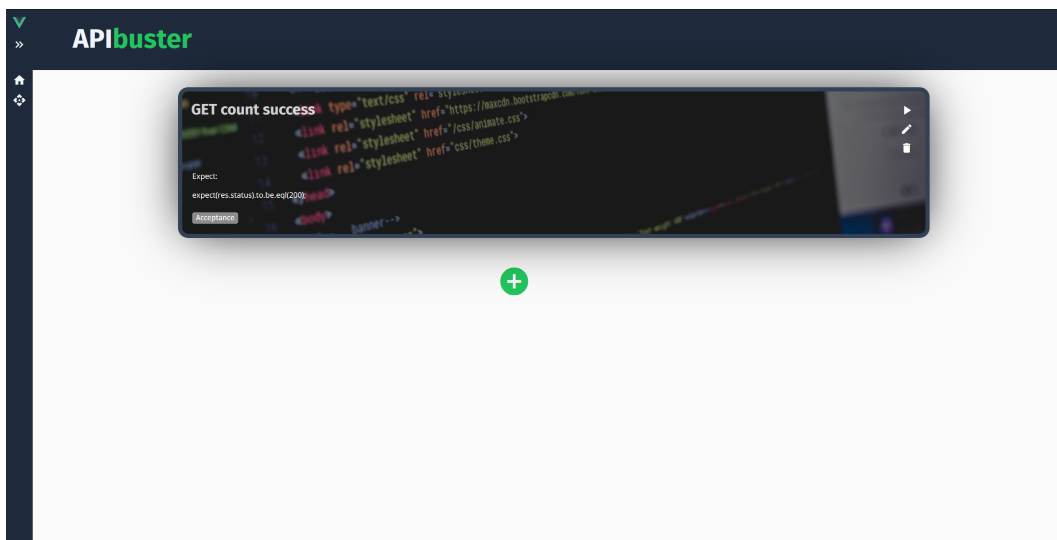


Figure D.11: APIbuster user dashboard: Tests page

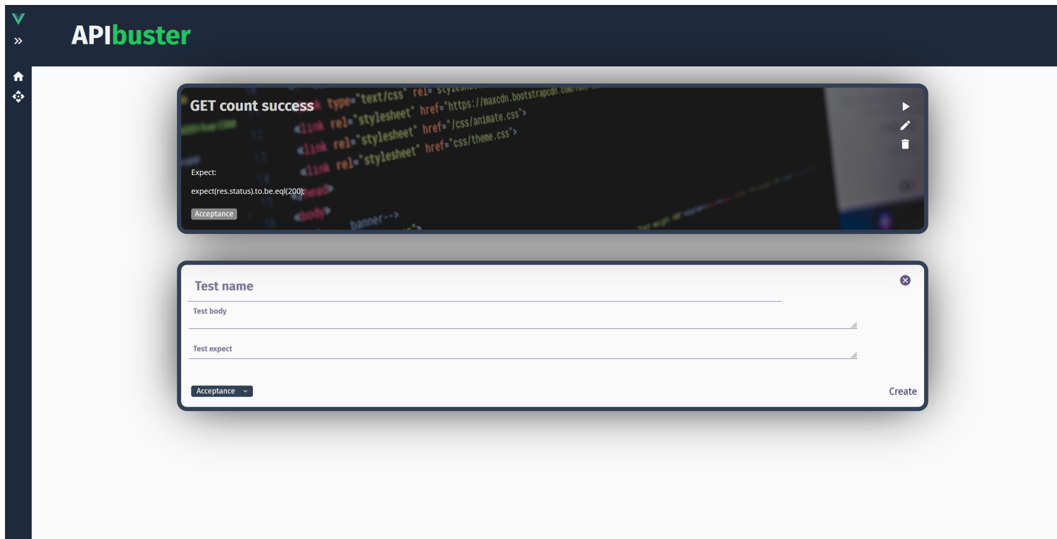


Figure D.12: APIbuster user dashboard: Adding a Test

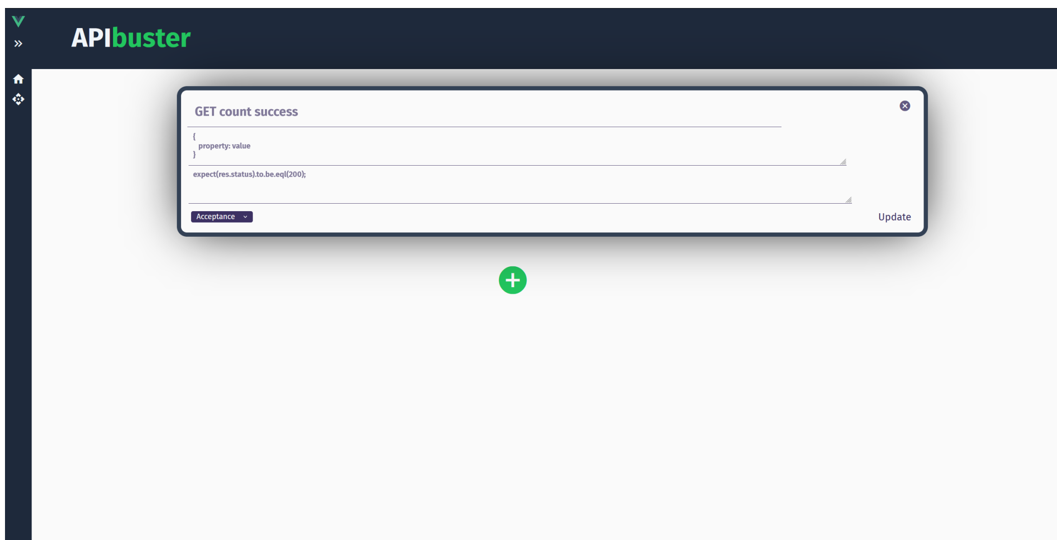


Figure D.13: APIbuster user dashboard: Updating a Test

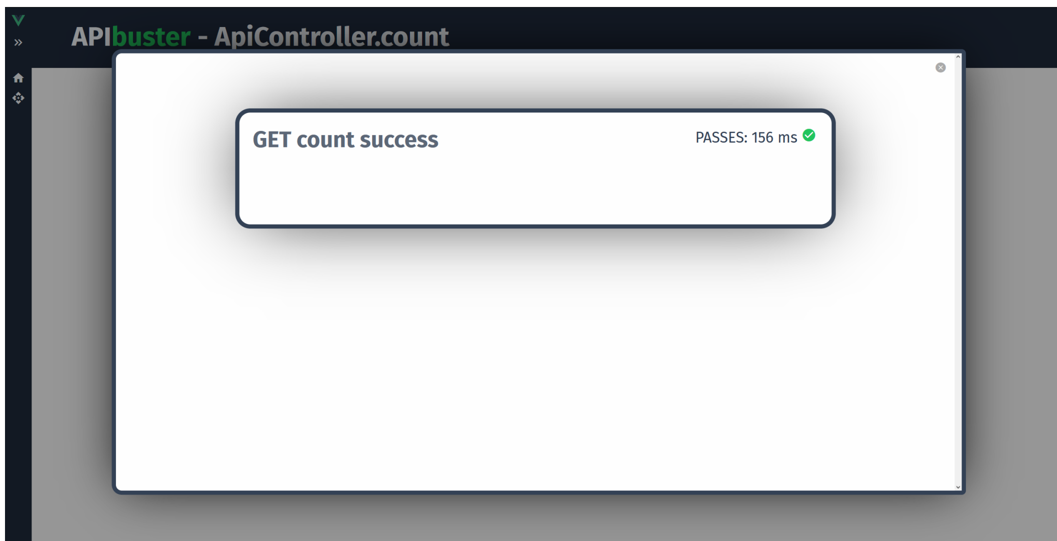


Figure D.14: APIbuster user dashboard: Results of the latest Test