Universidade do Minho
Escola de Engenharia
Departamento de Informática

André da Silva Gonçalves

Concepção e desenvolvimento de uma API REST com incorporação de mecanismos de segurança aplicacional

April 2022

Universidade do Minho
Escola de Engenharia
Departamento de Informática

André da Silva Gonçalves

Concepção e desenvolvimento de uma API REST com incorporação de mecanismos de segurança aplicacional

Dissertação de Mestrado
Mestrado Integrado em Engenharia Informática

Dissertação supervisionada por
Professor José Manuel Ferreira Machado
Ana Eduarda de Sá Silva

April 2022

AGRADECIMENTOS

Antes de iniciar a presente dissertação, estou grato ao Professor Doutor José Machado por toda a amizade demonstrada ao longo deste percurso, pela ajuda e aconselhamento em todos os momentos e por todas as palavras de incentivo que sempre teve para comigo.

Agradeço também à co-orientadora Eduarda Silva da UN1Qnx pela disponibilidade, apoio e dedicação com que me supervisionou ao longo dos últimos meses.

Os meus agradecimentos vão também para a empresa UN1Qnx por me ter fornecido as ferramentas e suporte para a realização deste trabalho e pela oportunidade que me foi proporcionada de colaborar num projeto num contexto empresarial.

A todos os meus amigos, aos que tive o prazer de conhecer ao longo deste percurso académico e aos que já faziam parte da minha vida agradeço-lhes por todos os bons momentos passados juntos.

Finalmente, aos meus pais, irmã e a toda a minha família, em especial ao meu padrinho, por me transmitirem valores que me fazem encarar o mundo com tolerância, curiosidade e liberdade e por estarem sempre ao meu lado e acreditarem em mim.

A todas estas pessoas um muito obrigado e espero conseguir retribuir todo o bem que me fizeram.

DECLARAÇÃO DE INTEGRIDADE

Declaro ter atuado com integridade na elaboração do presente trabalho académico e confirmo que não recorri à prática de plágio nem a qualquer forma de utilização indevida ou falsificação de informações ou resultados em nenhuma das etapas conducente à sua elaboração.

Mais declaro que conheço e que respeitei o Código de Conduta Ética da Universidade do Minho.

ABSTRACT

The constant technological evolution of the last decades makes more and more companies to focus on providing more resources to their customers, showing new perspectives for the development of solutions with high levels of performance, availability, scalability and flexibility.

One of the biggest contributions in this regard was the appearance of Application Programming Interfaces (APIs), increasingly crucial as integration, automation and efficiency become more important. With the abrupt emergence of APIs, API security has become a significant topic in the tech world. If an API does not have adequate security, it can be vulnerable to attacks that can compromise a company's data or system. Security should be considered from the beginning of any API development project and built into each step of the process to ensure that the API is adequately protected.

In this dissertation we intended to investigate the functioning of APIs, with special focus on the Representational State Transfer (REST) architecture and their security, allowing us to verify that, despite several techniques and tools for the creation of solid and robust REST APIs have already been studied in detail and applied to a wide variety of domains, REST services still need practical approaches specialized in the design and security of their APIs. It is proposed to fill this gap with the definition of a set of metrics capable of helping in the creation of a REST API with good design principles and absent of any vulnerabilities.

In the context of UN1Qnx as a company that develops authenticity solutions, an IT infrastructure capable of handling multiple customers and systems is essential for its business.

Bearing this need in mind, the opportunity arose to implement in practice the result of all the research carried out throughout the dissertation through the development of an Application Programming Interface (API) that follows the principles of architectural style based on REST in order to allow managing the data flow of the UN1Qnx system together with the definition of mechanisms to integrate the entire UN1Qnx service with third-party applications and services in order to automate procedures for creating and changing data.

KEYWORDS API, REST, RESTful, Web Services, Web Security

RESUMO

A constante evolução tecnológica das últimas décadas faz com que cada vez mais as empresas se foquem em fornecer mais recursos para os seus clientes, evidenciando novas perspetivas para o desenvolvimento de soluções com altos níveis de performance, disponibilidade, escalabilidade e flexibilidade.

Um dos maiores contributos neste sentido foi o aparecimento das Application Programming Interfaces (APIs), cada vez mais cruciais à medida que a integração, automatização e eficiência se tornam mais importantes. Com o surgimento abrupto das APIs, a segurança nas APIs tornou-se um tópico significativo no mundo da tecnologia. Se uma API não tiver a segurança adequada, ela pode ficar vulnerável a ataques que podem comprometer os dados ou o sistema de uma empresa. A segurança deve ser considerada desde o início de qualquer projeto de desenvolvimento de uma API e construída em cada etapa do processo para garantir que a mesma tenha a proteção adequada.

Nesta dissertação pretendeu-se investigar o funcionamento de APIs, com especial foco na arquitetura Representational State Transfer (REST) e na segurança das mesmas permitindo-nos verificar que, apesar de várias técnicas e ferramentas para a criação de APIs REST sólidas e robustas já tenham sido estudadas em detalhe e aplicadas a uma grande variedade de domínios, os serviços REST ainda necessitam de abordagens práticas especializadas no design e segurança das suas APIs. Propõe-se preencher esta vaziez com a definição de um conjunto de métricas capazes de auxiliar na criação de uma API REST com bons príncipios de design e ausente de quaisquer vulnerabilidades.

No contexto da UN1Qnx, empresa que desenvolve soluções de autenticidade e que permitiu o desenvolvimento desta dissertação em contexto empresarial, uma infraestrutura TI capaz de lidar com vários clientes e sistemas é essencial para o seu negócio.

Tendo esta necessidade em foco, surgiu a oportunidade de implememtentar na prática o resultado de toda a investigação efetuada ao longo da dissertação através do desenvolvimento de uma Interface de Programação de Aplicações (API) que siga os princípios do estilo arquitetural baseado em REST de forma a permitir gerir o fluxo de dados do sistema UN1Qnx a par com a definição de mecanismos para se integrar todo o serviço UN1Qnx com aplicações e serviços de terceiros no sentido de se automatizar procedimentos para criação e alteração de dados.

PALAVRAS-CHAVE API, REST, RESTful, Serviços Web, Segurança Web

CONTEÚDO

LISTA DE FIGURAS

LISTA DE TABELAS

LIST OF ABBREVIATIONS

AJAX Asynchronous JavaScript And XML. 47

API Application Programming Interface. c, d, v, 5, 6, 7, 9, 10, 11, 22, 23, 24, 26, 27, 28, 29, 30, 31, 32, 33, 36, 37, 38, 39, 40, 41, 42, 43, 46, 47, 48, 49, 51, 52, 53, 54, 55, 57, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69

CORS Cross-Origin Resource Sharing. 31, 46, 66

CRUD Create, Read, Update and Delete. vii, 18, 19

DoS Denial-of-Service attack. 29, 33, 66

DSR Design Science Research. v, 7, 8

DTO Data Transfer Object. 49, 50

gRPC Google Remote Procedure Call. 26

HATEOAS Hypermedia as the Engine of Application State. 22, 24

HTML HyperText Markup Language. 14, 16

HTTP Hypertext Transfer Protocol. v, vii, 9, 11, 12, 14, 15, 16, 17, 18, 19, 20, 21, 23, 24, 26, 27, 30, 31, 32, 37, 40, 41, 42, 43, 46, 47, 48, 49, 50, 62, 64, 65, 66

HTTPS Hypertext Transfer Protocol Secure. 27, 66

IdP Identity provider. 66

IoC Inversion of Control. 44, 46

IT Information Technology. c, 36, 69

JSON JavaScript Object Notation. 16, 26, 28, 48, 57

JWT JSON Web Token. 28, 46

MIME Multipurpose Internet Mail Extensions. 14, 31, 65

MVC Model-View-Controller. 33

OWASP Open Web Application Security Project. 28, 32, 33, 61

REST Representational State Transfer. c, d, v, 5, 7, 9, 10, 11, 14, 18, 20, 21, 22, 24, 25, 26, 27, 28, 30, 31, 32, 33, 37, 38, 39, 40, 43, 62, 63, 65, 66, 67, 68, 69

RPC Remote Procedure Call. v, 11, 12, 13, 26, 37

RSA Rivest-Shamir-Adleman. 56

SaaS Software as a Service. 33, 52

SDK Software Development Kit. 43, 52, 64
SHA Secure hash algorithm. 56
SMTP Simple Mail Transfer Protocol. 12
SOAP Simple Object Access Protocol. v, 12, 13, 23, 28, 37
SQL Structured Query Language. 29, 36
SUT System Under Test. 59

TCP Transmission Control Protocol. 14
TI Tecnologia da Informação. d, 7
TLS Transport Layer Security. 27

UDDI Universal Description, Discovery, and Integration. 13
UI User Interface. 64
URI Uniform Resource Identifier. 15, 16, 20, 21, 22, 23, 24, 40, 62
URL Uniform Resource Locator. 22, 24, 31, 47, 48, 63, 66

W3C World Wide Web Consortium. 12, 13, 31, 66
WSDL Web Services Description Language. 13

XML Extensible Markup Language. 12, 13, 16, 28
XSS Cross-site scripting. 31, 47, 65

ZAP Zed Attack Proxy. v, 61

1

INTRODUÇÃO

O presente documento descreve a pesquisa e análise para o desenvolvimento de uma Application Programming Interface (API) segura que siga os princípios do estilo arquitetural Representational State Transfer (REST). Este capítulo introdutório está dividido em quatro secções. Uma breve contextualização deste projeto é apresentada, incluindo referências aos principais tópicos de discussão, bem como a principal motivação que levou à sua realização (Secção 1.1). O subcapítulo seguinte inclui a escolha da metodologia de investigação adotada (Secção 1.2). Depois disso, os principais objetivos propostos para o desenvolvimento desta dissertação de mestrado são retratados (Secção 1.3). Por fim, finalizar-se-á com a apresentação da estrutura do documento para simplificar a sua leitura (Secção 1.4).

1.1 CONTEXTO & MOTIVAÇÃO

No mundo atual, com a evolução da tecnologia e o seu acesso cada vez mais facilitado, a sociedade habituou-se a recorrer às aplicações informáticas para as mais variadas áreas de trabalho, passando assim a ser cada vez mais relevante a evolução da forma como estas ferramentas estariam disponíveis para os utilizadores. Como tal, é dever das empresas acompanhar o avanço tecnológico que se faz sentir de dia para dia e aproveitá-lo da melhor maneira de modo a fazer com que o seu negócio atinja os clientes de forma mais fácil, rápida e global.

Um dos avanços tecnológicos com maior destaque nas últimas duas décadas é, sem dúvida, as APIs. O termo API é usado em diferentes contextos no campo da engenharia de software. Na generalidade, refere-se à interface de um elemento de software que pode ser chamado ou executado. As APIs fornecem uma interface para dados armazenados que se ajustam às necessidades de uma aplicação, permitindo que o software comunique entre si. Ou seja, as APIs funcionam como elo entre as diferentes aplicações ao serviço do utilizador. Possibilitam a integração entre distintos sistemas e tornam as ações dos utilizadores mais rápidas. A API funciona sempre em segundo plano, por isso o utilizador comum não chega sequer a ver a API em funcionamento tirando apenas partido da sua utilidade quando utiliza um site ou uma *app*.

De uma forma bastante simples, uma API permite a comunicação entre quaisquer duas aplicações. Por exemplo, uma aplicação X (utilizador) envia um pedido à aplicação Y (plataforma da empresa) e, a seguir, Y retorna uma resposta com as informações ou resultados de acordo com a ação solicitada no pedido de X.

Uma API no contexto desta tese é, portanto, um componente que oferece certas funcionalidades e dados a outros componentes por meio de uma interface definida que permite a execução de várias tarefas.

Atualmente, todo o software desenvolvido ou usa uma API ou é uma API. Estas tornaram-se um elemento essencial na economia digital atual. Prova disso é a taxa de crescimento das APIs que aumenta a um ritmo explosivo como podemos ver pela figura 1.

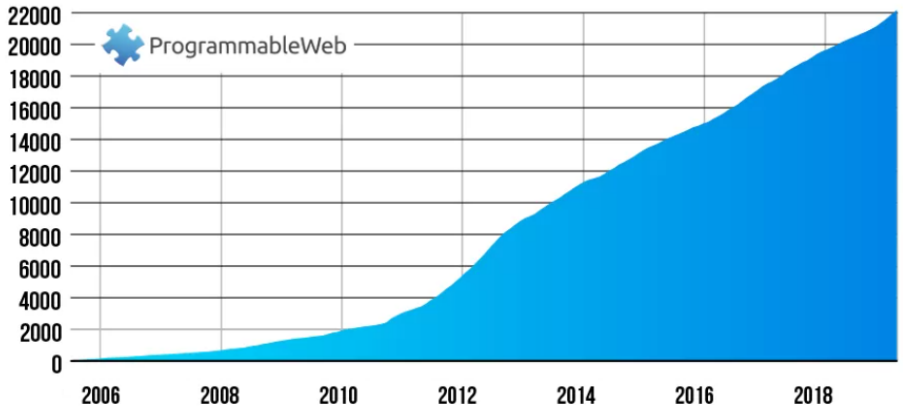


Figura 1: Crescimento da disponibilização de APIs (Fonte: ProgrammableWeb.com)

Para este crescimento, foi essencial o papel desempenhado por alguns gigantes tecnológicos como a eBay, Amazon, Facebook, entre outros, que através da disponibilização pública destas ferramentas inovadoras impulsionaram novos progressos tecnológicos.

Com a contínua transição mundial para serviços móveis e serviços *cloud* e o crescente poder, utilidade e omnipresença da Internet como um todo, o que antes parecia limitado ao setor da tecnologia é agora usado em quase todos os setores de negócios. Resumindo, quanto mais a internet se expande, mais o mundo das APIs cresce.

Com a atual ascensão das APIs, parece que os ciberataques estão a mudar o foco dos seus alvos tradicionais e a concentrar as suas energias nas APIs. Apesar de facilitar a troca de informações entre aplicações, as APIs enfrentam adversidades em oferecer um ambiente protegido e totalmente fiável.

De acordo com um relatório da Imperva, uma empresa de software e serviços de segurança cibernética, as violações de APIs são cada vez mais a cada ano. De acordo com o relatório, o número de novas vulnerabilidades de APIs aumentou cerca de 154% de 2015 a 2018:

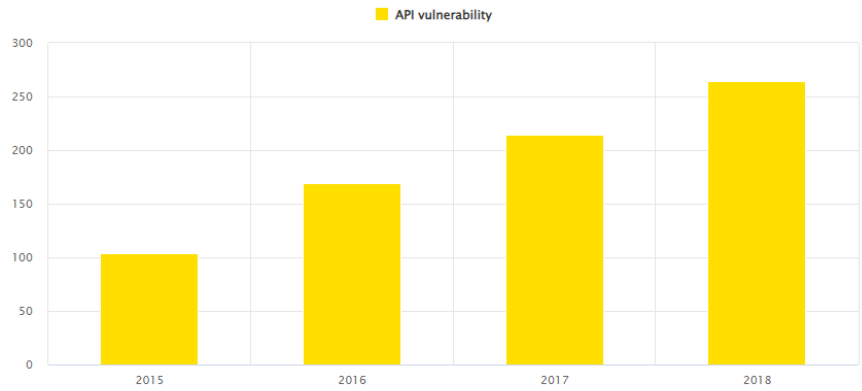


Figura 2: Número de vulnerabilidades de APIs de 2015 a 2018 (Fonte: Imperva)

Portanto, com estas crescentes preocupações com a segurança das APIs, as equipas de segurança das empresas devem abordar as APIs com o mesmo nível de seriedade oferecido a outras aplicações essenciais aos negócios. As medidas de segurança nas APIs devem ser consideradas um aspecto essencial do processo de desenvolvimento, e não apenas uma reflexão tardia.

É mediante esta perspetiva que a UN1Qnx também está a caminhar, fazendo da mesma o caso de estudo perfeito para o trabalho de investigação desenvolvido nesta dissertação através da continuação da implementação de uma API que siga os princípios do estilo arquitetural REST privilegiando a segurança de todos os processos envolvidos na mesma.

1.2 METODOLOGIA DE INVESTIGAÇÃO

Um projeto de pesquisa requer a adoção de metodologias para definir as suas fases, processos e métodos. Como tal, todo o estudo de Tecnologia da Informação (TI) deve pesquisar e analisar cuidadosamente as múltiplas metodologias e tecnologias disponíveis e viáveis para a concepção de uma solução. A decisão final deve ser feita tendo em conta quais as vantagens e desvantagens que um certo método oferece e, com base nisso, determinar o veredicto final.

Daqui em diante, o desenvolvimento desta dissertação seguirá a metodologia Design Science Research (DSR), frequentemente usada na construção e avaliação de soluções de TI confiáveis e rigorosas. Adicionalmente, em todas as fases durante a criação deste sistema, serão adotadas e implementadas múltiplas metodologias, tecnologias e ferramentas adequadas à definição e implementação de cada componente da solução final.

DSR é um método empírico (baseado em evidência) para a criação sistemática de soluções inovadoras (Hevner et al., 2004). Neste sentido, DSR prescreve um conjunto rigoroso de etapas para levar a avanços no Estado da Arte. Assim, o principal conceito em DSR é o artefacto, isto é, soluções para problemas materializadas em modelos, métodos, processos e ferramentas. DSR segue a premissa de que não é possível criar um artefacto

inovador para solucionar um problema em aberto sem que isso necessariamente passe pelo avanço do Estado da Arte no domínio em questão. Como o próprio nome diz, trata-se de uma "ciência de projeto", isto é, esta metodologia utiliza uma abordagem iterativa, permitindo ter a liberdade de adaptar a estrutura e avaliação do artefacto desenvolvido até que uma solução ideal seja encontrada (Horita et al., 2018).

A metodologia Design Science Research proposta por Peffers et al. está representada na figura 3.

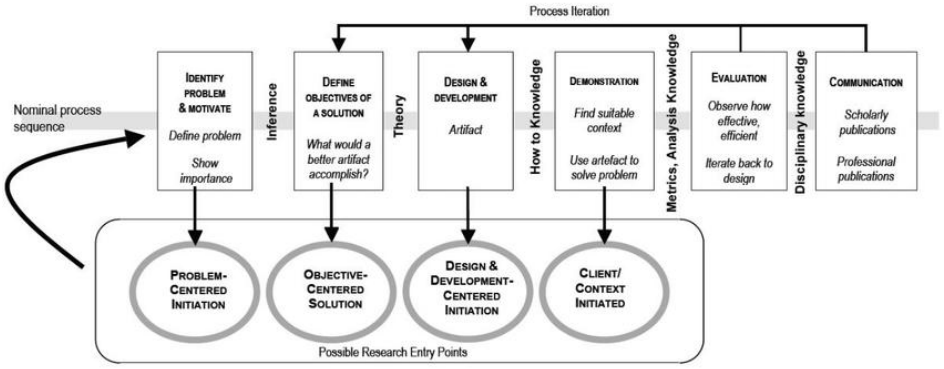


Figura 3: Metodologia Design Science Research (Fonte: Peffers et al. (2007))

Identificação do problema e motivação

Nesta fase, define-se o problema específico de investigação e justifica-se o valor da solução encontrada. A definição do problema ajuda na elaboração dos artefactos e estes, por sua vez, permitem facilitar a atingir a solução. De forma a justificar a solução, os investigadores são motivados a adquirir conhecimento sobre o estado da arte na área do problema e da importância ou relevância da solução.

Definição dos objetivos para a solução

Nesta fase, os objetivos da solução devem ser inferidos a partir da definição do problema. Estes objetivos podem ser quantitativos ou qualitativos. Se forem quantitativos, a solução proposta deve ser melhor que as já existentes. Se forem qualitativos, a solução deve descrever a forma como o novo artefacto suporta a solução do problema.

Desenho e desenvolvimento

Nesta fase, o artefacto da solução é criado desde a definição das suas funcionalidades e arquitectura, até ao seu desenho e desenvolvimento.

Demonstração

Nesta fase, deve ser feita a demonstração da utilização do artefacto para resolver uma, ou mais, questões do problema. Esta demonstração pode ser alcançada através de experiências, simulações, casos de estudo, entre outros.

Avaliação

Nesta fase verifica-se, observando ou medindo, se realmente o artefacto suporta a solução para o problema. Para isso, esta fase requer o conhecimento de métricas relevantes e técnicas de análise. No final desta atividade, dependendo dos resultados da avaliação, os investigadores podem decidir se devem iterar de volta para a 3ª etapa, na tentativa de melhorar a eficácia do artefacto ou de prosseguir para a etapa seguinte.

Comunicação e difusão do resultado

Nesta fase, é necessário comunicar o problema e a sua relevância, o artefacto, a sua utilidade, novidade, rigor e eficácia a outros investigadores e a outras audiências relevantes, como profissionais da área em exercício. Esta divulgação deve ser feita, de preferência, em revistas ou conferências da área de pesquisa.

1.3 OBJETIVOS

Sem uma definição de segurança de API adequada, uma empresa corre o risco de se tornar um alvo fácil para atacantes, dada a capacidade da API de fornecer informações a *developers* externos. Aplicar medidas corretas de segurança nas APIs garantirá que os dados digitais expostos, que são o essencial dos próprios negócios, sejam protegidos contra usos não autorizados.

A UN1Qnx, enquanto empresa que permitirá aplicar o conhecimento adquirido nesta dissertação num ambiente real, é uma empresa especializada no desenvolvimento e comercialização de sistemas físicos, eletrónicos e cibernéticos para soluções de validação e autenticação de produtos. Neste momento, a UN1Qnx encontra-se em grande crescimento e tem como objetivo tornar-se uma referência na área, sendo que os seus principais mercados são *fashion*, vinhos e bebidas espirituosas.

Face a isto, uma das necessidades encontradas por parte da UN1Qnx é a criação de uma ferramenta que permita aos seus utilizadores a validação e autenticação dos seus produtos de uma forma simples e eficiente tirando partido das múltiplas tecnologias atuais.

Desta forma, com esta dissertação pretendeu-se estudar a necessidade da existência de APIs REST seguras e de disponibilizar um serviço web baseado em Transferência de Estado Representativo (REST) nesse sentido. Este serviço torna acessível uma Interface de Programação de Aplicações (API), por forma a suportar o desenvolvimento de novas soluções na empresa, bastando para isso que a linguagem de programação usada suporte pedidos pelo protocolo Hypertext Transfer Protocol (HTTP).

Portanto, no contexto deste projeto de dissertação, surgiu a seguinte *Research Question* (questão de investigação):

- **Questão de investigação:** Como desenvolver uma API REST segura?

  Para responder a esta questão, destacaram-se as seguintes tarefas:

  - Revisão da literatura acerca do desenvolvimento de uma API;
  - Análise dos diferentes estilos de APIs e justificação da escolha da arquitetura REST;
  - Estudo de metodologias e procedimentos que devem ser adotados no desenvolvimento de uma API REST;
  - Estudo dos problemas relacionados com questões de segurança nas APIs;
  - Investigação da utilidade das APIs REST e de soluções existentes no mercado de forma a perceber como os problemas identificados podem ser ultrapassados;
  - Análise dos processos envolvidos no serviço UN1Qnx, de todos os seus componentes, das suas relações e do método de desenvolvimento utilizado, de forma a saber quais as limitações que este trabalho apresenta;
  - Levantamento de requisitos do projeto e do desenvolvimento da plataforma;
  - Análise das ferramentas e tecnologias atuais tanto para o desenvolvimento como para o teste da solução;
  - Implementação de todos os componentes e validação no sistema UN1Qnx.

1.4 ESTRUTURA DO DOCUMENTO

Esta dissertação está estruturada em cinco capítulos diferentes:

- Introdução: Neste capítulo é apresentado o contexto deste projeto de dissertação, os motivos para a sua existência, o que se pretende alcançar bem como a metodologia de investigação adotada.
- Estado da Arte: Neste capítulo é feita a revisão de assuntos relacionados com o tema da dissertação.
- Projeto: Neste capítulo é apresentada a solução proposta e os problemas resolvidos e desafios superados para atingir com sucesso os objetivos previamente definidos .
- Resultados e Discussão: Neste capítulo é feita a discussão sobre o trabalho desenvolvido e a adequação da solução apresentada.
- Conclusões: Neste capítulo final é feita uma revisão final do que foi produzido e aborda-se o trabalho futuro a ser realizado.

2

ESTADO DA ARTE

Nesta secção são expostos os resultados dos artigos analisados e da pesquisa efetuada sobre as técnicas do Estado da Arte, a fim de melhor compreender como são utilizadas para superar os desafios propostos. Será primeiramente abordada uma perspetiva histórica do surgimento e evolução da utilização de *web services* bem como uma menção ao HTTP, devido à sua relevância para a implementação de uma API REST. De seguida, é explicada a arquitetura geral das APIs REST juntamente com outros tipos de APIs existentes. Por fim, são também focadas questões de segurança relacionadas com as mesmas.

2.1 REMOTE PROCEDURE CALL

Os web services evoluíram originalmente através do Remote Procedure Call (RPC) que é um protocolo vanguardeiro na comunicação entre aplicações e que permite a uma aplicação invocar uma função remota como se de uma função local se tratasse, precisando apenas de conhecer em que endereço essa função está disponível, e qual a sua assinatura (Birrell and Nelson, 1984).

A sequência de eventos numa *remote procedure call* é a seguinte:

- O cliente chama o *stub* do cliente. A chamada é uma chamada de procedimento local com parâmetros colocados na *stack* de maneira normal;
- O *stub* do cliente empacota os parâmetros do procedimento para uma mensagem e efetua uma *system call* para enviar a mensagem. Este processo do empacotamento dos parâmetros é chamado de *marshalling*;
- O sistema operativo local do cliente envia a mensagem da máquina do cliente para a máquina do servidor remota;
- O sistema operacional do servidor passa os pacotes de entrada para o *stub* do servidor;
- O *stub* do servidor desempacota os parâmetros da mensagem (*unmarshalling*);
- O procedimento do servidor é então invocado pelo *stub* do servidor para executar o pedido;
- O resultado dessa invocação é depois enviado para o cliente usando um mecanismo semelhante.

Um dado importante de referir é que o cliente é bloqueado enquanto o servidor processa a chamada e só retoma a execução após o servidor terminar.

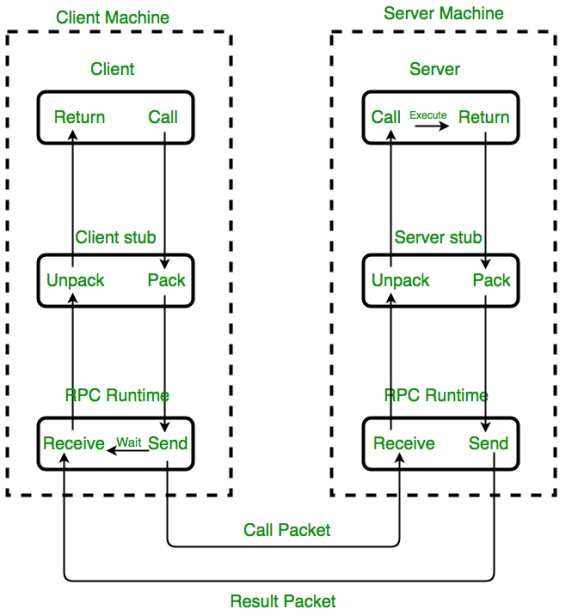


Figura 4: Implementação do mecanismo RPC (Fonte: GeeksforGeeks.org)

A maior vantagem do RPC é permitir o uso de linguagens diferentes, pois não é focado apenas numa linguagem de programação específica e torna o desenvolvimento de programas com sistemas distribuídos mais simples, podendo ser utilizado tanto em ambientes locais, como em ambientes distribuídos.

Por outro lado, também possui algumas desvantagens tais como ser vulnerável a falhas quer seja ao nível da rede ou ao nível do funcionamento do servidor e diminuição da eficiência devido ao processo de (*un*)*marshalling* e à camada de comunicação. Adicionalmente, como pode ser implementado de diferentes formas não existe uma especificação standard deste conceito.

2.2 SOAP

Para fazer face às dificuldades referidas anteriormente e face ao crescente êxito da Web, os princípios da arquitetura subjacente à Web começaram a ser estudados para conceber outros tipos de sistemas distribuídos. Simple Object Access Protocol (SOAP) foi um dos modelos que ganhou mais destaque e durante algum tempo, os serviços Web eram sinónimo de SOAP.

SOAP é uma recomendação do World Wide Web Consortium (W3C) que utiliza as potencialidades de protocolos já existentes, como o Hypertext Transfer Protocol (HTTP) ou o Simple Mail Transfer Protocol (SMTP) recorrendo ao Extensible Markup Language (XML) como formato de empacotamento.

Uma mensagem SOAP é um documento XML comum contendo um elemento chamado Envelope que identifica o documento XML como uma mensagem SOAP, um elemento Header (opcional) que contém informações sobre o cabeçalho do documento, e um elemento Body que é o conteúdo da mensagem, sendo que este varia conforme as necessidades da implementação e tipicamente inclui os métodos e respetivos argumentos a invocar. Dentro do corpo pode ainda existir o elemento Fault que contém erros e informações de status (W3C, a).

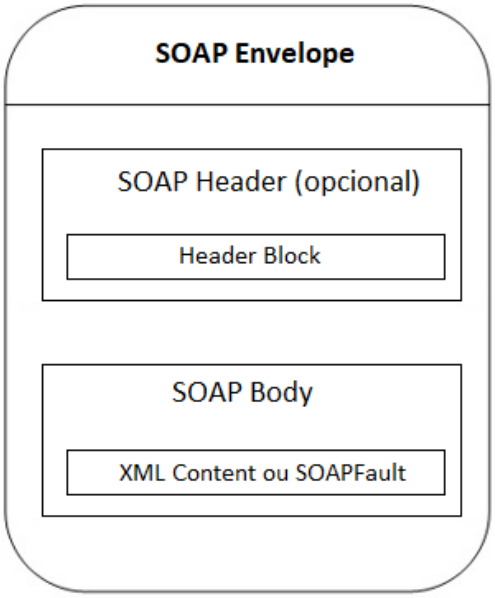


Figura 5: Estrutura de um envelope SOAP

Além do objetivo principal do protocolo de tornar possível o encapsulamento e o envio de pedidos RPC, o SOAP disponibiliza também mecanismos de extensibilidade designados como ”WS-*” (WS-), permitindo acrescentar funcionalidades aos serviços ao nível dos metadados, da segurança, das mensagens e das transações.

Uma adição relevante aos serviços web baseados em SOAP foi também a oportunidade de os disponibilizar publicamente. Para tal, e também definido pela W3C, existe um formato baseado em XML denominado Web Services Description Language (WSDL) que além de descrever o serviço, especifica como acedê-lo e quais as operações ou métodos disponíveis (W3C, b).

De seguida, uma outra norma standard que também se destaca nos serviços web é o Universal Description, Discovery, and Integration (UDDI) que permite criar metainformação associada a um serviço web. O seu funcionamento é bastante simples: os registos UDDI são armazenados em repositórios sendo que estes possuem uma interface de procura para permitir a uma aplicação cliente pesquisar e localizar um serviço.

Apesar de todos estes processos serem bastante notáveis, rapidamente se constatou que o SOAP não se adaptava da melhor forma aos conceitos arquiteturais da Web, nomeadamente, no que concerne à complexi-

dade e tamanho das mensagens (Suda, 2003). Em contrapartida, logo após o aparecimento do REST, este consagrou-se como um melhor seguidor dos princípios arquiteturais da Web.

Na secção 2.4 irá ser introduzida a arquitetura REST, no entanto para a sua melhor compreensão torna-se fundamental perceber um conceito básico que está na base desta arquitetura, nomeadamente o protocolo de comunicação HTTP.

2.3 HTTP

O Hypertext Transfer Protocol é um protocolo de comunicação da camada de aplicação para sistemas distribuídos que permite aos utilizadores comunicar dados na rede mundial de computadores.

O HTTP foi inventado em conjunto com o HyperText Markup Language (HTML) para criar o primeiro navegador interativo baseado em texto: a World Wide Web original. Nos tempos atuais, o protocolo continua a evoluir e a afirmar-se como um dos principais meios de uso da Internet.

Como é um protocolo de solicitação-resposta, o HTTP oferece aos utilizadores uma maneira de interagir com recursos da web, como arquivos HTML, transmitindo mensagens de hipertexto entre clientes e servidores, sendo que, geralmente, os clientes HTTP usam conexões TCP (Transmission Control Protocol) para comunicarem com os servidores (David Gourley, 2002).

Existem três carateristicas básicas que tornam o HTTP um protocolo simples, mas bastante poderoso R. Fielding (a,b):

- HTTP não mantém a conexão, ou seja, o cliente HTTP inicia uma requisição HTTP e, depois da solicitação ser feita, espera pela resposta. O servidor processa a solicitação e envia uma resposta de retorno, após isso, o cliente desconecta-se. Deste modo, o cliente e o servidor têm as informações um do outro apenas durante o pedido e resposta atuais. De cada vez que o cliente efetua um novo pedido, inicia-se uma nova conexão diferente;

- HTTP é independente de *media*, ou seja, qualquer tipo de dados pode ser enviado por HTTP desde que o cliente e o servidor saibam como lidar com o conteúdo dos mesmos. Para tal, é necessário que tanto o cliente como o servidor especifiquem o tipo de conteúdo usando o MIME-type apropriado;

- HTTP é um protocolo sem estado: como foi supracitado, HTTP não mantém a conexão e isso é um resultado direto de o protocolo ser *stateless*. Cada requisição é como uma transação independente que não está relacionada com qualquer requisição anterior, de forma a que a comunicação consista de pares de requisição e resposta independentes. Devido a esta natureza do protocolo, nem o cliente nem o navegador podem reter informações entre os diferentes pedidos nas páginas da web.

Mensagens

Os dados são trocados entre o servidor e cliente por meio de mensagens HTTP. Há dois tipos de mensagens: requisições (*requests*) enviadas pelo cliente para disparar uma ação no servidor, e respostas (*responses*), enviadas pelo servidor.

Um pedido HTTP é composto pelos seguintes elementos:

- Um método HTTP, que define qual a operação que o cliente quer fazer;
- Um Uniform Resource Identifier (URI) que identifica o recurso sobre o qual aplicar a requisição;
- A versão do protocolo HTTP;
- Cabeçalhos opcionais que contém informações adicionais para os servidores;
- Finalmente o corpo da mensagem que pode nem existir porque, normalmente, o propósito de um pedido HTTP é requisitar conteúdo.

Uma resposta HTTP consiste maioritariamente nos mesmos elementos:

- A versão do protocolo HTTP;
- Um código de status, indicando se a requisição foi bem sucedida, ou não;
- Uma mensagem de status, uma pequena descrição informal sobre o código de status;
- Cabeçalhos HTTP, como aqueles das requisições;
- Um corpo com os dados do recurso requisitado.

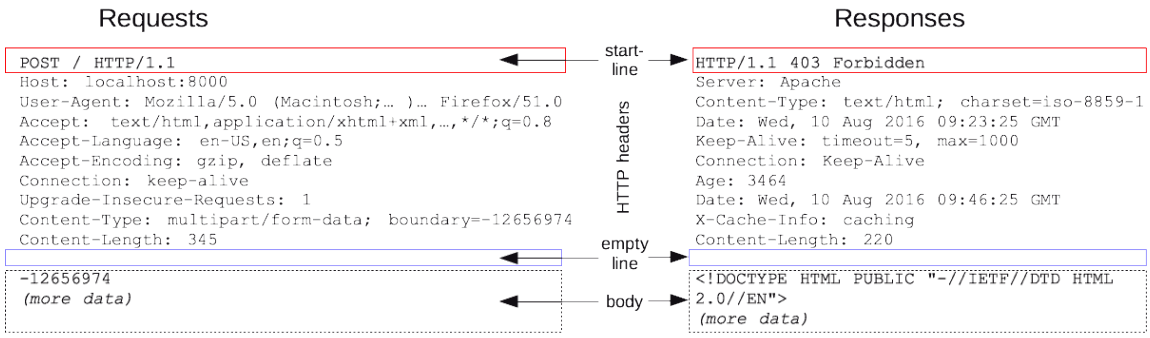


Figura 6: Estrutura de mensagens HTTP (Fonte: Developer.Mozilla.org)

Recursos

No decorrer deste documento o termo “recurso web” é referido com alguma frequência, no entanto é necessário esclarecer de forma correta o seu significado. O alvo de uma requisição HTTP é chamada de *resource*, ou recurso em português. O HTTP não restringe a natureza dos recursos, podendo assim ser dos mais variados tipos. O tipo de recursos mais simples são os ficheiros estáticos, tais como: ficheiros de texto, ficheiros HTML, imagens e vídeos. Contudo, os recursos podem não ser estáticos, podendo ser softwares que geram conteúdos consoante os parâmetros de entrada, imagens em direto de uma câmara ou o resultado de uma *query* aplicada a uma base de dados.

Para usar um recurso é necessário uma maneira de o identificar na rede e de o manipular. Para este propósito, o HTTP usa o identificador uniforme de recurso (URI).

Representações

Uma representação é informação que reflete o estado de um recurso, sendo que o formato desta informação pode ser comunicado através do protocolo HTTP. Esta é uma das carateristicas mais importantes do HTTP porque podemos ter várias representações para o mesmo recurso sendo que os formatos mais utilizados são o Extensible Markup Language (XML) ou JavaScript Object Notation (JSON). Esta especificaçao do formato é feita no cabeçalho, no campo do Content-Type.

Métodos

Qualquer requisição que seja uma mensagem HTTP, deve obrigatoriamente conter um dos métodos HTTP (ou verbos HTTP). Estes métodos dizem ao servidor que tipo de requisição ele deseja efetuar.

Método	Descrição
GET	O método GET solicita a representação de um recurso específico. Requisições utilizando o método GET devem retornar apenas dados.
HEAD	O método HEAD solicita uma resposta de forma idêntica ao método GET, porém sem conter o corpo da resposta.
POST	O método POST é utilizado para submeter uma entidade a um recurso específico, frequentemente causando uma mudança no estado do recurso ou efeitos colaterais no servidor.
PUT	O método PUT substitui as atuais representações do recurso de destino pelos dados da requisição.
PATCH	O método PATCH é utilizado para aplicar modificações parciais a um recurso.
DELETE	O método DELETE remove um recurso específico.
CONNECT	O método CONNECT estabelece um túnel para o servidor identificado pelo recurso de destino.
OPTIONS	O método OPTIONS é usado para descrever as opções de comunicação com o recurso alvo.
TRACE	Executa um teste com o intuito de verificar como é que a mensagem é manipulada ao longo do caminho até ao recurso de destino .

Tabela 1: Métodos HTTP

Cada um deles implementa uma semântica diferente, mas alguns recursos podem ser do mesmo tipo: seguro, idempotente ou *cacheable*.

Um método HTTP é considerado seguro se nunca modificar o recurso. GET, HEAD, OPTIONS e TRACE são exemplos de métodos seguros.

Todos os métodos seguros também são idempotentes, mas nem todos os métodos idempotentes são seguros.

Um método HTTP é considerado idempotente se independentemente do número de vezes que a requisição é repetida, o resultado retornado for o mesmo. Implementados corretamente, o GET, HEAD, PUT, DELETE OPTIONS e TRACE são métodos idempotentes.

Códigos de estado da resposta

Quando um servidor responde a um pedido de um cliente, inclui na mensagem de resposta um código composto por três dígitos. O propósito deste código é retribuir ao cliente o resultado da tentativa do servidor em

perceber e satisfazer o pedido. Os códigos podem ser divididos em 5 classes de status tal como mostra a tabela 2.

Classe	**Definição**	**Códigos de *status* mais utilizados**
1XX	Informativa	100 (*Continue*)
2XX	Sucesso	200 (*Ok*) 201 (*Created*) 204 (*No Content*)
3XX	Redireccionamento	304 (*Not Modified*)
4XX	Erro do Cliente	400 (*Bad Request*) 401 (*Unauthorized*) 403 (*Forbidden*) 404 (*Not Found*) 409 (*Conflict*)
5XX	Erro do Servidor	500 (*Internal Server Error*)

Tabela 2: Classes de status HTTP e respetivos códigos status HTTP mais utilizados

A Tabela 3 representa os métodos utilizados para as diferentes operações CRUD (*Create, Read, Update and Delete*) e os código de status HTTP mais utilizados na arquitetura REST.

Verbos HTTP	**Função CRUD**	**Status HTTP**
POST	*Create*	Sucesso: 201 (*Create*) Erro: 400 (*Bad Request*) 404 (*Not Found*) 409 (*Conflict*)
GET	*Read*	Sucesso: 200 (*Ok*) Erro: 400 (*Bad Request*)
PUT	*Update/Replace*	Sucesso: 200 (*Ok*) 204 (*No Content*) Erro: 404 (*Not Found*)
PATCH	*Update/Modify*	Sucesso: 200 (*Ok*) 204 (*No Content*) Erro: 404 (*Not Found*)
DELETE	*Delete*	Sucesso: 200 (*Ok*) 204 (*No Content*) Erro: 404 (*Not Found*)

Tabela 3: Métodos HTTP utilizados para as diferentes operações CRUD

2.4 REST

Representational State Transfer (REST) é um estilo de arquitetura e uma abordagem de comunicação utilizada no desenvolvimento de *web services*. A arquitetura REST foi primeiramente apresentada pelo Dr. Roy Thomas Fielding, investigador que esteve no desenvolvimento do protocolo HTTP e URI, na sua tese de doutoramento intitulada "Architectural Styles and Design of Network-based Software Architectures" (Fielding, 2000).

É um estilo arquitetónico, para aplicações de rede, muito popular devido à sua simplicidade e ao facto de se basear em sistemas e funcionalidades existentes a partir do protocolo de camada de aplicação Hypertext Transfer Protocol (HTTP), a fim de alcançar os seus objetivos, em vez de criar novos standards, *frameworks* e tecnologias.

Fielding descreveu que a arquitetura REST é derivada de um conjunto de restrições. Ao seguir estas restrições, é suposto que o resultado seja um sistema altamente escalável, com capacidade para crescer organicamente de uma maneira descentralizada. Seguidamente, são apresentadas as restrições que compõem o estilo arquitetural REST.

1- Null Syle

Existem duas perspetivas no processo do design das arquiteturas. Seguindo a primeira começamos a partir do nada e construímos a arquitetura a partir de componentes familiares aos utilizadores até atingir os objetivos desejados no sistema. Na segunda, começamos por pensar no produto final sem quaisquer condições e, apenas são implementadas mais restrições, caso seja necessário.

2- Cliente-servidor

É a restrição básica para uma aplicação REST. O objetivo desta divisão é a separação entre a interface do utilizador e o armazenamento de dados, o que permite a existência de portabilidade entre plataformas e melhora significativamente a escalabilidade do sistema. Neste modelo, o servidor espera pelas requisições do cliente, executa estes pedidos e devolve uma resposta.

3- Ausência de estado

Cada requisição do cliente ao servidor deve conter todas as informações necessárias para poder processar o pedido sem necessidade do uso de dados armazenados no servidor. O estado da sessão é, portanto, mantido inteiramente no lado do cliente. Deste modo, esta restrição *stateless* provoca repercussões bastantes positivas na visibilidade, fiabilidade e escalabilidade.

A visibilidade aumenta pois cada pedido apresenta toda a informação necessária, não sendo preciso verificar mais do que um pedido para se compreender a requisição.

A fiabilidade é melhorada porque facilita a resolução de falhas de comunicação, apenas sendo necessário para tal fazer o reenvio do pedido.

A escalabilidade é também beneficiada uma vez que, ao não ter que armazenar a sessão, o servidor pode libertar recursos mais rapidamente e sem dependências associadas.

4- Cache

De forma a evitar processamento desnecessário e aumentar substancialmente a performance, cada resposta deve assinalar um pedido como *cacheable* ou não e por quanto tempo as respostas podem ser armazenadas em cache no lado do cliente. Desta maneira, uma resposta a um pedido pode ser guardada temporariamente em cache para, posteriormente, ser aproveitada para requisições equivalentes.

Uma cache bem gerida elimina parcialmente ou completamente algumas interações cliente-servidor, melhorando ainda mais a escalabilidade e o desempenho. Contudo, por vezes existe a possibilidade do utilizador receber dados desatualizados.

5- Interface uniforme

A característica central que distingue o estilo arquitetónico REST de outros estilos baseados em rede é a sua ênfase numa interface uniforme entre os componentes do sistema. Dentro desta regra, existe um conjunto de diretrizes para fazer essa comunicação uniforme:

Identificação de recursos via URI

Cada *resource* deve ter um URI único e global para poder ser acedido. Cada um desses URIs direcionam para um controlador que retorna (via HTTP) a representação desse recurso;

Manipulação de recursos

Os componentes REST manipulam os recursos por meio do uso de representações que refletem o estado atual do recurso, ou seja, o mesmo recurso pode, por exemplo, ser manipulado para ser representado para clientes diferentes de maneiras diferentes. No paradigma REST, recursos e representações mantêm a mesma definição do contexto referido anteriormente no protocolo HTTP.

Mensagens auto-descritivas

As mensagens trocadas entre os componentes têm que ser auto-descritivas, ou seja, a mensagem deve conter todas as informações necessárias para a sua compreensão. O facto de a ligação ser *stateless* e de se utilizar tipos de *media* normalizados são aspetos que contribuem para as mensagens serem auto-descritivas.

HATEOAS

O acrónimo HATEOAS vem de Hypermedia as the Engine of Application State e consiste em devolver todas as informações necessárias na resposta para que o cliente possa navegar e ter acesso a todos os recursos (URIs) da aplicação a partir de um único ponto de partida. Desta forma, ao consumir uma API REST que segue o padrão HATEOAS, o cliente descobrirá URLs conforme navega entre os recursos, acedendo aos mesmos de forma circular, já que a API consegue ser autoexplicativa, guiando o consumidor através das suas requisições contribuindo assim para o desacoplamento entre cliente e servidor.

Esta restrição é, provavelmente, a mais importante para suportar o desacoplamento entre um cliente e um servidor, porque os recursos podem ser descobertos em tempo de execução e pode-se interagir com eles através da interface uniforme sem a necessidade de entendimento prévio entre as partes intervenientes.

6- Sistema em camadas

A fim de melhorar ainda mais o comportamento dos requisitos de escalabilidade da Internet, é possível ter um sistema organizado em camadas. Um dos princípios da aplicação desta restrição é que o cliente não consegue distinguir se está a chamar diretamente o servidor da aplicação ou uma camada intermediária, isso garante que o cliente apenas se preocupe com a comunicação com o intermediário e que este possa gerir as requisições pelos diferentes servidores. A principal desvantagem desta restrição é que adiciona *overhead* e latência no processamento dos dados, mas isso até pode ser benéfico numa network que suporte cache.

7- Code on demand

Esta funcionalidade permite que o cliente seja capaz de transferir e executar o código do seu lado, através de scripts de modo a que o sistema seja mais eficiente quando a quantidade de dados a processar é avultada. Esta restrição é, no entanto, opcional uma vez que apesar de aumentar a extensibilidade do sistema, a visibilidade é reduzida.

RESTful

De forma a que uma API possa ser considerada *RESTful* esta deve seguir estritamente as regras definidas na arquitetura REST explanadas anteriormente. Contudo, ao longo do tempo sempre houve bastantes dúvidas ao classificar uma API. Tendo isso em conta, Leonard Richardson desenvolveu um modelo de Maturidade, modelo de Richardson, que apresentamos de seguida na figura 7:

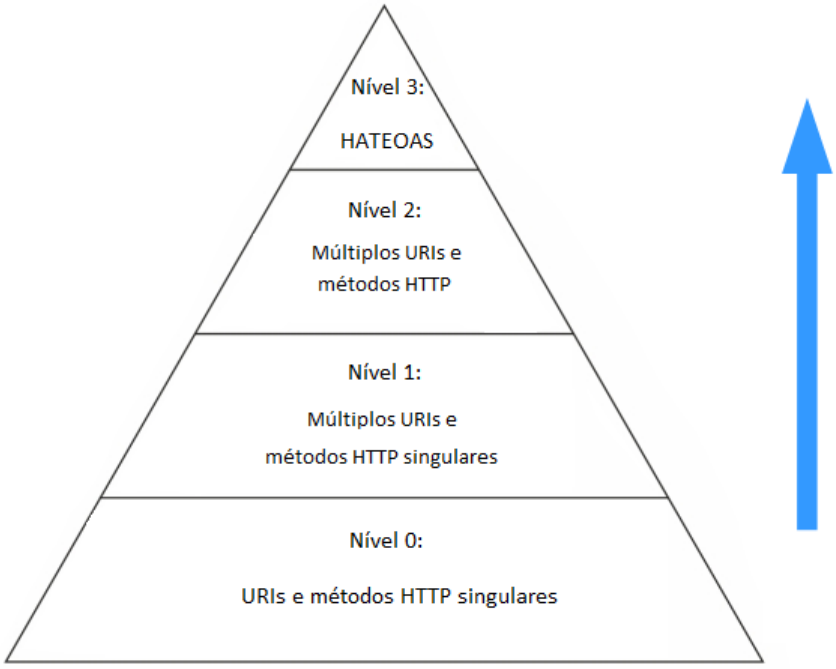


Figura 7: Modelo de maturidade de Richardson

Nível 0 - Serviço de Transporte

O nível 0, o mais básico do modelo, representa os serviços que têm apenas um URI e que usam somente um método HTTP sem qualquer critério para a utilização de métodos, ou de rotas. Um exemplo típico de serviços que se enquadram neste nível são os serviços SOAP.

Nível 1 - Recursos

Neste nível fazemos uso de recursos para modelar a API, os serviços deste nível disponibilizam vários URIs para aceder aos recursos expostos. Deste modo, em vez de se fazer todos os pedidos para um único URI, que representa um recurso, os pedidos são feitos para os vários recursos disponíveis, usando para isso diferentes URIs. Os recursos são mapeados, mas ainda não emprega o uso eficiente dos verbos. Geralmente utilizam apenas GET e POST.

Nível 2 - Verbos HTTP

Assim como no nível anterior, os serviços do nível 2 disponibilizam vários URIs mas cada URI suporta múltiplos métodos HTTP. Trata-se de serviços que usam devidamente os verbos (ou métodos) HTTP: GET para

leitura, POST para inserir, PUT para substituir um registo, PATCH para modificar parcialmente o conteúdo de um recurso e DELETE para eliminar.

Nível 3 - HATEOAS

O nível três de maturidade faz o uso eficiente dos três fatores: URI, HTTP e HATEOAS.

Uma API que implementa este nível fornece aos seus clientes links que indicarão como poderá ser feita a navegação entre os seus recursos. Isto é, quem for consumir a API precisará de saber apenas a rota principal e na resposta dessa requisição terá todas as demais rotas possíveis.

Apenas APIs neste nível podem ser consideradas *RESTful*.

2.5 GRAPHQL

GraphQL é uma tecnologia inicialmente criada no Facebook em 2012 sendo em 2015 lançado como *open sourced*. É descrito como "uma linguagem de consulta para APIs"(GraphQL) e foi construído como resposta a um problema de desempenho durante a mudança do Facebook para aplicações móveis nativas (Byron).

Basicamente, é uma tecnologia para construir APIs desenhada para ser uma alternativa ao REST uma vez que corrige algumas das suas fraquezas:

- O problema de *overfetching*, quando a resposta contém mais dados do que aqueles necessários;
- O problema de fazer muitos pedidos de HTTP para vários tipos de recursos ou o problema de ter de fazer vários pedidos para o mesmo recurso, um por um;
- O problema de, por vezes, os Uniform Resource Locators (URLs) ficarem demasiado confusos com relações complexas e com muito *nesting*.

As principais características são as seguintes:

- Utiliza apenas um único *endpoint* usando o protocolo HTTP;
- Um *type system* é usado para definir e descrever os dados;
- O cliente especifica exatamente os dados de que precisa;
- Múltiplas fontes de dados podem ser consultadas na mesma solicitação;
- GraphQL segue o relacionamento entre objetos, hierarquicamente. Os dados devolvidos ao cliente seguem uma estrutura gráfica.

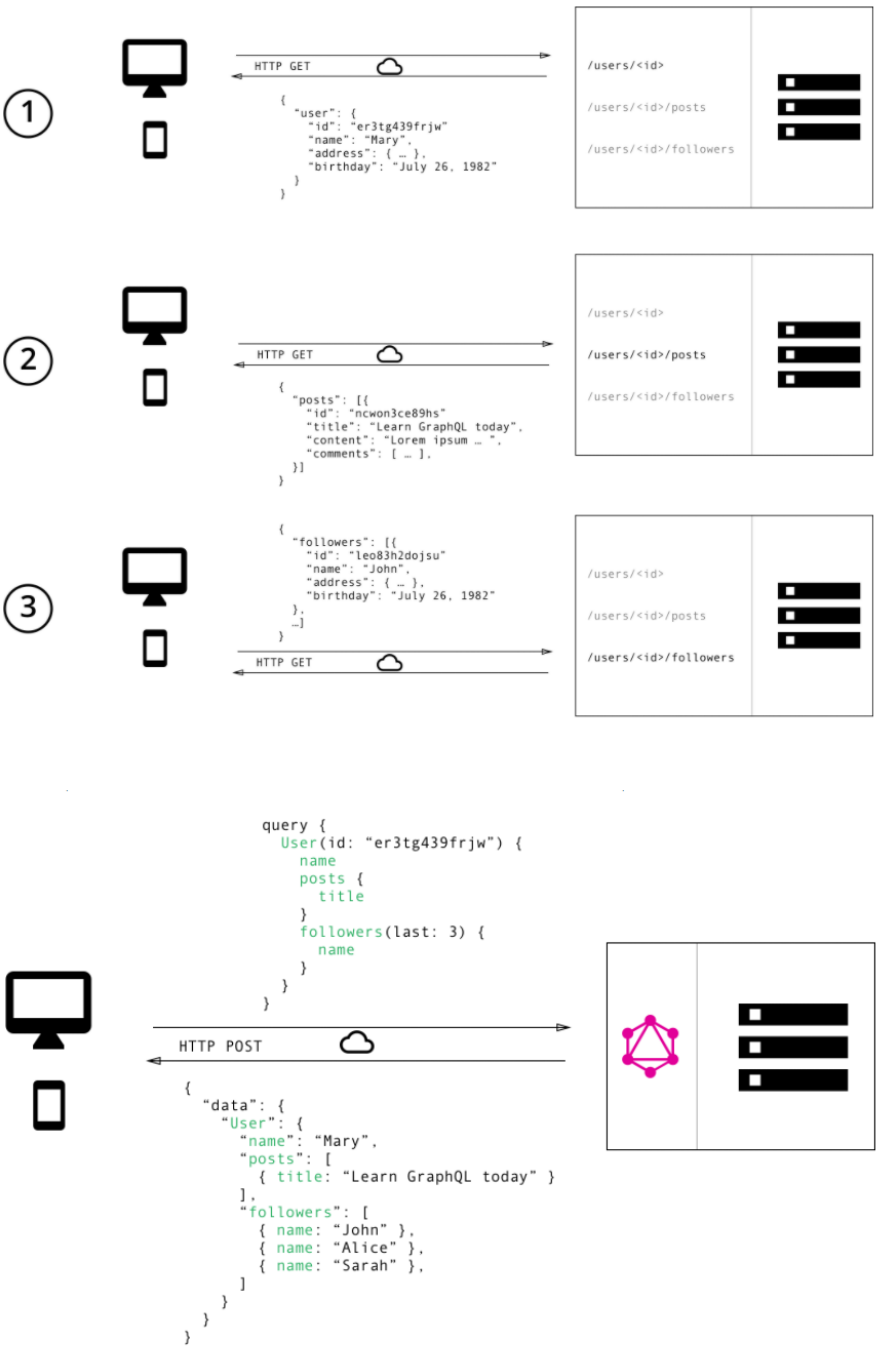


Figura 8: As mensagens no paradigma REST (em cima) e GraphQL (em baixo). (Fonte: howtographql.com)

Como demonstra a figura 8, no GraphQL escrevemos simplesmente uma *query* singular para o servidor GraphQL que inclui os requisitos dos dados a obter. De seguida, o servidor responde com um objeto JSON contendo apenas os dados solicitados.

2.6 GRPC

gRPC significa Google Remote Procedure Call e é uma moderna *open source framework* de RPC que pode ser executada em qualquer ambiente criada pela empresa Google (gRPC Authors).

O gRPC fornece alto desempenho e comunicação RPC segura entre serviços, bem como vários tipos de *streaming* de dados. O alto desempenho e a velocidade de transferência são garantidos principalmente pelos *buffers* de protocolo (*protobuf*) e pelo uso do protocolo HTTP/2 para a transferência de dados.

Através dos *buffers* de protocolo, é possível otimizar o tráfego de dados devido à sua serialização, na qual são compactados os dados de forma binária antes do envio ao cliente, que possui um interpretador destes dados. Os interpretadores e chamadas de serviços são gerados automaticamente a partir dos arquivos escritos na linguagem proto, que definem o modo pelo qual as aplicações se comunicam.

Ao usar o HTTP/2 permite seguir um modelo de comunicação de resposta-cliente. Estas condições suportam comunicação bidirecional e de *streaming* devido à capacidade do gRPC de receber múltiplos pedidos de vários clientes e lidar com esses pedidos simultaneamente através do *streaming* constante de informação. Além disso, o gRPC também pode lidar com interações *unary* como as construídas em HTTP 1.1 usadas pela arquitetura REST [9] . Aliás, este é um dos pontos fortes da tecnologia gRPC em relação ao REST.

Apesar das APIs REST também poderem ser construídas em HTTP/2, o modelo de resposta a pedidos de comunicação permanece o mesmo, o que proíbe as APIs de REST de aproveitar ao máximo as vantagens de desempenho do HTTP/2, tais como comunicação de *streaming* e suporte bidirecional.

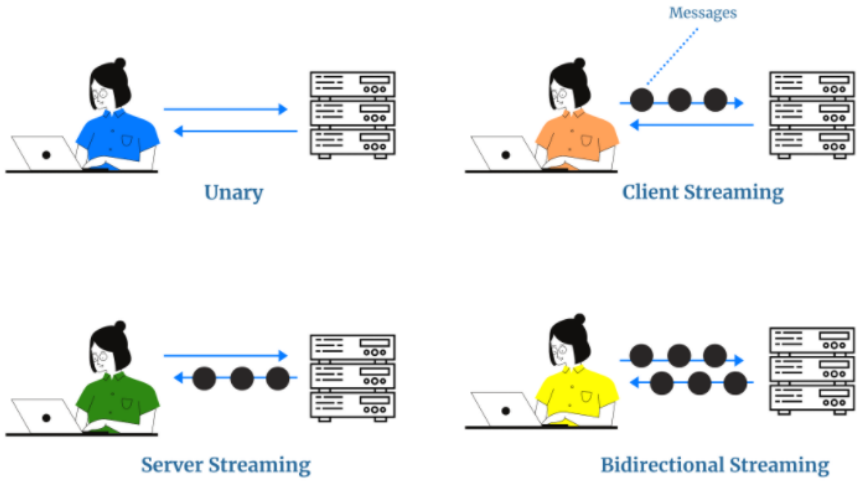


Figura 9: Tipos de *streaming* (Fonte: imaginarycloud.com)

2.7 SEGURANÇA API

A maioria das APIs é exposta à Internet, portanto, é necessário ter mecanismos de segurança adequados para evitar abusos, proteger dados confidenciais e garantir que apenas utilizadores autenticados e autorizados possam acedê-los.

Por mais que a autenticação impulsione a Internet moderna, este tópico costuma ser confundido com um termo intimamente relacionado: autorização.

Autenticação é quando uma entidade prova uma identidade. Por outras palavras, a autenticação prova que o utilizador é de facto quem diz ser. Por sua vez, a autorização é o processo que ocorre após ser validada a autenticação. A autorização define quais os direitos e as permissões que o utilizador do sistema tem. Dito isto, a autorização prova que o utilizador tem o direito de aceder ou não a determinados recursos.

A segurança começa com a própria conexão HTTP. APIs REST seguras devem fornecer apenas terminais Hypertext Transfer Protocol Secure (HTTPS) para garantir que toda a comunicação da API seja criptografada usando Transport Layer Security (TLS). Isso permite proteger as credenciais da API e os dados transmitidos.

Muitas APIs da web estão disponíveis apenas para utilizadores autenticados, por exemplo, porque são privadas ou exigem registo ou pagamento. Os métodos de segurança em termos de autenticação e autorização mais comuns nas APIs REST são Neumann et al. (2018):

- Basic Authentication: as credenciais são enviadas diretamente nos cabeçalhos HTTP na codificação Base64 sem criptografia. Este é o método de autenticação mais simples e mais fácil de implementar. Também é o menos seguro, uma vez que os dados confidenciais são transmitidos como texto simples, portanto, só deve ser usado em combinação com HTTPS;

- API Keys: Na autenticação baseada em chaves de API é gerada uma chave, exclusiva, por cada cliente. Quando o servidor recebe um pedido, com uma chave, irá verificar se esta é uma chave válida. Também só devem ser usadas em conjunto com canais de comunicação seguros uma vez que o comprometimento da *API Key* permitiria a qualquer indivíduo executar operações como se do próprio cliente se tratasse;
- OAuth: Mecanismos OAuth 2.0 padrão podem ser usados para autorização. Estes permitem às aplicações comunicar com os recursos hospedados em serviços de terceiros sem requerer a partilha da password por parte do utilizador;
- JSON Web Token (JWT): Credenciais e outros parâmetros de acesso são enviados como estruturas de dados JSON. Esses *tokens* de acesso podem ser assinados criptograficamente e são a forma preferencial de controlar o acesso às APIs REST.
- OpenID Connect: O OpenID Connect é construído sobre o OAuth 2 e é um protocolo de autenticação que acrescenta a camada de identificação. O OpenID Connect pode fornecer um *token* de acesso mais um *token* de identificação. O OpenID Connect foi projetado para ser a resposta para a maioria dos casos de uso SAML/WS-Fed sem a sobrecarga baseada em XML e SOAP para aplicações modernas.

2.7.1 *Vulnerabilidades*

Segundo a Open Web Application Security Project (OWASP), entidade que se define como uma "nonprofit foundation that works to improve the security of software. Through community-led open-source software projects, hundreds of local chapters worldwide, tens of thousands of members, and leading educational and training conferences, the OWASP Foundation is the source for developers and technologists to secure the web.", a lista das vulnerabilidades mais recorrentes em APIs em 2019 é OWASP (a):

API1: Broken Object Level Authorization (Quebra da autorização ao nível do objeto)

Os atacantes substituem o ID do seu próprio recurso na chamada à API por um ID de um recurso pertencente a outro utilizador. A falta de verificações de autorização adequadas permite que os atacantes acedam ao recurso desejado.

API2: Broken authentication (Quebra de autenticação)

As funções da aplicação relacionadas com a autenticação e a gestão da sessão geralmente são implementadas de forma incorreta, permitindo que os atacantes comprometam senhas, chaves e *tokens* de sessão ou, ainda, explorem outras falhas de implementação para assumir a identidade de outros utilizadores.

API3: Excessive data exposure (Exposição de Dados Sensíveis)

A API pode expor muito mais dados do que o cliente legitimamente precisa, contando com o cliente para fazer a filtragem. Se os atacantes forem diretamente à API, terão acesso a tudo.

API4: Lack of resources and rate limiting (Falta de recursos e de limites de taxa de utilização)

A API não é protegida contra uma quantidade excessiva de chamadas ou tamanhos de carga útil.

Isso pode não apenas impactar o desempenho do servidor API, levando à negação de serviço (DoS), mas também torna possível a existência de falhas de autenticação através de ataques de força bruta.

API5: Broken function level authorization (Quebra do nível da função de autorização)

Políticas complexas de controle de acesso com diferentes hierarquias, grupos e funções, e uma separação pouco clara entre funções administrativas e regulares, tendem a levar a falhas de autorização. Ao explorar esses problemas, os atacantes obtêm acesso aos recursos de outros utilizadores e/ou a funções administrativas.

API6: Mass assignment (Atribuição em massa)

A API recolhe os dados fornecidos pelo cliente e armazena-os sem a filtragem adequada das propriedades baseadas na lista de permissões. Os atacantes podem tentar adivinhar as propriedades do objeto ou fornecer propriedades adicionais nas requisições, ler a documentação ou verificar os *endpoints* da API em busca de pistas de modo a encontrar falhas para modificar as propriedades de objetos que não eram supostos.

API7: Security misconfiguration (Configuração Incorreta de Segurança)

Atacantes podem conseguir executar ações indesejadas por falta de configuração correta de segurança como por exemplo: utilização de configurações *default* inseguras, software desatualizado, protocolos implementados incorretamente e *features* desnecessárias ativas.

API8: Injection (Injeção)

As falhas de Injeção, tais como injeção de SQL, de sistema operativo, etc., ocorrem quando dados não confiáveis são enviados para um interpretador como parte de um comando ou consulta. Os dados manipulados pelo atacante podem iludir o interpretador para que este execute comandos indesejados ou permita o acesso a dados não autorizados.

API9: Improper assets management (Gestão inadequada de ativos)

Os atacantes encontram versões não finais da API (por exemplo, de teste ou versões anteriores) que não são tão protegidas quanto a API final e utilizam-nas para fazer os seus ataques.

API10: Insufficient logging and monitoring (Login e monotorização insuficientes)

A insuficiência no registo de eventos e monitorização, em conjugação com a falta ou ineficácia da integração com o alerta de incidentes, permite que ataques e os seus respetivos autores passem despercebidos levando a que estes possam persistir os seus ataques, alcançar outros sistemas e ainda extrair ou destruir dados.

2.7.2 *Práticas a seguir para o desenvolvimento de arquiteturas REST seguras*

No seguimento das vulnerabilidades apresentadas, existe também um conjunto de boas práticas de segurança que devemos ter em conta quando estamos a desenvolver uma API REST (OWASP, b):

Restringir métodos HTTP

- Criar uma lista com apenas os métodos HTTP que são permitidos;
- Rejeitar todos os pedidos de métodos HTTP que não pertençam à lista com um código de resposta HTTP adequado;
- Assegurar que o utilizador está autorizado a usar o método HTTP no recurso em questão.

Validação de Input

As APIs são projetadas para acesso automatizado sem a necessidade de interação por parte do utilizador, por isso é especialmente importante garantir que todas as entradas sejam válidas e de acordo com o esperado. Todas as requisições que não estejam em conformidade com a especificação da API devem ser rejeitadas. As diretrizes típicas de melhores práticas para validação de input são:

- Considerar todos os parâmetros, objetos e dados de input como não confiáveis;
- Verificar o *request size*, o *content-length* e o *content-type*;
- Usar *strong typing* nos parâmetros da API;
- Usar *logs* para registar *input validation failures* de modo a avaliar possíveis ocorrências de ataques;
- Restringir o input de *strings* sempre que possível com *regexps*;
- Usar *frameworks* ou bibiotecas de validação associados à linguagem de programação.

Validar content-types

O corpo de um pedido ou de uma resposta deve corresponder ao tipo de conteúdo pretendido no cabeçalho. Os serviços REST devem definir precisamente os tipos de conteúdo permitidos e rejeitar as requisições que não tenham as declarações corretas nos seus cabeçalhos HTTP. Isso significa especificar cuidadosamente os tipos permitidos nos cabeçalhos *Content-Type* e *Accept*.

Para além disso, é comum nos serviços REST permitir-se múltiplos tipos de resposta (por exemplo, *application/xml* ou *application/json*), e o cliente deve especificar qual prefere através do cabeçalho *Accept* ao fazer o pedido. Caso o cliente especifique um tipo não permitido, deve-se rejeitar o pedido.

Fazer tratamento de erros

Usar mensagens de erro genéricas: evitar revelar demasiados detalhes da falha ocorrida ou informações técnicas para o cliente.

Usar Security Headers

Para garantir que o conteúdo de um determinado recurso seja interpretado corretamente pelo navegador, podem ser definidos cabeçalhos de segurança HTTP adicionais para restringir ainda mais o tipo e o âmbito das requisições. Alguns destes exemplos são:

- *X-Content-Type-Options*: *nosniff* para prevenir ataques XSS baseados em MIME *sniffing*;
- *X-Frame-Options*: *deny* para prevenir tentativas de *clickjacking* em navegadores mais antigos;
- *Content-Security-Policy*: *frame-ancestors 'none'* é uma *browser security policy* mais recente do que o header anterior. Mesmo que as respostas da API não sejam renderizadas em *frames*, nada impede que isso aconteça. Para garantir que as respostas da API não sejam vulneráveis a ataques obscuros, é recomendável definir ambos os headers em cada resposta da API.

Ter cuidado com o uso de CORS

Cross-Origin Resource Sharing (CORS) é uma especificação do W3C que, quando implementado pelo navegador, permite que um site aceda a recursos de outro *website* mesmo estando em domínios diferentes. Se o serviço não suportar *cross-domain calls*, então deve-se desativar o cabeçalho CORS. Caso o mecanismo CORS seja necessário, devem ser especificadas concretamente quais as origens permitidas.

Ter cuidado com a partilha de informação sensível nos pedidos HTTP

As chamadas de APIs geralmente incluem credenciais, chaves de API, *tokens* de sessão e outras informações confidenciais. Se incluídos diretamente nos URLs, estes detalhes podem ser armazenados em *logs* dos servidores da web e descobertos se os *logs* forem acedidos por *hackers*. Para evitar a divulgação de informações confidenciais, os serviços da Web REST devem sempre enviá-las no cabeçalho do pedido HTTP ou no corpo do pedido (para pedidos POST e PUT).

Usar HTTP Return Codes

Tal como referido nos capítulos anteriores, HTTP define códigos de status de resposta que são também um excelente padrão no que concerne a boas práticas de segurança.

2.7.3 *API Security Tests*

As arquiteturas REST, pelo facto de utilizarem o protocolo HTTP, partilham das mesmas vulnerabilidades e riscos de aplicações e *websites* mais tradicionais.

Em concordância com B. Potter (2004), garantir que um software é seguro é assegurar que esse mesmo software se comporta da forma esperada ainda que na presença de ataques maliciosos.

Os testes de segurança têm como principal objetivo validar a garantia de determinadas propriedades de segurança como a confidencialidade, integridade, autenticidade, disponibilidade e não-repúdio.

Podemos dividir os testes em dois grupos, testes de caixa-branca e testes de caixa-preta (Gary Hoglund, 2004). A diferença entre elas é que técnicas *White-Box* são técnicas usadas tendo conhecimento de toda a estrutura interna e externa do código, enquanto que as *Black-Box* são técnicas que usufruem dos requisitos e das funcionalidades do sistema para poderem criar cenários de teste. Além disso, é feito também referência a testes de *Gray Box* que consistem na utilização conjunta dos testes de White Box e testes de entradas *Black-Box*. Nestes casos, a análise *White-Box* é utilizada para encontrar áreas vulneráveis, e os testes *Black-Box* são usados para desenvolver ataques contra essas áreas.

Para ambas as situações, testes de caixa-branca e testes de caixa-preta, podem ser criados testes manualmente, no entanto, o tempo e o dinheiro podem ser recursos escassos para a maioria das empresas que desenvolvem software. É por isso que nos últimos anos têm surgido cada vez mais técnicas e ferramentas para testes automáticos para ambos os tipos (Iversen, 2018).

Exemplos de técnicas de testes

Uma das mais reconhecidas técnicas de caixa-branca é o *Code Coverage*. *Code coverage* pode ser definido como uma medida usada para descrever a percentagem de código que é executado quando um determinado conjunto de testes automatizados é executado.

Code coverage é executado pelos *developers* através de testes unitários para verificar a implementação do código de maneira a que quase todas as instruções de código sejam executadas. No geral, um sistema de *code coverage* coleta informações sobre o programa em execução e combina-as para gerar um relatório sobre o código coberto pelo conjunto de testes.

Penetration testing é um dos exemplos mais conhecido de técnicas de teste de caixa-preta. Permite proteger a aplicação de vulnerabilidades que possam ter surgido durante o desenvolvimento. Neste processo, os aspectos externos da API são atacados de forma deliberada num ambiente controlado.

As principais etapas durante um *Penetration test* são:

- Identificar uma lista de potencias vulnerabilidades aplicáveis;
- Ordenar essa lista de acordo com o grau de risco. Podemos classificar o risco de acordo com OWASP;
- Criar pedidos de modo a que incorporem esses ataques e usá-los no sistema;

- Se for feito um acesso não autorizado ao sistema, deve ser apresentado um relatório da vulnerabilidade e proceder à correção do problema.

Um teste de segurança ainda mais aprofundado no contexto de *Black-Box* é o *Fuzz testing*. *Fuzz testing* é o aspecto final de um processo de auditoria de segurança, no qual uma API é levada ao seu limite. Isso pode ser feito enviando grandes volumes de solicitações, tentando variar os dados de tantas maneiras criativas quanto possível para cobrir todas as possibilidades de vulnerabilidades emergentes com volumes elevados e que levem ao comprometimento da segurança. Essas vulnerabilidades podem ser exploradas por ataques de negação de serviço(DoS) ou *Overflow attacks*. *Fuzz testing* define os parâmetros de operação para valores inesperados de forma a provocar comportamentos inesperados e erros no *backend* da API.

2.8 TRABALHOS RELACIONADOS

Esta subsecção apresenta soluções no contexto do trabalho proposto. Deste modo, para uma melhor exposição, os trabalhos relacionados estão subdivididos em duas partes: trabalhos que desenvolveram APIs com base no estilo arquitetural REST e trabalhos na área da segurança de aplicações web.

O trabalho desenvolvido por Marques (2018) está inserido no contexto de desenvolvimento Web com base na arquitetura REST. Utilizou Python como linguagem de programação e a *framework* Django REST, utilizando o padrão Model-View-Controller (MVC) com a finalidade de desenvolver uma API REST para aplicação *cloud* num modelo de Software as a Service (SaaS).

De uma forma geral, podemos afirmar que tem sido realizado um trabalho notório quanto à classificação e avaliação de quais as ameaças de segurança mais críticas para as aplicações web. Tem-se tentado também aumentar a eficácia de deteção e a automação dos scanners de segurança. A OWASP é um exemplo de uma comunidade online que cria e disponibiliza de forma gratuita documentação, artigos, metodologias, ferramentas e tecnologias no campo da segurança de aplicações web. Aliás, uma das suas publicações de maior relevo é o *Top Ten* de vulnerabilidades web mais comuns OWASP (a) abordado anteriormente. No mesmo sentido, é importante destacar também o trabalho de APISecurity.io que é um website comunitário contemplando tudo acerca do tema de segurança em APIs, incluindo artigos, notícias, violações, vulnerabilidades, regulamentos, tecnologias, práticas recomendadas e até mesmo uma enciclopédia de segurança de APIs.

A tese de Ratos (2019) também se relaciona com este projeto uma vez que este estudou vulnerabilidades nos serviços API REST e as técnicas usadas para as encontrar, desenhou e implementou uma solução capaz de encontrar estas vulnerabilidades num cenário onde o sistema vulnerável é um servidor que expõe uma API REST.

3

PROJETO

3.1 METODOLOGIA DE TRABALHO

Existem diversas opções diferentes disponíveis para desenvolvimento web, tanto modernas quanto clássicas, e selecionar a abordagem apropriada é importante. Todas as opções têm os seus prós e contras, e as necessidades e requisitos individuais devem influenciar a decisão.

Inicialmente, foi definido com a empresa a adoção de uma metodologia Agile como abordagem elegida para o desenvolvimento do trabalho. A filosofia Agile foi arquiteturada por um conjunto de 17 pessoas em 2001 que a definiu através de um conjunto de valores e príncipios, sendo estes os seguintes (Beck et al., 2001):

Valores:

- Indivíduos e interações importam mais do que processos e ferramentas;
- Software a funcionar é mais importante do que ter a documentação completa;
- Colaboração com o cliente é mais relevante do que a negociação de contratos;
- Responder a alterações é prioritário a seguir um plano.

Princípios:

- A maior prioridade está em satisfazer o cliente por meio da entrega adiantada e contínua de software de valor;
- Alterações tardias ao projeto são aceites e bem-vindas;
- Entregar o software em funcionamento com frequência;
- Tanto pessoas relacionadas a negócios como *developers* devem trabalhar em conjunto durante todo o curso do projeto;
- Para construir projetos ao redor de indivíduos motivados, é preciso dar-lhes o ambiente e o suporte necessários, confiando assim que farão o seu trabalho bem;

- O método mais eficiente de transmitir informações tanto externas como internas para uma equipa de desenvolvimento é por meio de uma conversa cara a cara;
- Um software funcional é a medida primária de progresso;
- Processos ágeis promovem um ambiente sustentável. Os patrocinadores, *developers* e utilizadores devem ser capazes de manter um ritmo constante indefinidamente;
- A contínua atenção à excelência técnica e ao bom design aumenta a agilidade;
- Simplicidade: arte de maximizar a quantidade de trabalho que não é preciso ser feito;
- As melhores arquiteturas, os requisitos e os designs emergem de equipas auto-organizadas;
- Em intervalos regulares, a equipa deve refletir em como tornar-se mais efetiva, ajustando e otimizando o seu comportamento de acordo.

Este processo é notório por ajudar as equipas a fornecer respostas rápidas e imprevisíveis ao feedback que recebem no seu projeto e a criar oportunidades para avaliar a direção de um projeto durante o ciclo de desenvolvimento.

No nosso caso, o trabalho foi realizado segundo uma lógica de Scrum, que é uma metodologia Agile, através de sprints semanais, em que reunia com os orientadores da empresa, apresentava os progressos efetuados durante a semana anterior e, de seguida, eram distribuídas as tarefas para a semana seguinte. Para além disso, existiam também reuniões em que cada um dos elementos transmitia à equipa como estava o estado do seu trabalho e quais os seus bloqueios de forma a ter conhecimento alargado do estado das tarefas e a partilhar soluções quando surgiam problemas.

Isto resultou na realização de aproximadamente 50 reuniões ao longo do estágio.

3.2 UN1QNX

Tal como referido no capítulo introdutório, a UN1Qnx é uma empresa especializada em *brand protection* e *product authentication*.

O produto arquitetado pela UN1Qnx é uma peça física desenhada especificamente para assegurar a autenticidade de um produto.

Estas peças, denominadas de tags, são impossíveis de replicar uma vez que são concebidas com vários níveis de camadas diferentes.

As tags, apesar de garantirem sempre o mesmo nível de autenticidade, podem ser associadas com qualquer produto de diferentes formas:

- Standard - a tag é colocada na etiqueta descartável do produto;
- Attached – a tag é anexada ao produto;

- Embedded – a tag é embutida no produto.

Para a validação das mesmas, a UN1Qnx desenvolveu um algoritmo de visão por computador que utiliza métodos de *deep learning* de modo a obter resultados confiáveis.

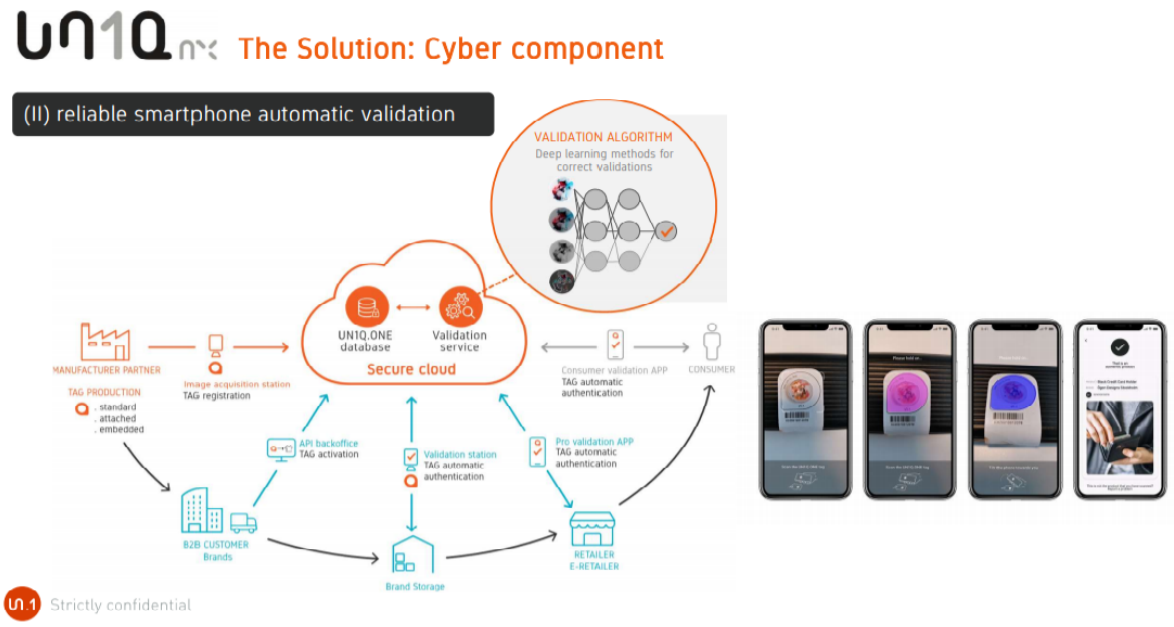


Figura 10: Diagrama do sistema UN1Qnx

O processo envolvido no serviço UN1Qnx é o seguinte:

- As tags após serem criadas, são digitalizadas por uma *image acquisition station* e são registadas na base de dados da UN1Qnx;
- As tags são posteriormente enviadas para os clientes (*brands*) e estes, de cada vez que associam uma tag a um produto, devem proceder à sua ativação através de uma API de *backend*;
- O consumidor final ou um intermediário do processo (por exemplo, revendedores) pode verificar a autenticidade do produto através de uma aplicação móvel ou de uma estação de validação que permite fazer o *scan* da *tag* que implementa o serviço de validação supracitado.

O serviço de garantia de autenticidade UN1Qnx tem como um dos seus pilares operacionais a validação digital automática, estruturada a partir de uma base de dados SQL. Ambos estes componentes fazem uso de recursos *cloud*. Neste âmbito, é importante garantir a integridade e confidencialidade de toda a infraestrutura IT, e definir uma arquitetura que maximize a escalabilidade da mesma mas também que possa otimizar todos os seus processos.

O intuito desta parte prática da dissertação é então contribuir para o desenvolvimento da arquitetura da API disponibilizada aos clientes da UN1Qnx.

3.3 WEB API

Neste capítulo é feita a descrição da API, destacado o porquê da escolha da abordagem REST face às restantes opções, da arquitetura desenvolvida, das suas funcionalidades, de alguns dos contratempos e limitações que ocorreram e da documentação gerada.

Depois de toda a investigação feita em torno de REST e SOAP, foi confirmado que a abordagem REST seria a mais indicada para resolver o problema em questão. Esta conclusão foi maioritariamente suportada pelas seguintes características:

- Em primeiro lugar, cada objeto físico ou conceito abstrato ao qual podemos nos referir é identificado como um recurso único. Isso implica que cada um deles pode ser acedido rapidamente por apenas um pedido depois de identificados. Isso resulta numa melhor navegabilidade e experiência para o utilizador e menos carga para o servidor;

- Em segundo lugar, devido ao princípio da ausência de estado, o REST torna o sistema realmente escalável, uma vez que os servidores não guardam nenhuma informação de nenhum dos clientes. Isso significa que cada máquina que faz parte do sistema é capaz de receber qualquer tipo de solicitação de qualquer um dos clientes e gerar uma resposta adequada. Outro efeito deste princípio é que os complexos balanceadores de carga não são mais necessários e os clientes que estão a usar o sistema não se apercebem quando um servidor cai, uma vez que o sistema se torna tolerante a falhas e recuperável;

- Além disso, há maior facilidade na mudança do código do lado do cliente seguindo a abordagem REST ao contrário de SOAP, se um serviço Web do tipo REST não vier com todas as funcionalidades para o cliente, a *user experience* não será tão prejudicada, já que, para obter novas funcionalidades, o cliente terá apenas de fazer a aquisição da nova versão. Visto que a aplicação está em constante desenvolvimento, faz sentido a adoção da abordagem REST;

- Por último, a capacidade de cache, que pode ser facilmente alcançada usando os cabeçalhos HTTP corretamente, reduz a carga nos servidores e melhora o tempo de resposta que é traduzido novamente numa melhor experiência para o utilizador.

Foram também analisadas e discutidas as opções GraphQL e gRPC que são relativamente novas. GraphQL é mais aconselhado para ser usado por aplicações *mobile* que precisem de realizar *queries* a dados *nested*. No caso do nosso projeto, REST sobrepõe-se uma vez que permite o upload de ficheiros e o uso de cache, para além de ser mais fácil a monitorização e o controlo de erros. gRPC é uma variante baseada na arquitetura RPC e é completamente diferente dos dois anteriores mas oferece muitas vantagens. Ao contrário do REST, aproveita ao máximo o HTTP/2, usando *streams* multiplexadas e seguindo o protocolo binário. Além disso, oferece benefícios de desempenho devido à estrutura da mensagem *Protobuf*. Esses motivos tornam o gRPC uma opção bastante promissora para a criação de novas APIs. No entanto, o baixo suporte de *browser* e o facto de a integração com os clientes ser complicada devido à dificuldade de implementação torna um desafio

competir com o suporte universal do REST. REST continua a ser, no momento, a solução mais popular uma vez que é uma arquitetura muito bem desenvolvida e bem-sucedida.

3.4 ARQUITETURA INICIAL

Visto que, o objetivo do projeto é criar uma nova versão de uma API já existente, com o intuito de recorrer a tecnologia atual, para além das características de robustez e segurança que são pretendidas, optou-se por desenvolver as soluções recorrendo ao que já estava a ser utilizado internamente : .NETCore para o desenvolvimento web e Microsoft SQL Server como sistema de base de dados:

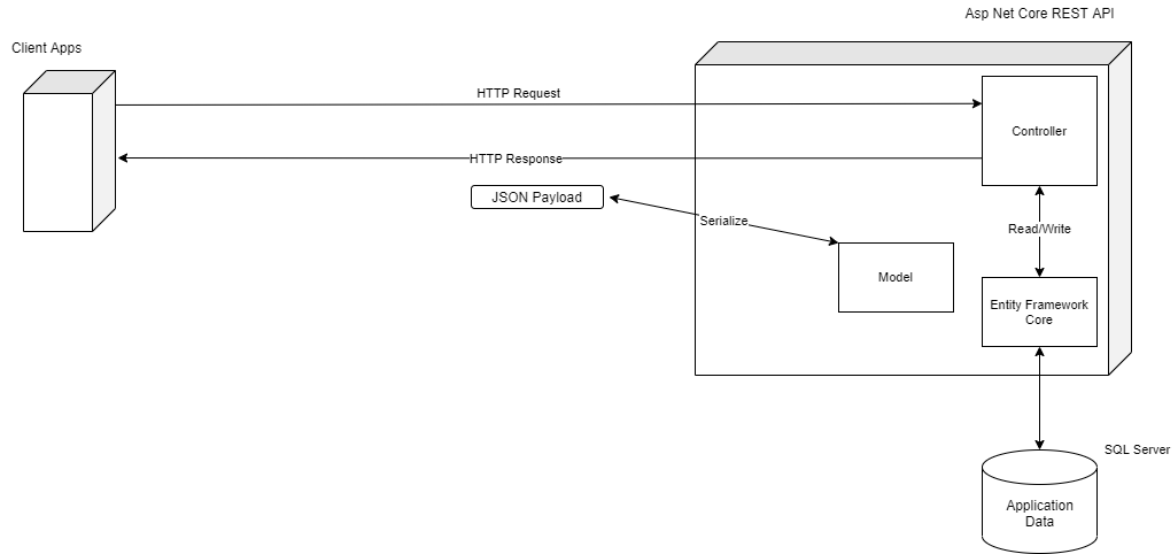


Figura 11: Arquitetura inicial

De seguida são descritas as principais tecnologias utilizadas na implementação da solução.

ASP.NET Core é uma ferramenta multiplataforma e de código aberto para o desenvolvimento de aplicações Web modernas. Suporta vários paradigmas de programação modernos e oferece o desempenho necessário para o projeto (Microsoft, a).

ASP.NET Core Identity é um serviço que fornece uma solução fácil de usar para lidar com a autenticação e gestão de utilizadores no desenvolvimento do ASP.NET Core.

Entity Framework Core (EF Core) é uma versão multiplataforma do Entity Framework e é usado como comunicação com a base de dados. EF Core pode servir como um mapeador relacional de objeto, que permite aos programadores trabalhar com a base de dados usando objetos .NET, bem como abstrair a lógica de base de dados subjacente. Outra vantagem que lhe é associada é o facto de oferecer suporte a muitos mecanismos de base de dados diferentes, o que significa que o mesmo código funcionará caso o mecanismo de base de dados seja alterado no futuro (Microsoft, b) .

O Microsoft SQL Server é um sistema de gestão de base de dados relacional, criado para armazenar e retornar dados através de aplicações diferentes. Foi escolhido pelo grupo do projeto como o sistema de gestão de base de dados, por se tratar de um sistema com o qual a equipa tinha familiaridade e experiência.

3.5 DESIGN

Este capítulo descreve o processo de design da API REST, ou seja, descreve a estrutura que a API REST terá, como ela será construída, que módulos farão parte , o que cada módulo fará e como eles irão interagir entre si. Além disso, é explicado como os princípios REST serão devidamente aplicados para que a API possa ser considerada *RESTful* (Rodríguez et al., 2016). Além disso, uma explicação do fluxo de trabalho para os principais casos de uso é fornecido.

3.5.1 *Recursos*

A primeira atividade do processo de design é identificar os recursos que vão ser acessíveis por meio da API. Isso foi conseguido com a obtenção de um esboço de como o sistema UN1Qnx se comporta e também falando e discutindo acerca disso com os *developers* da UN1Qnx que trabalham diariamente com o sistema.

A seguir, é fornecida uma explicação do que cada recurso representa. Também é descrito que dados eles contêm e que parâmetros são necessários para criá-los no momento de hoje no sistema UN1Qnx:

- Tags – O recurso tags é o principal recurso do sistema UN1Qnx e representa a peça física associada a um determinado produto de um determinado tenant. Para criar uma nova tag é necessário: Barcode, TagTypeId, ProductId, BatchId e TenantId;
- TagTypes - O recurso tagTypes representa os tipos de tags existentes no sistema UN1Qnx. Para criar um novo tagType é necessário: Name e Description;
- TagHistory - O recurso tagHistory regista todas as alterações feitas a uma determinada tag. É criado um novo TagHistory sempre que se altera o estado de uma tag;
- Products - O recurso products representa um produto de um determinado tenant. Para criar um novo produto é necessário: Name, Brand, Supplier, Description, SerialNumber, SerialNumberType e TenantId;
- Tenants - O recurso tenants representa os clientes da UN1Qnx. Para criar um novo tenant é necessário: Name, Website, Email, Description, Nif, ContactPersonName e ContactPersonEmail;
- User - O recurso user representa um utilizador da app ou do *backoffice* da UN1Qnx. Para criar um novo user é necessário: Name, Email, Password, PhoneNumber e user role;
- AcquisitionStations - representa o *pipeline* de validação a usar aquando de uma validação;

- ValidationSessions - é a construção que permite que o validador e o utilizador estabeleçam as condições de uma validação de uma tag.
- Validations - representa o resultado de uma validação.

3.5.2 *Identificação de recursos via URI*

Um dos princípios básicos do REST é que cada recurso deve ser endereçável através de um único URI, conforme dito na Seção 2.4 . Para cumprir este requisito na API desenvolvida, um URI único e previsível será atribuído a cada recurso:

- A primeira parte do URI mostra qual tipo de recurso o cliente pretende utilizar. Portanto, a primeira parte será /tags, /products , etc. Esses URIs representam o conjunto de tags, produtos, etc.
- Se o URI tiver uma segunda parte, ele representa o identificador do recurso ao qual uma ação é aplicada. Portanto, uma ação aplicada ao URI /tags/123 será endereçada à tag com o id número 123. Com esta estrutura de URI é alcançada uma forma simples, compreensível, fácil de usar e abordar diretamente um recurso específico.

3.5.3 *Ausência de estado*

Conforme explicado na Seção 2.4, cada solicitação HTTP deve acontecer de forma completa e isolada e nenhuma informação sobre o cliente deve ser armazenada no servidor de um pedido para outro. Resumindo, a API REST será completamente *stateless* seguindo o REST princípios que resultarão num sistema mais escalável.

3.5.4 *HATEOAS*

Uma aplicação bem conectada permite que o utilizador descubra por si mesmo a interface. Os recursos devem estar conectados a outros recursos e sua representação deve oferecer links de outros recursos e para outras acções do mesmo recurso. Este recurso é um dos mais simples de fornecer na API REST que foi projetada. Cada lista de recursos, como tags, products, etc. contém um link para os recursos relacionados:

```
{
  "serialNumberType": null,
  "tenantId": 2,
  "links": [
    {
      "href": "https://localhost:5001/api/v2/products/5",
      "rel": "self",
      "method": "GET"
    },
    {
      "href": "https://localhost:5001/api/v2/products",
      "rel": "createProduct",
      "method": "POST"
    },
    {
      "href": "https://localhost:5001/api/v2/products/5",
      "rel": "patchProduct",
      "method": "PATCH"
    },
    {
      "href": "https://localhost:5001/api/v2/products/5",
      "rel": "updateProduct",
      "method": "PUT"
    },
    {
      "href": "https://localhost:5001/api/v2/products/5/image",
      "rel": "patchProductImage",
      "method": "PATCH"
    },
    {
      "href": "https://localhost:5001/api/v2/products/5",
      "rel": "deleteProduct",
      "method": "DELETE"
    },
    {
      "href": "https://localhost:5001/api/v2/products/5/image",
      "rel": "deleteProductImage",
      "method": "DELETE"
    }
  ],
  "id": 5,
  "name": "14kt gold and sterling silver chain necklace",
  "description": null,
  "brand": "Marla Aaron",
  "supplier": null,
  "imageUrl": "https://un1qone.blob.core.windows.net/products/5/82de9f20-d401-4844-a7df-e929ba78bbb0.jpg",
  "serialNumber": null,
  "created": "2020-04-10T11:33:42.084581",
  "modified": null
}
```


Figura 12: Exemplo de resposta com links Hateoas

3.5.5 *Interface uniforme*

A maioria das APIs fornecidas por sistemas baseados na web na época presente permitem apenas GET e POST métodos para executar diferentes ações sobre seus recursos (obter, criar, atualizar ou eliminar um recurso). Por outro lado, uma API *RESTful* deve usar o Métodos HTTP GET, POST, PUT, PATCH e DELETE

corretamente, conforme explicado no capítulo anterior. Assim, na API construída, estes métodos são usados para executar a função para a qual foram projetados.

3.5.6 *Cache*

No mundo de hoje, o armazenamento em cache de dados e respostas é de extrema importância, onde quer que sejam aplicáveis ou possíveis.

Cache é a capacidade de armazenar cópias de dados acedidos com frequência em vários lugares ao longo do caminho de solicitação-resposta.

O armazenamento em cache pode ser implementado no servidor ou no cliente.

Quando um consumidor solicita uma representação de recurso, a solicitação passa por um cache ou uma série de caches (cache local, cache de *proxy* ou *proxy* reverso) em direção ao serviço que hospeda o recurso.

Se qualquer uma das caches ao longo do caminho da solicitação tiver uma nova cópia da representação solicitada, essa cópia será usada para atender à solicitação. Se nenhuma das caches puder satisfazer a solicitação, a solicitação viaja para o serviço (ou servidor de origem, como é formalmente conhecido).

Usando cabeçalhos HTTP, um servidor de origem indica se uma resposta pode ser armazenada em cache e, em caso afirmativo, por quem e por quanto tempo.

As caches ao longo do caminho de resposta podem fazer uma cópia de uma resposta, mas apenas se os metadados de cache permitirem.

Otimizar a rede usando cache melhora a qualidade geral do serviço reduzindo a largura de banda, reduzindo a latência, reduzindo a carga nos servidores e ocultando falhas de rede.

O objetivo é armazenar em cache o máximo de recursos possível para diminuir a carga dos servidores.

No contexto da API desenvolvida, todas as respostas são marcadas como não *cacheable* uma vez que cada resposta depende do utilizador em questão que efetuou o pedido e, do lado do cliente, não temos como efetuar a verificação destas informações protegidas.

Quanto a cache do lado do servidor, foi elaborada uma versão de teste de uma cache para os pedidos GET utilizando Redis. Foi decidido que as listas de recursos como /tags não são armazenáveis em cache, uma vez que tendem a mudar rapidamente. Por outro lado, no caso de um conjunto de recursos não ser tão dinâmico como /products,/tagTypes, /tenants, /validationSessions ou /tagHistory podem ser armazenados em cache e um grande lucro será obtido com esta decisão.

Contudo, neste momento a empresa não considerou prioritário incorporar esta funcionalidade na implementação atual.

3.6 ARQUITETURA APLICAÇÃO

Quando a complexidade de uma aplicação aumenta, a tarefa de construí-la torna-se mais difícil. Essa complexidade pode ser gerida a partir de várias perspectivas diferentes, seja através da utilização de software existente ou do planeamento prévio do sistema desenvolvido. O design deve ser resistente às mudanças que acontecem

ao longo do tempo, mudanças incluindo novos requisitos, mudanças de hardware ou software, ou mesmo da equipa que mantém o suporte da aplicação.

Nesta tese, a arquitetura engloba todas as diferentes partes da aplicação desenvolvida e das relações entre si e com os sistemas externos.

A API REST usa as bibliotecas do ASP.NET Core para lidar com as necessidades comuns de aplicações da Web, enquanto o Entity Framework é usado para aceder à base de dados e fazer o saneamento dos parâmetros. A API foi criada usando um modelo incluído no Software Development Kit (SDK) do ASP.NET Core, ao qual foi retirado o código desnecessário e personalizado o restante para se adequar às necessidades do projeto.

3.6.1 *The program class and the WebHost*

A camada da API possui HTTP *endpoints* chamados de Controllers que podem ser chamados pelo cliente e é hospedada em tempo de execução no ASP.NET Core como um *console application* que é iniciado pela função Main de um Programclass. O ponto de entrada da aplicação é a classe Program mostrada na figura 13.

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
                webBuilder.UseKestrel((options) =>
                {
                    options.AddServerHeader = false;
                });
            });
}
```

Figura 13: A classe Program de uma aplicação ASP.NET Core WebAPI

A classe Program tem uma função Main, que aceita argumentos do ambiente de execução. Na função CreateWebHostBuilder, é criado um *runtime host* para o Startup da aplicação e para a gestão do seu *lifetime* usando as configurações criadas na classe Startup. Estabelece também o conteúdo do *root folder*, carrega as informações do ficheiro appsettings.json que possui as *connection strings* da base de dados e as configurações para a autenticação e *logging*, entre outras.

3.6.2 *Classe Startup e middleware*

No centro da aplicação, a classe que estabelece os serviços necessários e que junta todas as diferentes partes da arquitetura é a classe Startup. Cada pedido que o cliente faz passa pelo ASP.NET Core middleware

pipeline, no qual os pedidos são processados e direcionados para o bloco de código correspondente. O *pipeline* pode ser configurado pelo *developer* na classe Startup, que é chamada quando o programa inicia.

O código para a classe Startup é descrito na figura 14. No seu construtor, a classe recebe o objeto IConfiguration que tem o valor desserializado do ficheiro appsettings.json.

```
public class Startup
{
    public struct ConnectionStrings
    {
        public const string Database = "database";
        public const string IdentityServerDatabase = "IdentityServerDatabase";
        public const string Storage = "storage";
    }

    private const string CorsAllowAllPolicyName = "AllowAllOrigins";

    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    //Add services to the .NET Core IoC container
    public void ConfigureServices(IServiceCollection services)
    ...
    // Configure the HTTP request pipeline
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env, IMapper mapper, IApiVersionDescriptionProvider apiVersionDescripionProvider, AppDbContext context)
    ...
```

Figura 14: A classe Startup e os seus métodos

Além dos argumentos no construtor, a classe possui dois métodos que são chamados quando a aplicação é iniciada: ConfigureServices e Configure. Esses métodos são examinados a seguir.

Enquanto o *host* fornece serviços por meio do construtor da classe Startup, serviços adicionais são configurados com o método ConfigureServices usando o objeto IServiceCollection. O método ConfigureServices é o sítio onde podemos registar as *dependent classes* com o *container* integrado Inversion of Control (IoC). Depois de registar a *dependent class*, ela pode ser usada em qualquer lugar da aplicação. Para isso, basta incluí-la no parâmetro do construtor da classe onde queremos usá-la e o *container* IoC injeta-a automaticamente. A figura 15 mostra uma parte do método ConfigureServices e alguns dos serviços importantes que foram configurados nele.

```
public void ConfigureServices(IServiceCollection services)
{
    var tokenOptionsConfigurationSection = Configuration.GetSection(nameof(Services.TokenOptions));
    var tokenOptions = tokenOptionsConfigurationSection.Get<Services.TokenOptions>();
    services.Configure<Services.TokenOptions>(tokenOptionsConfigurationSection);

    services.AddHttpContextAccessor();

    services.AddCors(options =>
    {
        options.AddPolicy(CorsAllowAllPolicyName, builder =>
        {
            // TOOD: revert this once in production.
            builder
                .AllowAnyOrigin()
                .AllowAnyHeader()
                .AllowAnyMethod();
        });
    });
```

```
services.AddDbContext<AppDbContext>(options =>
{
    options.UseSqlServer(Configuration.GetConnectionString(ConnectionStrings.Database));
});

services.AddDbContext<AppIdentityDbContext>(options =>
{
    options.UseSqlServer(Configuration.GetConnectionString(ConnectionStrings.IdentityServerDatabase));
});

services
    .AddIdentity<ApplicationUser, IdentityRole<long>>(options =>
    {
        options.User.RequireUniqueEmail = true;
    })
    .AddEntityFrameworkStores<AppIdentityDbContext>()
    .AddDefaultTokenProviders();

services.AddAutoMapper((serviceProvider, automapper) =>
{
    automapper.AddCollectionMappers();
    automapper.UseEntityFrameworkCoreModel<AppDbContext>(serviceProvider);
}, typeof(AppDbContext).Assembly);

services.AddMvc(options =>
{
    options.ReturnHttpNotAcceptable = true;
});

services.AddControllers().AddJsonOptions(options =>
{
    options.JsonSerializerOptions.Converters.Add(new JsonStringEnumConverter());
});

services
    .AddAuthentication(options =>
    {
        options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
        options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
    })
    .AddJwtBearer(options =>
    {
        options.Authority = tokenOptions.Authority;
        options.Audience = tokenOptions.Audience;
        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuerSigningKey = false,
            ValidateLifetime = true,
            IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(tokenOptions.Key)),
            ValidateAudience = true,
            ValidAudience = tokenOptions.Audience,
            ValidIssuer = tokenOptions.Issuer
        };
    });

services.AddAuthorization(options =>
{
    options.AddPolicy(AuthService.Policies.BackOfficeView, policy =>
        policy.RequireRole(
            AuthService.Roles.Un1qnxAdmin,
            AuthService.Roles.BackofficeAdmin,
            AuthService.Roles.BackofficeUser,
            AuthService.Roles.BackofficeViewer));
```

Figura 15: Secção do método ConfigureServices

Primeiramente, uma política de *open* CORS foi adicionada porque a aplicação estava em execução apenas na intranet. Finalmente, a conexão com a base de dados da UN1Qnx e com a base de dados para uso do Identity Server que irá ser aprofundado mais à frente, é configurado com o método AddDbContext. O DbContext é explorado mais tarde na tese. Em seguida, é adicionado o Automapper para fazer o mapeamento de objetos, bem como configurada a autenticação através de *tokens* JWT provenientes do Identity Server e adicionadas políticas de acesso através da configuração da autorização para os diferentes intervenientes no sistema(Figuras 29,30,31,32 e 33. Por fim, uma documentação Swagger é adicionada juntamente com o versionamento da API para fornecer aos *developers* ou até mesmo aos utilizadores uma maneira de explorar os terminais que a API fornece.

Um serviço explícito foi criado para cada entidade que exigiu alguma lógica na aplicação. Por exemplo, o StorageService lida com as operações relacionadas com imagens armazenadas num Azure Blob storage e o ValidationService é usado sempre que queremos proceder à validação de uma tag.

A figura 16 mostra a maneira como os serviços criados foram injetados no tempo de execução da aplicação usando as funcionalidades de injeção de dependência integradas fornecidas pela estrutura ASP.NET Core. Os serviços são injetados com o método AddScoped juntamente com filtros criados para serem usados nos métodos dos Controllers. O ciclo de vida de uma classe injetada desta forma dura apenas uma solicitação HTTP, sendo criadas instâncias diferentes para pedidos HTTP diferentes.

```
services.AddScoped<StorageService>();
services.AddScoped<AuthService>();
services.AddScoped<TokenService>();
services.AddScoped<ValidationService>();

services.AddScoped<VerifyProductTenant>();
services.AddScoped<VerifyPostProductTenant>();
services.AddScoped<VerifyTagTenantById>();
services.AddScoped<VerifyTagTenantByTagcode>();
services.AddScoped<VerifyTagPatchById>();
services.AddScoped<VerifyTagPatchByTagcode>();
services.AddScoped<VerifyPostUserTenant>();
services.AddScoped<ValidateMediaTypeAttribute>();

services.AddScoped<HateoasLinksService>();
```

Figura 16: Dependency injection no método ConfigureServices

Como podemos observar na figura 14, o método Configure inclui três parâmetros IApplicationBuilder, IHostingEnvironment e ILoggerFactory. Esses serviços são serviços de estrutura injetados pelo *container* IoC integrado. Em execução, o método ConfigureServices é chamado antes do método Configure. Isso permite registar serviços personalizados com o *container* IoC que podem ser usados no método Configure.

O método Configure estabelece a ordem nas quais as diferentes partes da aplicação correm. A ordem em que o código é executado é importante. Por exemplo, para autenticar um utilizador antes de direcionar a solicitação para o terminal. Neste método podemos também adicionar componentes de *middleware* que serão executados em todos os pedidos. Este *pipeline* que lida com os pedidos é composto por uma série de componentes de

middleware. Cada componente executa operações no HttpContext e invoca o próximo *middleware* no *pipeline* ou encerra a solicitação.

O *pipeline* implementado pode ser visto na figura 17 com os componentes essenciais ao funcionamento da aplicação.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env, IMapper mapper, IApiVersionDescriptionProvider apiVersionDescripionProvider, AppDbContext context)
{
    mapper.ConfigurationProvider.AssertConfigurationIsValid();

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    context.Database.Migrate();

    app.UseCors(CorsAllowAllPolicyName);

    app.UseSwagger(options =>
    {
        options.SerializeAsV2 = true;
    });

    app.UseSwaggerUI(options =>
    {
        options.RoutePrefix = string.Empty;

        foreach (var description in apiVersionDescripionProvider.ApiVersionDescriptions)
        {
            options.SwaggerEndpoint(
                $"/swagger/{description.GroupName}/swagger.json",
                description.GroupName);
        }

    });

    app.UseRouting();

    app.UseMiddleware<SecurityHeadersMiddleware>();
    app.UseAntiXssMiddleware();

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

Figura 17: Método Configure

A destacar deste *pipeline*, são dois *middlewares* bastante importantes do ponto de vista da segurança:

- AntiXssMiddleware: Middleware que tem como objetivo prevenir ataques de Cross-site scripting (XSS). Foi implementado o AntiXssMiddleware de forma a que qualquer script injetado no URL ou no *Body* irá ser devolvido erro.
- SecurityHeadersMiddleware: Middleware que adiciona um conjunto de *headers* às respostas da API entre os quais, *X-Content-Type-Options*,*X-Frame-Options* e *Content-Security-Policy*.

3.6.3 *Routing*

Em aplicações da web, o *routing* é o ato de direcionar solicitações para os terminais corretos. Essas solicitações podem ser feitas a partir da barra de endereço de um navegador da web, uma chamada Asynchronous JavaScript And XML (AJAX) ou em qualquer outra aplicação que use HTTP como protocolo. No ASP.NET Core existem duas maneiras de implementar o *routing*:

• Um *routing* baseado em *middleware* em que a classe de inicialização é usada para definir as rotas e conjuntos de regras que ditam como as solicitações são direcionadas. Assim, é possível manter o código da lógica de *routing* num lugar separado.

• Um *routing* baseado em atributos que usa o recurso de atributos da linguagem C#. Nas classes dos *controllers* , as próprias classes e seus métodos podem ser decorados com um atributo que especifica a rota e o verbo HTTP que a classe ou o método ao qual estaremos vinculados.

Essas duas formas não são mutuamente exclusivas e podem ser usadas simultaneamente. Usando uma combinação de ambas as estratégias, é possível criar regras gerais para *routing* e adicionar exceções via atributos. Ao usar ambas as estratégias, o ASP.NET Core tenta encontrar a melhor rota para cada solicitação HTTP. Nesta tese, apenas o *routing* baseado em atributos foi usado tanto em classes como nos seus métodos.

Os atributos foram usados para definir os verbos HTTP e, em alguns casos, as rotas específicas para certos métodos. No entanto, apenas os atributos HTTPPost ,HTTPGet , HTTPPatch, HTTPPut e HTTPDelete foram usados.

Na figura 18, um *endpoint* HTTP é mostrado. Este é um método numa classe de controlador que retorna um objeto do tipo ActionResult. O atributo é mencionado acima da declaração do método:

```
[HttpGet("{id}")]
public async Task<ActionResult<TagTypeDto>> GetTagType(long id)
{
    var tagType = await db.TagTypes.FindAsync(id);

    if (tagType == null)
    {
        return NotFound();
    }

    var tagTypeDto = mapper.Map<TagTypeDto>(tagType);

    tagTypeDto.Links = hateoasLinksService.CreateLinksForGetTagType(id);

    return Ok(tagTypeDto);
}
```

Figura 18: *Exemplo de um Controller*

ActionResult é a classe base para o resultado de um método de ação que inclui na resposta o *header*: application/json; charset=utf-8. Na foto acima os dois resultados possíveis são NotFound e Ok. Eles retornam as respostas HTTP com seus códigos de status de resposta HTTP específicos.

3.6.4 *Controllers*

Os *controllers* são responsáveis por capturar solicitações HTTP específicas com base no URL e no verbo HTTP usado e são amplamente baseados nas convenções definidas pelo ASP.NET Core.

Esta camada não precisa de saber que tipo de cliente envia as solicitações e, no caso de uma API, não é responsável por renderizar nenhuma visualização. O *controller* recebe solicitações, valida a carga útil e, em seguida, retorna as respostas adequadas com dados no formato JSON.

No ASP.NET Core, todos os controladores herdam de uma classe de controlador base que está incluída nas bibliotecas fornecidas pelo *framework*. Esta classe está ligada ao *pipeline* do ASP.NET Core, que direciona as solicitações para as suas respectivas classes.

As convenções do ASP.NET Core recomendam nomear todos os controladores criados terminando com um sufixo "controller"para que sejam descobertos pelo *pipeline* integrado da estrutura. A primeira parte do nome funciona em conjunto com o sistema de *routing* para definir a classe como um ponto final para uma solicitação. Por exemplo, a classe TagsController é para responder às solicitações feitas ao /tags.

Um total de dez controladores foram criados para a API, a maioria deles representando um modelo de domínio. Todos eles herdam da classe ControllerBase e contêm o atributo ApiController que nos fornece um conjunto de comportomentos específicos para APIs entre os quais, fazer a validação automática do input, essencial para os níveis de segurança que pretendemos.

Modelos DTO

Classes que modelam o *payload* para solicitações de dados e respostas foram criadas. As classes de solicitação foram mapeadas para os parâmetros esperados em solicitações de HTTP e tinham regras de validação usando atributos de propriedade. Classes de resposta foram mapeadas aos dados necessários para a funcionalidade da aplicação cliente. Em geral, estas classes são chamadas de objetos de transferência de dados DTO, pois a sua principal responsabilidade é transportar dados. Um exemplo de um modelo com atributos de validação é mostrado na figura 19. A validação exigida era limitada principalmente a campos obrigatórios. Os requisitos de limites foram ditados do mesmo modo como a base de dados tinha sido configurada.

```
public class TagHistoryDto
{
    public long Id { get; set; }

    public long TagId { get; set; }

    public long UserId { get; set; }

    public string ClientId { get; set; }

    public TagState State { get; set; }

    public DateTime? Modified { get; set; }

    /// <summary>
    /// Hateoas links.
    /// </summary>
    public IEnumerable<Link> Links { get; set; }
}
```

Figura 19: Exemplo de um DTO

Um total de 15 modelos de DTO foram criados para a aplicação, conforme mostrado na figura 20. As classes foram nomeadas usando uma convenção em que o nome da classe informava ao utilizador a finalidade da classe. As classes são usadas para a solicitação HTTP ou a resposta HTTP.

Figura 20: Lista de todos os DTOs cridados

3.6.5 *Camada de acesso de dados*

Entity classes

As classes de entidade são uma parte central do Entity Framework que são usadas pela estrutura para construir um modelo que é usado ao aceder à base de dados. São classes C# que têm propriedades que refletem as tabelas e colunas da base de dados ao usar uma base de dados relacional, respectivamente. Por exemplo, uma classe de user será mapeada para a tabela de users e suas propriedades, como Nome, serão mapeadas para as suas respectivas colunas na tabela de users.

Esses mapeamentos podem ser substituídos, mas as convenções padrão do Entity Framework Core foram usadas na maioria dos casos. As entidades podem ser configuradas usando atributos de anotação de dados nas próprias entidades. A classe BaseDao foi criada para ser usada apenas para unir todas as entidades com os campos Id, Created e Modified.

DbContext

DbContext é o objeto no EntityFramework Core usado para aceder aos dados da base de dados subjacente. Tipicamente as propriedades da classe DbContext são usadas numa classe herdada na qual os acessos à base de dados são declarados nas propriedades do tipo DbSet <T>. Esses DbSets podem então ser consultados usando LINQ que permite escrever *strongly typed queries*. LINQ to SQL ajuda a prevenir ataques de *SQL injection*, porque passa todos os dados para a base de dados através de *SQL parameters*, ou seja, as consultas LINQ não são compostas usando manipulação ou concatenação de strings. O DbContext implementado é mostrado na figura 21:

```
public partial class AppDbContext : DbContext
{
    public AppDbContext() { }

    public AppDbContext(DbContextOptions<AppDbContext> options) : base(options) { }

    public virtual DbSet<AcquisitionPipelineStepType> AcquisitionPipelineStepTypes { get; set; }
    public virtual DbSet<AcquisitionPipelineStep> AcquisitionPipelineSteps { get; set; }
    public virtual DbSet<AcquisitionPipeline> AcquisitionPipelines { get; set; }
    public virtual DbSet<AcquisitionStationType> AcquisitionStationTypes { get; set; }
    public virtual DbSet<AcquisitionStation> AcquisitionStations { get; set; }
    public virtual DbSet<TagType> TagTypes { get; set; }
    public virtual DbSet<Tag> Tags { get; set; }
    public virtual DbSet<TagHistory> TagsHistory { get; set; }
    public virtual DbSet<User> Users { get; set; }
    public virtual DbSet<ValidationSession> ValidationSessions { get; set; }
    public virtual DbSet<Validation> Validations { get; set; }
    public virtual DbSet<Product> Products { get; set; }
    public virtual DbSet<Tenant> Tenants { get; set; }
    public virtual DbSet<UserRole> UserRoles { get; set; }

    public virtual DbSet<DigitalSignatureKey> DigitalSignatureKeys { get; set; }
```

Figura 21: AppDbContext implementado

3.7 FUNCIONALIDADES

Seguidamente são apresentadas as funcionalidades da Web API, cada uma dotada de uma breve explicação, do processo da sua implementação e, se pertinente, uma explicação da sua utilização face a outras opções existentes.

Tal como referido anteriormente, para fazer a autenticação de um cliente, foram consideradas diversas maneiras de abordar este problema. A autenticação é necessária quando uma aplicação precisa de saber a identidade do utilizador atual. Normalmente, as aplicações gerem dados em nome desses utilizadores e precisam de ter certeza que esse utilizador só pode aceder a dados para o qual tem permissão.

Os protocolos de autenticação mais comuns são SAML2p, WS-Federation e OpenID Connect. SAML2p é o mais popular e mais amplamente implantado.

OpenID Connect é o mais novo dos três, mas é considerado o futuro porque tem o maior potencial para aplicações modernas. Foi construído para cenários de aplicativos móveis desde o início e é projetado para ser API *friendly*.

3.7.1 *Acesso API*

As aplicações têm duas maneiras fundamentais através das quais comunicam com APIs - usando a identidade da aplicação ou delegando a identidade do utilizador. Às vezes, os dois métodos precisam de ser combinados.

OAuth2 é um protocolo que permite às aplicações solicitar *tokens* de acesso de um serviço de *token* de segurança e usá-los para comunicarem com APIs. Essa delegação reduz a complexidade tanto do lado dos clientes quanto nas APIs, uma vez que a autenticação e a autorização podem ser centralizadas.

O OpenID Connect e o OAuth 2.0 são muito semelhantes - na verdade, o OpenID Connect é uma extensão do OAuth 2.0. As duas questões fundamentais de segurança, autenticação e acesso à API, são combinadas num único protocolo - muitas vezes com apenas uma única viagem de ida e volta para o serviço de *token* de segurança. Deste modo, acredita-se que a combinação de OpenID Connect e OAuth 2.0 é a melhor abordagem para proteger aplicações modernas para um futuro próximo.

IdentityServer4 é uma implementação desses dois protocolos e é altamente otimizado para resolver os problemas de segurança típicos das aplicações móveis, nativas e da web de hoje.

IdentityServer é um *middleware* que adiciona os pontos de extremidade OpenID Connect e OAuth 2.0 compatíveis com as especificações a uma aplicação ASP.NET Core.

Normalmente, constrói-se (ou reutiliza-se) uma aplicação que contém uma página de *login* e *logout* e o *middleware* IdentityServer adiciona os cabeçalhos de protocolo necessários para que as aplicações clientes possam comunicar com ele usando esses protocolos padrão.

A parte mais importante é o facto de muitos aspectos do IdentityServer poderem ser personalizados para atender às necessidades. Uma vez que IdentityServer é uma estrutura e não um produto em série ou um SaaS, podemos escrever código para adaptar o sistema da maneira que faz mais sentido para o cenário em questão.

3.7.2 *Fluxo da API*

A utilização da Web API é descrita neste subcapítulo.

Para que um utilizador possa aceder a qualquer recurso da API, necessita de um *token* de acesso. Este *token* é concedido pelo servidor de autenticação (IdentityServer), caso as credenciais enviadas pelo utilizador sejam válidas, permite fazer pedidos à API. Existem dois *usecases*, um para clientes que desejem integrar a API nos seus SDKs e outro para clientes que desejem utilizar o *backoffice* da UN1Qnx.

Para auxiliar a compreensão do fluxo de atividades entre as diferentes entidades presentes na utilização da Web API por um utilizador ou cliente com credenciais válidas, apresentam-se os seguintes diagramas de sequência com os diferentes métodos OAuth 2.0 usados:

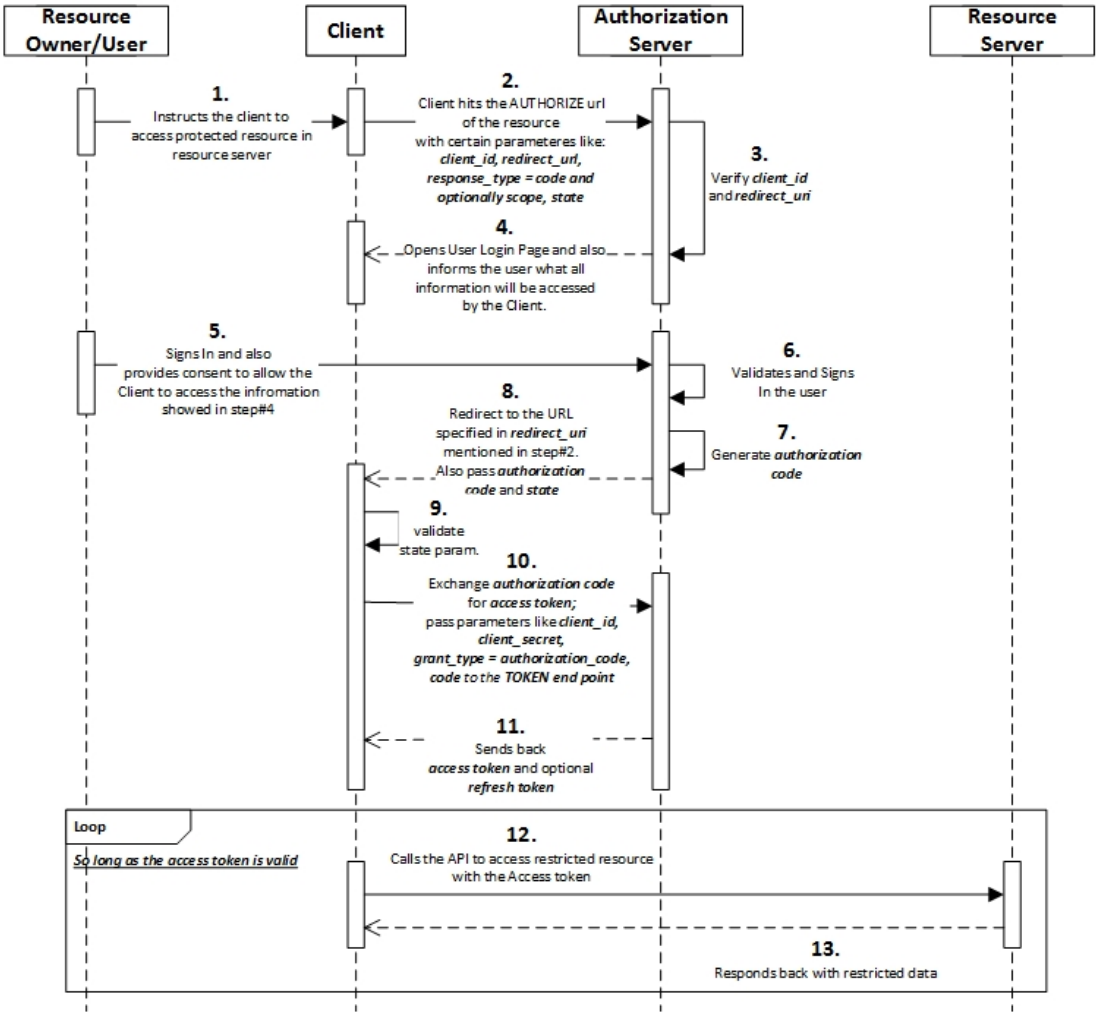


Figura 22: Diagrama de sequência de Authorization Code Flow (Fonte: Md (a))

No caso de utilizadores do *backoffice* da UN1Qnx foi utilizado *Authorization Code Flow* .

Na figura acima *Resource Owner* representa o utilizador do *backoffice*, *Resource Server* representa o servidor da API, *Client* representa o *backoffice* e *Authorization Server* representa o IdentityServer.

O utilizador faz um pedido normal no *backoffice* da UN1Qnx para aceder a um determinado recurso. Depois disso, o *backoffice* procede ao envio de um pedido para o *endpoint* de autorização do servidor OAuth (IdentityServer) que apresenta ao utilizador a página de login. O utilizador autentica-se usando uma das opções de *login* configuradas. O servidor de autorização redireciona o utilizador de volta para a aplicação *backoffice* com um código de autorização, que é válido apenas para uma utilização. A aplicação do *backoffice* envia esse código para o IdentityServer juntamente com o seu ClientID e ClientSecret. O IdentityServer verifica o código, o ClientID e Client Secret e devolve o *access token* que o *backoffice* usará para fazer as operações através da API com as informações relacionadas com o utilizador.

Neste caso, o *token* é guardado internamente no *backoffice* para ser anexado em todos os pedidos futuros. O utilizador pode, por exemplo, fazer um pedido para obter informações acerca de uma tag, para tal, o website anexa o *token* ao pedido, que é depois processado pela Web API que verifica a validade do *token* e, caso seja válido, faz um pedido à API para obter o objeto desejado, que é então retornado ao utilizador. Depois de obter o objeto desejado, o utilizador pode, por exemplo, editá-lo através de um novo pedido e, tal como para o pedido de leitura do objeto, é feito todo um processo idêntico em que o *token* é validado e uma resposta de sucesso é devolvida.

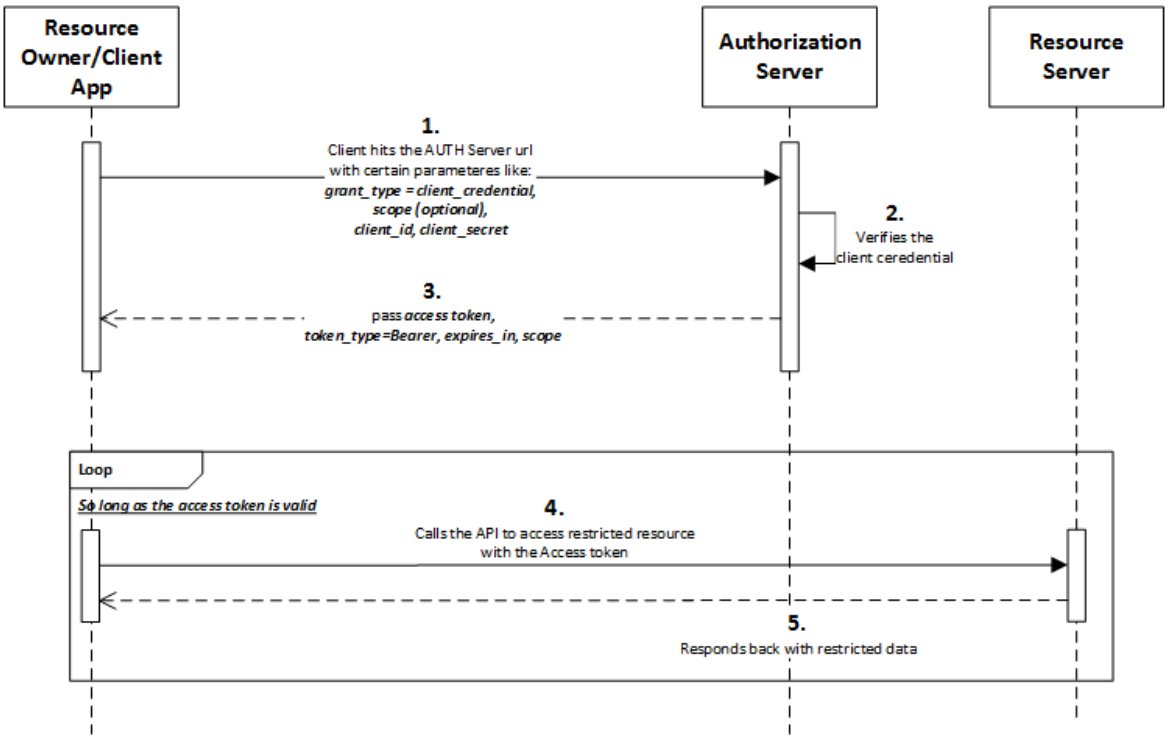


Figura 23: Diagrama de sequência de Client Credential Flow (Fonte: Md (b))

Neste caso foi utilizado *Client Credentials grant* que permite às aplicações de clientes solicitar *tokens* de acesso para aceder aos seus próprios recursos e não apenas em nome de algum utilizador como anteriormente.

A forma como isto funciona é a seguinte: uma aplicação cliente da API envia uma solicitação ao IdentityServer solicitando um *token* de acesso à API fornecendo o seu ClientId e ClientSecret previamente definidos. O servidor de autorização valida as credencias e devolve um *access token*. Esse *token* é então anexado ao pedido sempre que o cliente quiser usar a API.

3.7.3 *Resposta assinada digitalmente*

O objetivo de todo o processo UN1Qnx, tal como foi explicado anteriormente, é o de garantir a autenticidade de um produto através das tags. Posto isto, o resultado de uma sessão de validação de uma tag é assinado digitalmente através de criptografia de chave pública.

Criptografia de chave pública, também conhecido como criptografia assimétrica, é um sistema que usa pares de chaves para criptografar e autenticar informações. Uma chave do par é uma chave pública que pode, como o nome sugere, ser amplamente distribuída sem afetar a segurança.

Assinatura digital é um *hash* criado usando os dados de uma mensagem. Quando essa mensagem é enviada, a assinatura pode ser verificada pelo destinatário usando a chave pública do remetente para autenticar a origem da mensagem e garantir que ela não tenha sido adulterada.

As etapas deste processo são as seguintes:

1. Aplica-se uma função de *hash* sobre os dados que queremos transmitir de forma a criar um *hash value*. Este *hash value* irá funcionar como prova de que nenhumas alterações foram feitas aos dados;
2. Este *hash value* é encriptado com a chave privada do servidor (API);
3. A assinatura digital é concluída neste passo;
4. A assinatura é anexada com os dados e são enviados para o cliente;
5. O cliente usa a chave pública para desencriptar o *hash value*;
6. O cliente calcula o *hash* dos dados recebidos;
7. Finalmente o cliente pode comparar os dois valores de forma a chegar à conclusão se a integridade dos dados foi comprometida ou não.

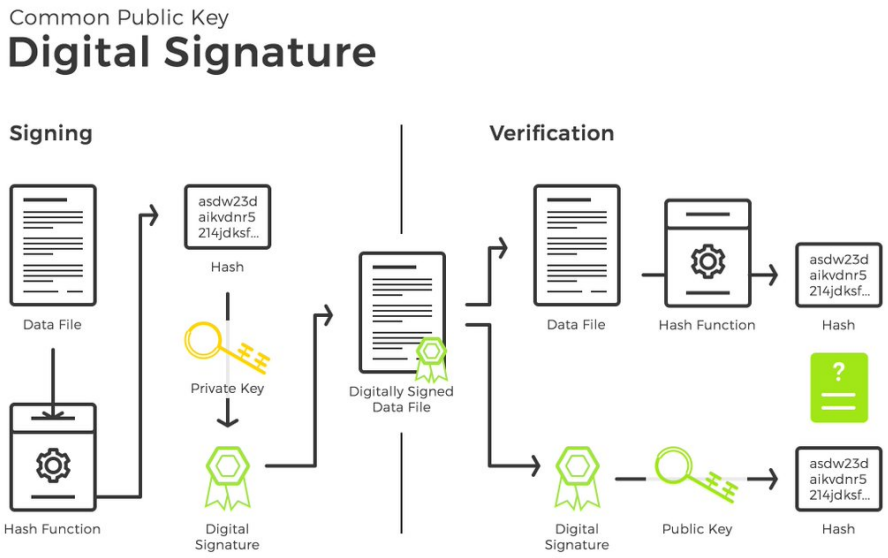


Figura 24: Exemplificação da Assinatura digital (Fonte: Pahwa)

Neste momento, estamos a utilizar Rivest-Shamir-Adleman (RSA) como algoritmo de sistemas de criptografia de chave pública para a transmissão segura de dados e SHA-256 como função *hash* criptográfica. Para os clientes obterem a chave pública, existe um *endpoint* no Controller Tenants que devolve as informações necessárias para o cliente comprovar a autenticidade do resultado da validação.

3.7.4 *Arquitetura final*

Na figura 25, está representada a nova arquitetura implementada, que difere da anterior unicamente na questão da incorporação do IdentityServer como servidor de autenticação do sistema. Podemos também analisar os diagramas das base de dados criadas nas figuras 34 e 35.

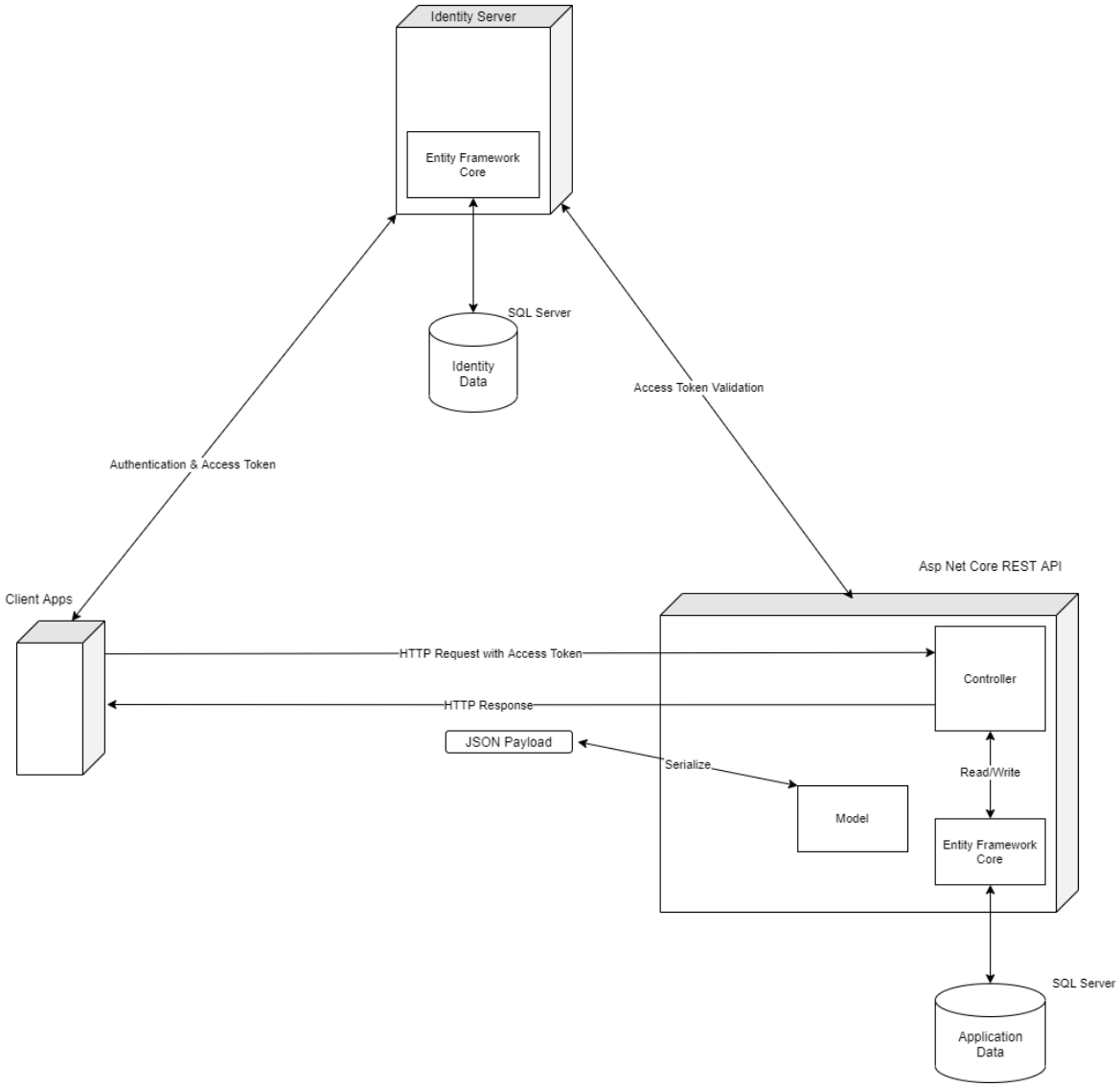


Figura 25: Arquitetura final

3.7.5 *Documentação*

Uma boa documentação é sempre importante. Um dos fatores que mais motivou a procura de criar uma boa documentação, foi a existência de entidades externas que fazem uso da API desenvolvida nas suas aplicações.

Para a documentação da Web API, foi utilizada a *framework* Swagger, capaz de gerar automaticamente a documentação a partir dos métodos existentes em cada controlador da Web API (Figura 36). Esta *framework* permite que o consumidor da Web API possa facilmente visualizar os diferentes controladores, métodos e atributos. No caso dos atributos é também exposto o seu tipo, e em caso de ser um objeto personalizado, é apresentado um valor esperado, em formato JSON (Figura 26).

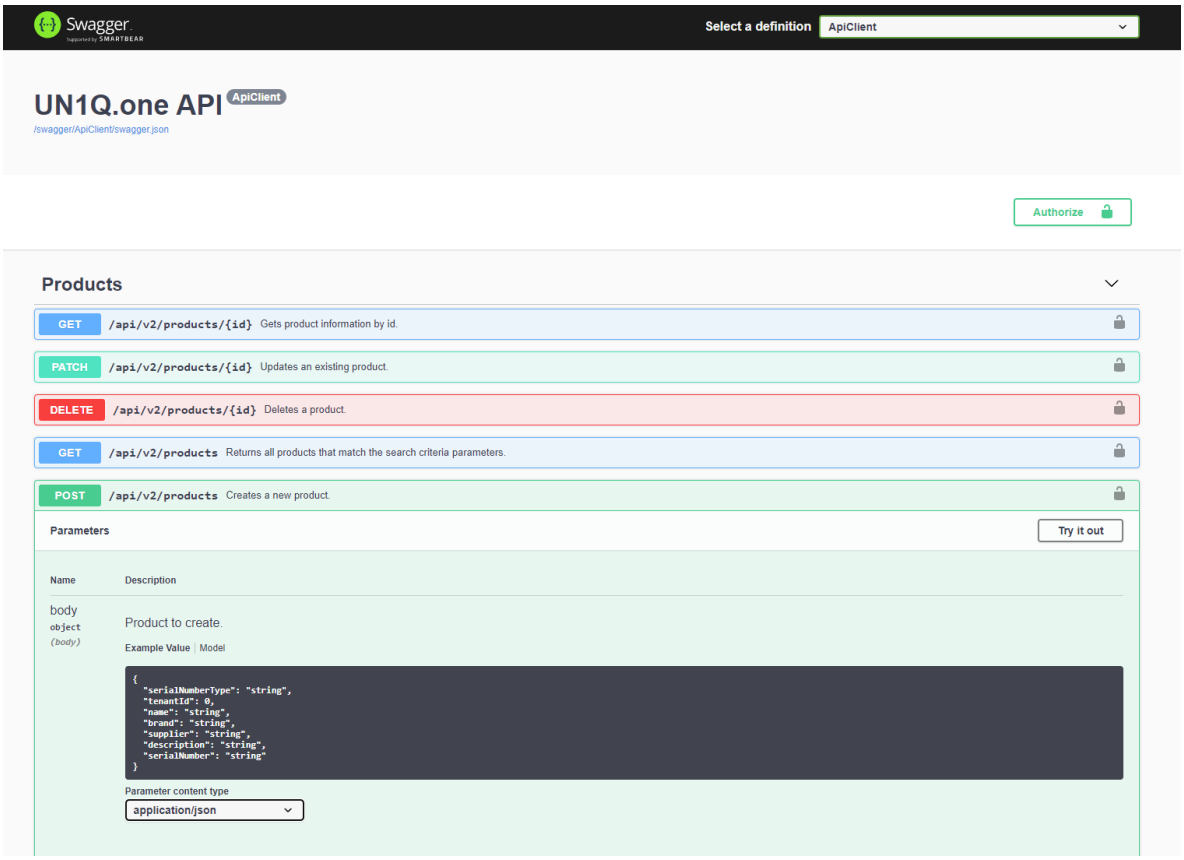


Figura 26: Documentação Swagger

4

RESULTADOS E DISCUSSÃO

Neste capítulo são apresentados alguns testes funcionais ao sistema, bem como interpretados os resultados obtidos dos diferentes componentes do trabalho prático desta dissertação e avaliar se estes vão de encontro com o pretendido.

4.1 TESTES

4.1.1 *Integration tests*

Foram elaborados um conjunto de testes de integração para avaliar o funcionamento da aplicação.

Os testes de integração avaliam os componentes de uma aplicação num nível mais amplo do que os testes de unidade. Enquanto que os testes de unidade são usados para testar componentes de software isolados, como métodos de classes individuais, os testes de integração confirmam que dois ou mais componentes da aplicação funcionam em conjunto de modo a produzir o resultado esperado (Microsoft, c).

Ou seja, através dos testes de integração iremos simular, ou melhor, consumir a API e verificar se tudo ocorreu conforme o esperado.

A seleção da *framework* de testes foi limitada a algumas diferentes estruturas que estavam disponíveis na plataforma .NET Core. As três principais foram MSTest, NUnit e XUnit. Para a tese, a ferramenta escolhida foi o XUnit e a decisão foi tomada porque, ao contrário do MSTest e do NUnit, o XUnit segue as convenções de teste e usa uma sintaxe semelhante às estruturas de teste disponíveis noutras linguagens.

Os testes de integração seguem uma sequência de eventos que incluem as etapas padrão de teste Arrange, Act e Assert (AAA):

- O *host* do System Under Test (SUT) é configurado;
- Um cliente de servidor de teste é criado para enviar solicitações à aplicação;
- A etapa de teste Arrange é executada: a aplicação de teste prepara uma solicitação;
- A etapa de teste Act é executada: o cliente envia a solicitação e recebe a resposta;
- A etapa de teste Assert é executada: a resposta real é validada como aprovada ou reprovada com base no comportamento esperado;

- O processo continua até que todos os testes sejam executados;
- Os resultados do teste são divulgados.

O *host* de teste é configurado de forma diferente do *host* da aplicação normal para as execuções de teste. Por exemplo, no projeto desenvolvido uma base de dados diferente e configurações de aplicações diferentes são usados para os testes.

Foram realizados testes para todos os *endpoints* e dinâmicas da API criada, o que resultou num total de 69 testes:

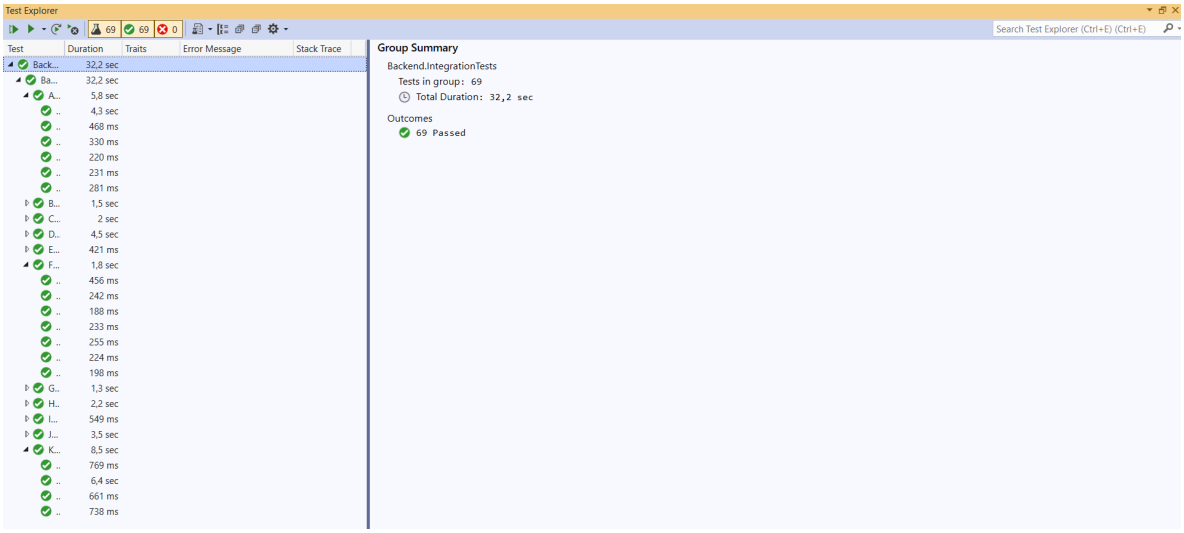


Figura 27: Resultado dos Testes de Integração

O tempo de resposta da aplicação foi testado num computador com Windows instalado. Note-se que o servidor que futuramente vai hospedar a versão final da aplicação dispõe de hardware mais rápido. Apesar das limitações de hardware durante a fase de testes, verificou-se que todos os pedidos que utilizavam apenas a lógica interna aplicação davam a resposta entre 200 a 600 milissegundos – o tempo de resposta da API foi testado utilizando o software Postman. Como já foi dito anteriormente, o único método implementado que utiliza serviços externos é o POST /validationSessions/Id/validations uma vez que efetua a submissão de uma imagem para validação no contexto de uma determinada sessão de validação. É importante referir que uma sessão de validação pode conter mais do que uma validação e que para além disso, o tempo de resposta está também dependente da infraestrutura externa da *cloud* e dos respetivos recursos computacionais associados. Por essas razões, neste caso foram obtidos tempos de resposta bastante dispersos e bastante mais longos do que os anteriores. Após discussão com os elementos da UN1Qnx , considerou-se que os tempos de resposta obtidos são satisfatórios. Além disso, estes serão inferiores em ambiente de produção. Posto isto, todos estes valores enquadram-se nos requisitos impostos pela empresa UN1Qnx .

4.1.2 *Análise de segurança e de vulnerabilidades*

Com o intuito de encontrar vulnerabilidades e falhas de segurança na API, foi utilizada a ferramenta ZAP, mais concretamente, ZAP Desktop UI.

Zed Attack Proxy (ZAP) é uma ferramenta gratuita de teste de penetração mantida pela Open Web Application Security Project (OWASP). O ZAP foi projetado especificamente para testar aplicações da web e é bastante flexível e extensível.

ZAP é o que é conhecido como *man-in-the-middle proxy*. Atua entre o navegador de teste e a aplicação web para que possa intercetar e inspecionar mensagens enviadas entre o navegador e a aplicação web, modificar o conteúdo se necessário e, em seguida, encaminhar esses pacotes para o destino.

Explicado de forma mais simplificada, o ZAP envia solicitações à aplicação que imitam os ataques que um invasor mal intencionado usaria. Com base na resposta recebida da aplicação, o ZAP destaca quaisquer vulnerabilidades em potencial.

O ZAP pode ser considerado como a ferramenta mais cotada dentro das ferramentas *open source* de testes de seguranças de aplicações, e, uma das razões que a diferencia é a sua capacidade de automatização. O ZAP possui uma panóplia de testes automáticos, alguns dos utlizados neste projeto foram: Active Scan, Spider e Fuzzing.

De seguida, apresenta-se alguns dos resultados obtidos:

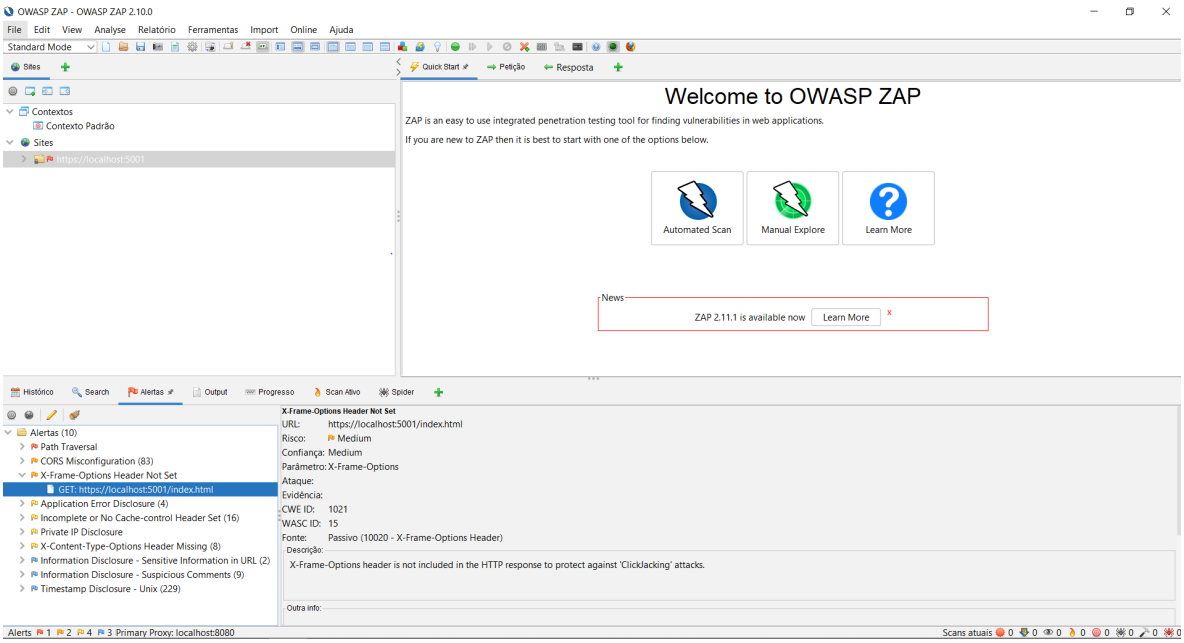


Figura 28: Exemplo de resultados do ZAP

O uso desta ferramenta revelou-se vital para o desenvolvimento da aplicação visto que permitiu alertar para algumas vulnerabilidades existentes aquando das fases iniciais de desenvolvimento.

4.2 RESULTADOS

O uso de APIs está a aumentar e a fortalecer as empresas de uma forma cada vez mais dinâmica. No entanto, apesar das vantagens da utilização desses recursos, as organizações precisam de estar cientes das potenciais brechas de segurança, ou seja, garantir que a segurança é a prioridade número um.

Uma API insegura ou até apenas um *endpoint* da aplicação pode servir como uma porta de entrada para os invasores atacarem o sistema. O ponto diferencial desta dissertação é contribuir para um enriquecimento no desenvolvimento de APIs, com foco nas APIs REST, no que toca a princípios de design e de melhores práticas de segurança, com o intuito máximo na proteção das APIs. De seguida apresenta-se o culminar de toda a investigação realizada juntamente com a aprendizagem obtida com o decorrer do desenvolvimento do projeto:

4.2.1 *Design*

Usar naming standards

- Todas as partes do URI, incluindo os nomes dos recursos, devem ser escritas em letras minúsculas;
- Não devemos usar verbos nos *endpoints*. Em vez disso, devemos usar os substantivos que representam a entidade que é o ponto de extremidade que estamos a obter ou a alterar como o nome do caminho. Isso deve-se ao facto do método de solicitação HTTP já possuir o verbo. Ter verbos nos caminhos de *endpoint* da API não é útil e torna-o desnecessariamente longo, pois não transmite nenhuma informação nova;
- O nome dos recursos deve estar no plural;
- Devemos usar *camelCase* ou *snake_case* para o nome dos *query parameters* e para o nome dos campos no *body* dos pedidos e das respostas.

Usar nesting lógico nos endpoints

Ao projetar *endpoints*, faz sentido agrupar aqueles que contêm informações associadas. Ou seja, se um objeto pode conter outro objeto, deve-se projetar o ponto de extremidade para refletir isso. Essa é uma boa prática, independentemente dos dados estarem estruturados dessa forma na base de dados ou não.

Por exemplo, se quisermos que um *endpoint* obtenha os tipos de um artigo de tags, devemos anexar o caminho /types ao final do caminho /tags.

Fornecer links HATEOAS corretamente

Para ajudar a orientar os utilizadores, links relacionais da API, devem ser fornecidos. Estes links permitem a navegação na API, informando os utilizadores sobre o que eles podem fazer a seguir.

Um *array* de links referentes a recursos no sistema relacionados com o recurso atual deve ser fornecido. Cada objeto deste *array* deve conter os seguintes elementos:

- href - *string* contendo o URL para o recurso;
- rel - *string* textual que descreve o que esta entidade é;
- method - o método HTTP que deve ser usado neste recurso.

Exemplo:

```
"links": [
{
    "href": "https://localhost:5001/api/v2/products/5",
    "rel": "self",
    "method": "GET"
}
```

Permitir filtering, sorting, e pagination

As base de dados que suportam as API REST podem ficar muito grandes. Às vezes, há tantos dados que não deveriam ser devolvidos todos de uma só vez, porque é muito lento ou porque pode até mesmo derrubar o sistema. Portanto, precisamos de maneiras de filtrar os itens.

Também precisamos de maneiras de paginar os dados para que retornemos apenas alguns resultados por vez. Não queremos ocupar recursos por muito tempo tentando obter todos os dados solicitados de uma só vez.

A filtragem e a paginação aumentam o desempenho, reduzindo o uso de recursos do servidor. À medida que mais dados se acumulam na base de dados, mais importantes esses recursos se tornam.

Usar cache para melhorar a performance

Podemos adicionar cache para retornar os dados da cache da memória local ao invés de consultar a base de dados sempre que quisermos recuperar alguns dados solicitados pelos utilizadores. A vantagem do armazenamento em cache é que os utilizadores podem obter os dados mais rapidamente. No entanto, temos de gerir cuidadosamente a implementação da mesma pelo facto dos dados que os utilizadores obtêm poderem estar desatualizados.

Usar versionamento API

Devemos ter versões diferentes da API se estivermos a fazer alterações nelas que possam interromper os clientes. Dessa forma, podemos eliminar gradualmente os *endpoints* antigos em vez de forçar todos a migrar para a nova API ao mesmo tempo. O *endpoint* v1 pode permanecer ativo para pessoas que não querem mudar, enquanto o v2, com novos recursos, pode servir para aqueles que estão prontos para atualizar. Isso é especialmente importante se a API for pública. Devemos criar versionamento para não quebrar as aplicações de terceiros que usam as nossas APIs.

Documentação

O objetivo é apresentar uma documentação que permita que qualquer pessoa, que ainda não tenha pleno conhecimento do domínio da aplicação, desenvolva soluções de maneira rápida, eficaz e autónoma a partir da mesma.

A documentação de uma API é uma entrega técnica de conteúdo, contendo instruções sobre como usar e integrar efetivamente a solução.

Sem documentar uma API de maneira adequada, a pessoa que vai utilizá-la perde tempo a tentar desvendar o seu funcionamento, o que cria barreiras para a adoção do serviço em questão.

A API *documentation* deve ser bem completa e seu foco principal deve ser nos recursos e *endpoints* disponíveis.

Usar uma API Specification Framework

As APIs *frameworks* são uma tentativa de padronizar os processos de desenvolvimento em todos os setores. Normalmente, consistem num arsenal de ferramentas que cobrem todo o ciclo de vida de desenvolvimento, do conceito à produção. Embora seja verdade que aderir a uma especificação como OpenAPI/Swagger durante o desenvolvimento de uma API seja opcional, através destas conseguimos fornecer uma melhor interoperabilidade de ferramentas. A automação é sempre visto como uma vantagem, ter a capacidade de gerar documentação, SDKs e pontos de interação UI sempre que o código é alterado é muito útil. Por tudo isto, são opções muito sólidas que compensam imediatamente, uma vez que descrevem corretamente as APIs.

Tratar erros de forma correta e usar códigos de erro standard

Para evitar a possível confusão dos utilizadores da API quando ocorre um erro, devemos tratar os erros de maneira adequada e genérica não revelando demasiados detalhes da falha ocorrida. Simultaneamente devemos devolver os códigos de resposta HTTP que indicam o tipo de erro ocorrido.

A configuração correta de *timeouts* é também muito importante para o tratamento de erros e *troubleshooting* uma vez que podem provocar vários problemas e interrupções na API. Uma configuração correta de *timeout* não deve permitir que a conexão com o cliente falhe antes do sistema devolver a resposta.

4.2.2 *Segurança*

Restringir métodos HTTP

- Criar uma lista com apenas os métodos HTTP que são permitidos;
- Rejeitar todos os pedidos de métodos HTTP que não pertençam à lista com um código de resposta HTTP adequado;
- Assegurar que o utilizador está autorizado a usar o método HTTP no recurso em questão.

Validação de Input

As APIs são projetadas para acesso automatizado sem a necessidade de interação por parte do utilizador, por isso é especialmente importante garantir que todas as entradas sejam válidas e de acordo com o esperado. Todas as requisições que não estejam em conformidade com a especificação da API devem ser rejeitadas. As diretrizes típicas de melhores práticas para validação de input são:

- Considerar todos os parâmetros, objetos e dados de input como não confiáveis;
- Verificar o *request size*, o *content-length* e o *content-type*;
- Usar *strong typing* nos parâmetros da API;
- Usar *logs* para registar *input validation failures* de modo a avaliar possíveis ocorrências de ataques;
- Restringir o input de *strings* sempre que possível com *regexps*;
- Usar *frameworks* ou bibiotecas de validação associados à linguagem de programação.

Validar content-types

O corpo de um pedido ou de uma resposta deve corresponder ao tipo de conteúdo pretendido no cabeçalho. Os serviços REST devem definir precisamente os tipos de conteúdo permitidos e rejeitar as requisições que não tenham as declarações corretas nos seus cabeçalhos HTTP. Isso significa especificar cuidadosamente os tipos permitidos nos cabeçalhos *Content-Type* e *Accept*.

Para além disso, é comum nos serviços REST permitir-se múltiplos tipos de resposta (por exemplo, *application/xml* ou *application/json*), e o cliente deve especificar qual prefere através do cabeçalho *Accept* ao fazer o pedido. Caso o cliente especifique um tipo não permitido, deve-se rejeitar o pedido.

Usar Security Headers

Para garantir que o conteúdo de um determinado recurso seja interpretado corretamente pelo navegador, podem ser definidos cabeçalhos de segurança HTTP adicionais para restringir ainda mais o tipo e o âmbito das requisições. Alguns destes exemplos são:

- *X-Content-Type-Options*: *nosniff* para prevenir ataques XSS baseados em MIME *sniffing* ;
- *X-Frame-Options*: *deny* para prevenir tentativas de *clickjacking* em navegadores mais antigos.
- *Content-Security-Policy*: *frame-ancestors 'none'* é uma *browser security policy* mais recente do que o header anterior. Mesmo que as respostas da API não sejam renderizadas em *frames*, nada impede que isso aconteça. Para garantir que as respostas da API não sejam vulneráveis a ataques obscuros, é recomendável definir ambos os headers em cada resposta da API.

Para além disto, devemos também remover *fingerprinting headers* como *X-Powered-By, Server, X-AspNet-Version*, etc.

Ter cuidado com o uso de CORS

Cross-Origin Resource Sharing (CORS) é uma especificação do W3C que, quando implementado pelo navegador, permite que um site aceda a recursos de outro *website* mesmo estando em domínios diferentes. Se o serviço não suportar *cross-domain calls*, então deve-se desativar o cabeçalho CORS. Caso o mecanismo CORS seja necessário, deve ser especificado concretamente quais as origens permitidas.

Ter cuidado com a partilha de informação sensível nos pedidos HTTP

As chamadas de APIs geralmente incluem credenciais, chaves de API, *tokens* de sessão e outras informações confidenciais. Se incluídos diretamente nos URLs, estes detalhes podem ser armazenados em *logs* dos servidores da web e descobertos se os *logs* forem acedidos por *hackers*. Para evitar a divulgação de informações confidenciais, os serviços da Web REST devem sempre enviá-las no cabeçalho do pedido HTTP ou no corpo do pedido (para pedidos POST e PUT).

Usar HTTPS

Os serviços REST seguros devem fornecer apenas *endpoints* HTTPS de forma a proteger contra *man in the middle attacks*, *replay attacks* e *snooping*.

Estabelecer Acess Control

Os serviços REST privados devem executar o controlo de acesso em cada *endpoint* da API a fim de minimizar a latência, melhorar o desacoplamento entre os serviços e evitar acessos indevidos à API. Para além disso, a autenticação do utilizador deve ser centralizada num Identity provider (IdP), que emite *tokens* de acesso.

Limitar o número de requests - Throttling API

Um dos ataques mais comuns na Internet é um ataque de negação de serviço (DoS), que envolve o envio de um grande número de solicitações a um servidor. O servidor tenta responder a cada pedido e eventualmente fica sem recursos. De forma a mitigar ataques DoS podemos usar *rate-limit throttle* que limita o número de pedidos durante um determinado período de tempo ou usar *IP–based throttling* em que limitamos ou bloqueamos os pedidos de um determinado endereço IP.

Usar Auditing e Logging

Auditing nunca deve ser ignorado, deve ser usado como uma técnica para detectar de forma proativa e prevenir ataques. O *logging* deve ser sistemático, independente e resistente a ataques de injeção de *logs*.

Testar a API

Devem ser realizados testes explorando todo o tipo de utilização da API e não apenas considerando a forma como prevemos que deve ser utilizada com o intuito de descobrir potenciais erros e falhas no sistema.

Não criar soluções próprias em assuntos de segurança

No que toca às questões de autenticação, autorização ou geração de *tokens* nunca devemos tentar criar a nossa própria solução. Existe já um grande número de protocolos e soluções testadas que podemos usar e adaptar aos nossos projetos consoante a linguagem de programação ou *framework* que estejamos a usar.

4.3 DISCUSSÃO

No contexto de uma dissertação, o importante é poder contribuir com conhecimento atual num domínio particular, neste caso, para a literatura sobre o desenvolvimento de APIs seguras. Posto isto, o trabalho elaborado poderá servir de guia para auxiliar na construção de uma API REST segura uma vez que fornece um conjunto de princípios que ajudam a complementar o estilo arquitetural apresentado por Fielding e porque expõe as principais vulnerabilidades a ter em conta na criação de uma API REST bem como uma lista de requisitos de segurança que auxilia na prevenção das mesmas.

A API criada neste projeto é o exemplo prático da aplicação de todas essas informações recolhidas. Podemos afirmar que é segura uma vez que foi submetida a um conjunto de testes de segurança bastante rigorosos e destaca-se de outras soluções uma vez que foi construída tendo em conta todas as possíveis falhas e potenciais ameaças existentes neste contexto usando o pensamento crítico sobre diversas práticas de segurança e aplicando-as para evitar violações.

Um fator que condicionou o desenvolvimento da API foi o planeamento inicial do projeto, uma vez que os requisitos foram inicialmente definidos de forma bastante genérica, sendo melhor especificados ao longo do desenvolvimento da API o que levou a constantes alterações a nível de programação. Apesar disso, foram implementadas todas as funcionalidades mais importantes e, em alguns casos, apesar de não terem sido incorporadas, foram apresentadas e discutidas novas soluções, como é o caso do uso de cache e do *throttle* da API que na demo criada faz uso do *package* AspNetCoreRateLimit que permite limitar o número de pedidos consoante o IP ou ID do cliente. O resultado final alcançado foi de encontro ao esperado mesmo sendo utilizadas ferramentas com um elevado grau de complexidade graças às suas medidas de segurança e a um design simples, adaptável e contemporâneo. Para além disso, todos os aspectos da solução foram criados com o objetivo de manter o controlo absoluto internamente e simultaneamente fornecer aos *developers* externos flexibilidade. Deste modo, em termos de objetivos propostos podemos concluir que foram atingidos. No término deste projeto foi possível definir uma nova estrutura sólida, escalável e vigente e expor uma solução adequada e viável.

5

CONCLUSÕES E TRABALHO FUTURO

Neste capítulo analisa-se o trabalho desenvolvido durante este projeto de investigação. As questões de investigação levantadas no início desta dissertação são agora diretamente respondidas e os esforços futuros que poderão ou deverão ser assumidos de forma a melhorar a solução são abordados também.

5.1 CONCLUSÕES

Na fase de conclusão de um projeto de dissertação, é necessário cogitar sobre a questão de investigação previamente definida, com o propósito de dar resposta à mesma de uma forma assertiva e aceitável.

Questão de investigação: Como desenvolver uma API segura?

APIs REST são um dos tipos mais comuns de serviços da Web disponíveis no mundo atual. Portanto, é muito importante projetar APIs REST corretamente para que não tenhamos problemas no futuro. Após toda a pesquisa realizada no Estado de Arte foi possível reunir um conjunto de métricas importantes de seguir ao elaborar uma API REST tendo para tal tido principalmente em consideração a segurança, o desempenho e a facilidade de uso para os consumidores da API.

Primeiramente, tal como foi referido em capítulos anteriores, uma API deve seguir todos os princípios REST apresentados por Fielding.

Para além disso, de uma forma geral o design eficaz de uma API terá as seguintes características:

- Fácil de ler e trabalhar: Uma API bem projetada será fácil de trabalhar e os seus recursos e operações associadas devem poder ser memorizados rapidamente por *developers* que trabalham com ela constantemente. Para além disso, a implementação e integração com uma API com um bom design será um processo direto e a probabilidade de um uso indevido das mesmas deve ser baixo. Deve ter ainda um feedback informativo e não impor diretrizes rígidas ao consumidor final da API.

- Completo e conciso: Uma API bem estruturada possibilitará que os *developers* criem aplicações completas a partir das APIs com base nos dados que são expostos. Aliás, sabemos que a busca pela integridade requer esforço e engenho e, por isso, os designers e *developers* de APIs devem construía-las incrementalmente com base nas APIs já existentes. É uma noção que qualquer engenheiro ou empresa deverá ter sempre presente na concepção de uma API .

Finalmente, no que toca à segurança, esta deverá ser sempre a prioridade número um ao delinear uma API. Neste documento são apontadas as principais vulnerabilidades que afetam as APIs na atualidade e possíveis métodos para as prevenir e garantir a segurança do sistema. No entanto, *developers* costumam ter uma mentalidade chamada de *feature–driven*, onde a funcionalidade tem precedência sobre a segurança. Porém, no panorama de segurança vigente, vulnerabilidades e ameaças são cada vez mais crescentes e com consequências cada vez maiores, portanto devemos contrariar esta tendência dando primazia absoluta à temática da segurança .

5.2 TRABALHO FUTURO

O desenvolvimento de APIs está em constante evolução e todas as soluções e ferramentas relacionadas necessitam de acompanhá-lo. Por sua vez, as melhores práticas que lhe são associadas e que foram apresentadas devem também manter-se em constante atualização e aperfeiçoamento devido ao facto de no contexto da web surgirem a cada dia novas vulnerabilidades e ameaças a nível da segurança.

Para finalizar, sendo esta dissertação um trabalho de investigação, uma possibilidade para a disseminação desta abordagem na comunidade tecnológica seria a publicação de um artigo científico.

No que toca ao caso de estudo, a concepção e planeamento da arquitetura de interoperabilidade é o início de um longo investimento e evolução da UN1Qnx no IT. Assim, no presente, existe a necessidade de continuar a implementar os serviços que se pretendem disponibilizar, e prosseguir na construção e aperfeiçoamento das plataformas já existentes de modo a satisfazer as necessidades reais dos clientes do sistema UN1Qnx.

No momento da escrita deste documento a API concebida ainda não foi lançada, dado que ainda serão adicionadas novas funcionalidades bem como melhoradas as já existentes, com vista a promover a escalabilidade e a adaptação da aplicação. Neste sentido, em termos de trabalho futuro, uma das primeiras tarefas a realizar será a atualização da versão .NETCore utilizada e alojamento da nova API REST desenvolvida em ambiente *cloud*.

BIBLIOGRAFIA

Web services coordination (ws-coordination) version 1.2, 2009. URL http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.2-spec-os/wstx-wscoor-1.2-spec-os.html.

APISecurity.io. URL https://apisecurity.io/.

G. McGraw B. Potter. *Software security testing*. IEEE Security & Privacy, 2004.

Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for agile software development, 2001. URL http://agilemanifesto.org/.

Andrew Birrell and Bruce Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984. URL https://birrell.org/andrew/papers/ImplementingRPC.pdf.

L. Byron. "designing a data language", presented at the strange loop 2016, st. louis, sep. 2016. URL https://www.youtube.com/watch?v=Oh5oC98ztvI.

Brian Totty David Gourley. *HTTP: The Definitive Guide*. O'Reilly Media, Inc., 2002.

Developer.Mozilla.org. Mensagens http. URL https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Mensagens.

Roy Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, 2000.

Greg McGraw Gary Hoglund. *Exploiting Software How to Break Code*. Addison Wesley, 2004.

GeeksforGeeks.org. Remote procedure call (rpc) in operating system. URL https://www.geeksforgeeks.org/remote-procedure-call-rpc-in-operating-system/.

GraphQL. Graphql documentation. URL https://graphql.org/.

gRPC Authors. grpc documentation. URL https://grpc.io/.

Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS Quarterly*, 28(1):75–105, 2004. URL http://www.jstor.org/stable/25148625.

Flávio Horita, Valdemar Graciano Neto, and Rodrigo dos Santos. *Design Science Research em Sistemas de Informação e Engenharia de Software: Conceitos, Aplicações e Trabalhos Futuros*, pages 192–210. 04 2018. ISBN 978-85-7669-458-8.

howtographql.com. Graphql is the better rest. URL https://www.howtographql.com/basics/1-graphql-is-the-better-rest/.

imaginarycloud.com. Grpc vs rest: Comparing apis architectural styles. URL https://www.imaginarycloud.com/blog/grpc-vs-rest/.

Imperva. The state of web application vulnerabilities in 2018. URL https://www.imperva.com/blog/the-state-of-web-application-vulnerabilities-in-2018/.

Petter Iversen. Specification-based security analysis of rest apis. Master's thesis, Norwegian University of Science and Technology, 2018. URL http://hdl.handle.net/11250/2560780.

Sam Ruby Leonard Richardson. *RESTful Web Services*. O'Reilly Media, 2007.

Ana Isabel Alves Marques. Desenvolvimento de api para aplicação cloud. Master's thesis, Instituto Politécnico de Leiria, 2018. URL https://iconline.ipleiria.pt/bitstream/10400.8/3263/1/2151668_Ana%20Marques_MEICM_Tese.pdf.

Sadruddin Md. Tutorial on oauth 2.0 authorization code (with refresh token) flow, a. URL https://iteritory.com/tutorial-on-oauth-2-0-authorization-code-with-refresh-token-flow/.

Sadruddin Md. Article on what is oauth 2.0 client credential grant type, b. URL https://iteritory.com/article-on-what-is-oauth-2-0-client-credential-grant-type/.

Microsoft. What is asp.net core?, a. URL https://dotnet.microsoft.com/learn/aspnet/what-is-aspnet-core.

Microsoft. Entity framework core, b. URL https://docs.microsoft.com/en-us/ef/core/.

Microsoft. Integration tests in asp.net core, c. URL https://docs.microsoft.com/en-us/aspnet/core/test/integration-tests?view=aspnetcore-6.0.

A. Neumann, N. Laranjeiro, and J. Bernardino. An analysis of public rest web service apis. *IEEE Transactions on Services Computing*, pages 1–1, 2018. doi: 10.1109/TSC.2018.2847344.

OWASP. Owasp api security project, a. URL https://owasp.org/www-project-api-security/.

OWASP. Rest security cheat sheet, b. URL https://cheatsheetseries.owasp.org/cheatsheets/REST_Security_Cheat_Sheet.html.

Mayur Pahwa. Digital signature. URL https://www.mayurpahwa.com/2019/01/digital-signature.html.

Ken Peffers, Tuure Tuunanen, Marcus A. Rothenberger, and Samir Chatterjee. A design science research methodology for information systems research. *Journal of Management Information Systems*, 24 (3):45–77, 2007. doi: 10.2753/MIS0742-1222240302. URL https://doi.org/10.2753/MIS0742-1222240302.

ProgrammableWeb.com. Apis show faster growth rate in 2019 than previous years. URL https://www.programmableweb.com/news/apis-show-faster-growth-rate-2019-previous-years/research/2019/07/17.

J. Reschke R. Fielding. Hypertext transfer protocol (http/1.1): message syntax and routing, rfc 7230 (proposed standard),internet engineering task force, a. URL https://www.rfc-editor.org/rfc/inline-errata/rfc7230.html.

J. Reschke R. Fielding. Hypertext transfer protocol (http/1.1): semantics and content, rfc 7231 (proposed standard),internet engineering task force, b. URL https://www.rfc-editor.org/rfc/inline-errata/rfc7231.html.

Rodrigo Miguel Corredoura Janota Ratos. Sora - finding workflow violation attacks in rest apis. Master's thesis, Instituto Superior Técnico da Universidade de Lisboa, 2019. URL https://fenix.tecnico.ulisboa.pt/departamentos/dei/dissertacao/1691203502343443.

Carlos Rodríguez, Marcos Baez, Florian Daniel, Fabio Casati, Juan Carlos Trabucco, Luigi Canali, and Gianraffaele Percannella. Rest apis: A large-scale analysis of compliance with principles and best practices. In Alessandro Bozzon, Philippe Cudre-Maroux, and Cesare Pautasso, editors, *Web Engineering*, pages 21–39, Cham, 2016. Springer International Publishing. ISBN 978-3-319-38791-8.

Brian Suda. Soap web services. Master's thesis, School of Informatics University of Edinburgh, 2003. URL https://suda.co.uk/publications/MSc/brian.suda.thesis.pdf.

W3C. Soap version 1.2 part 1: Messaging framework, 2007, a. URL https://www.w3.org/TR/soap12-part1/.

W3C. Web services description language, 2001, b. URL https://www.w3.org/TR/wsdl20/.

A

USE CASES

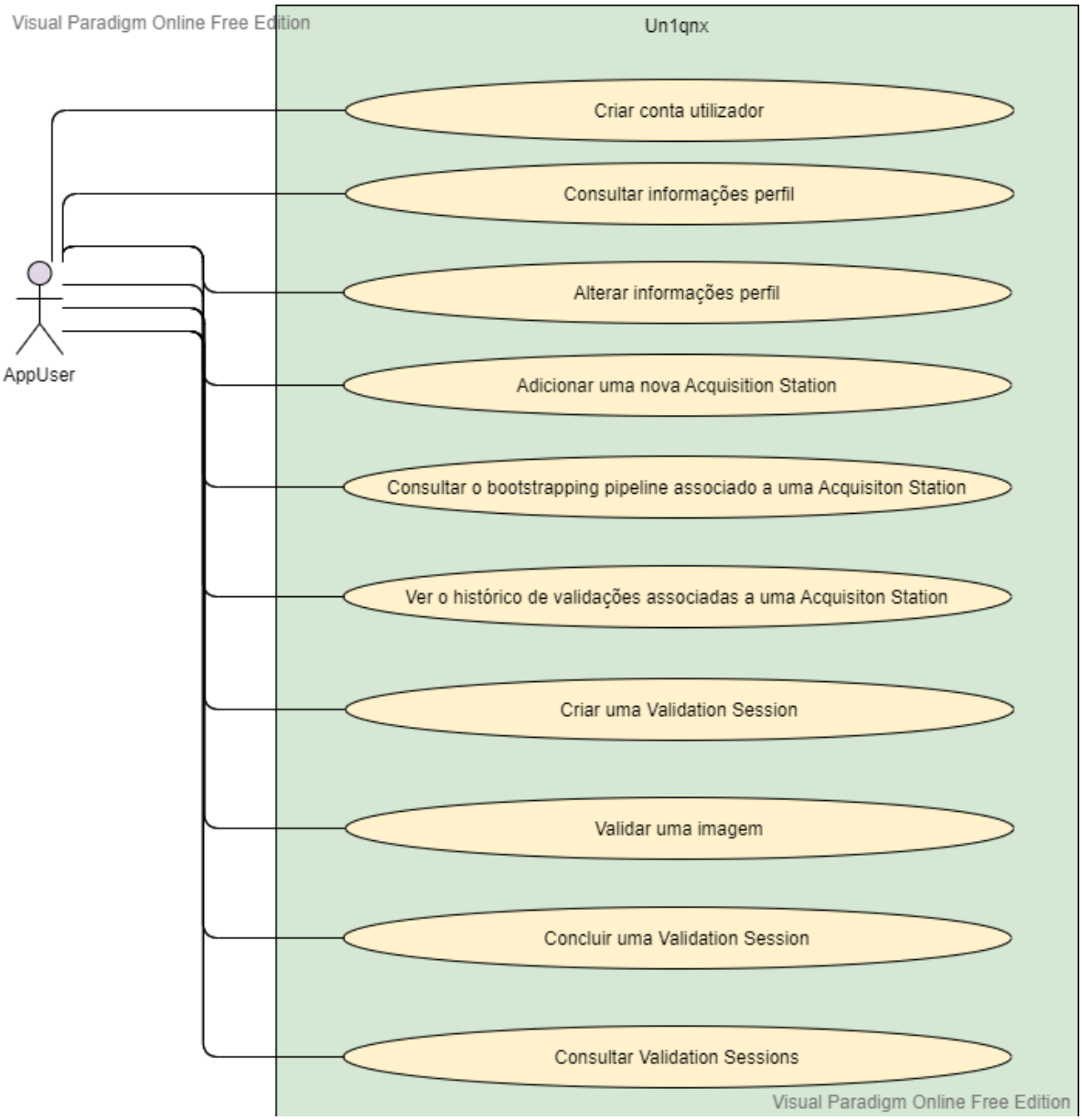


Figura 29: Use Cases do AppUser

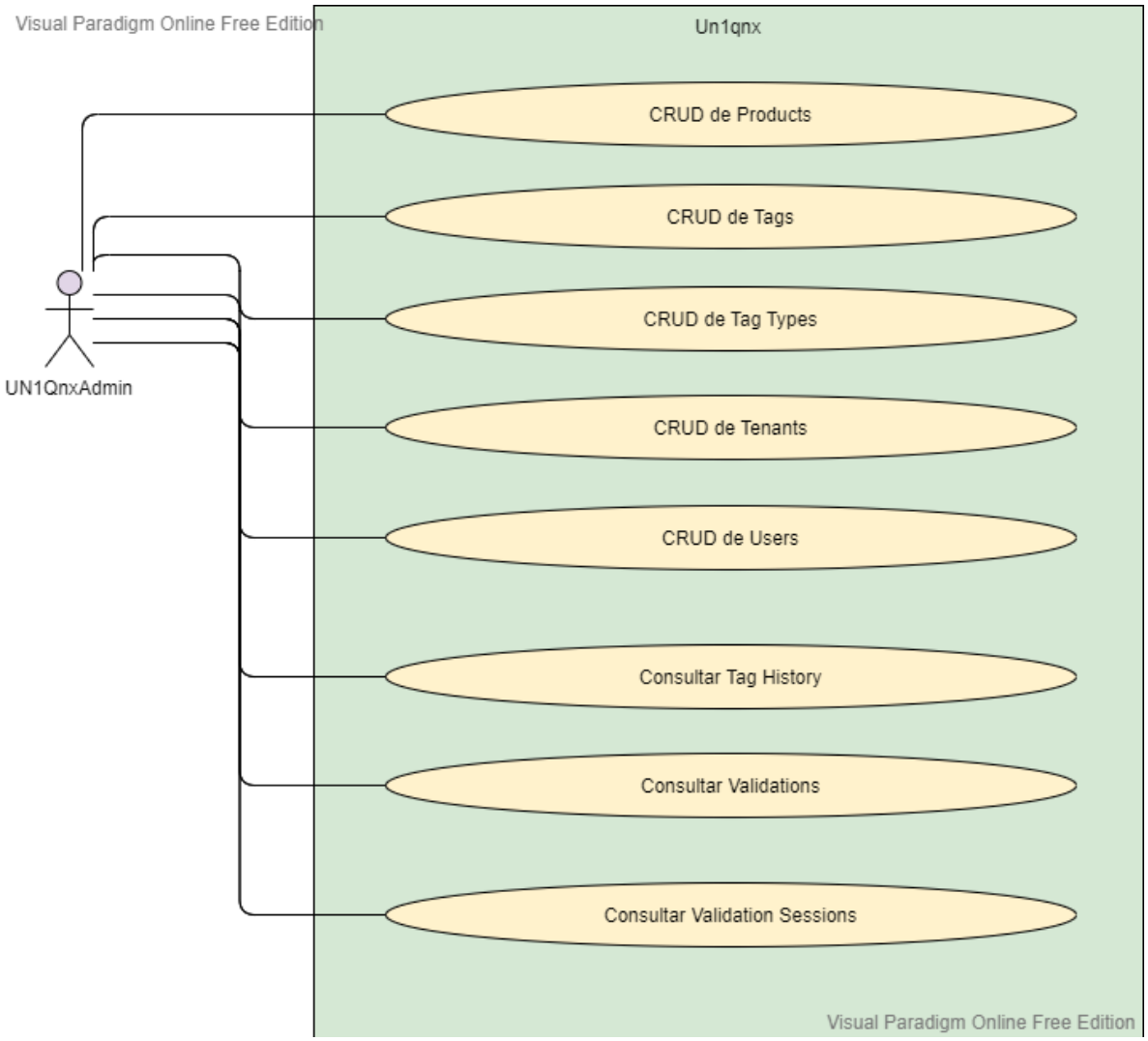


Figura 30: Use Cases do UN1QnxAdmin

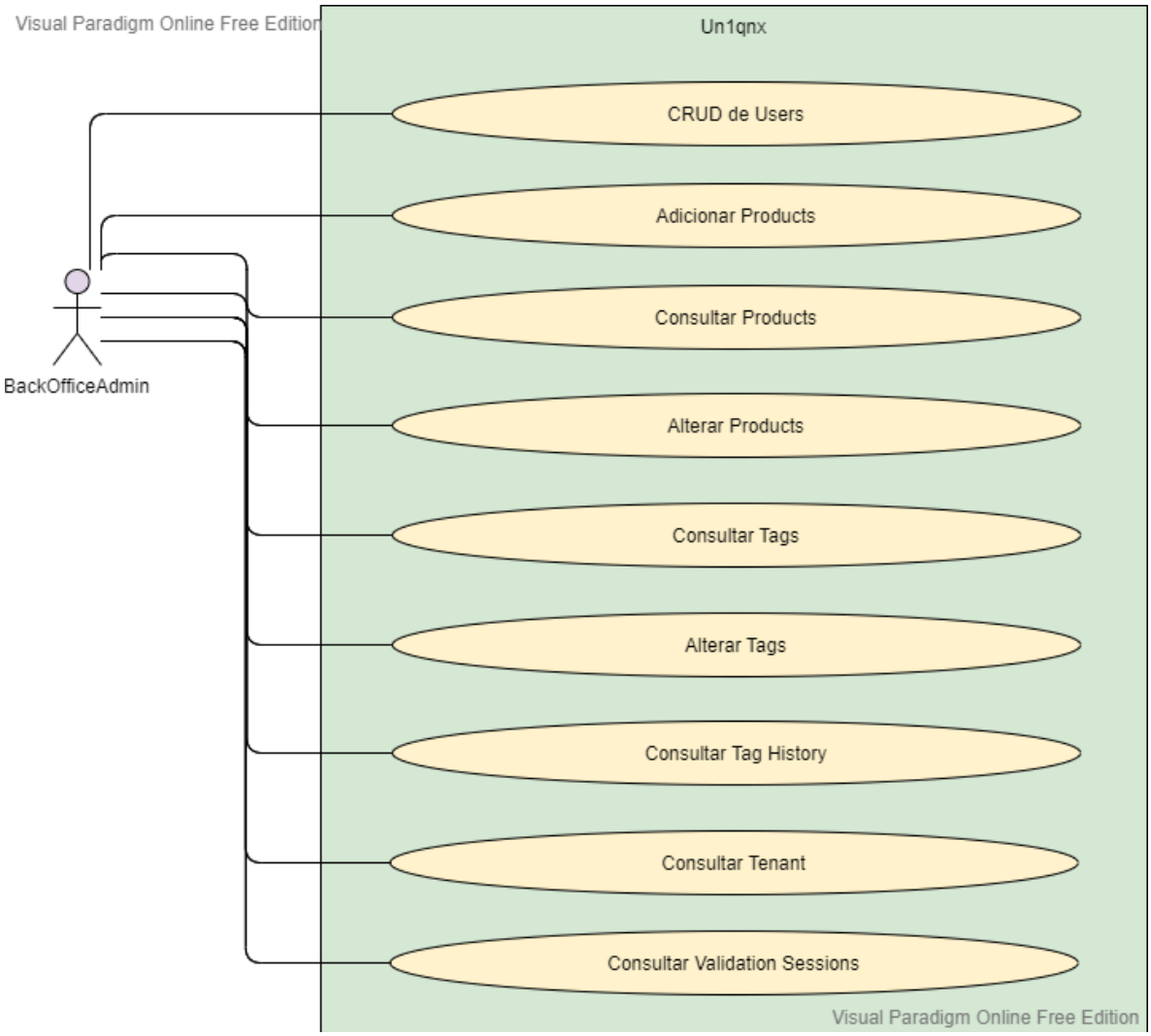


Figura 31: Use Cases do BackOfficeAdmin

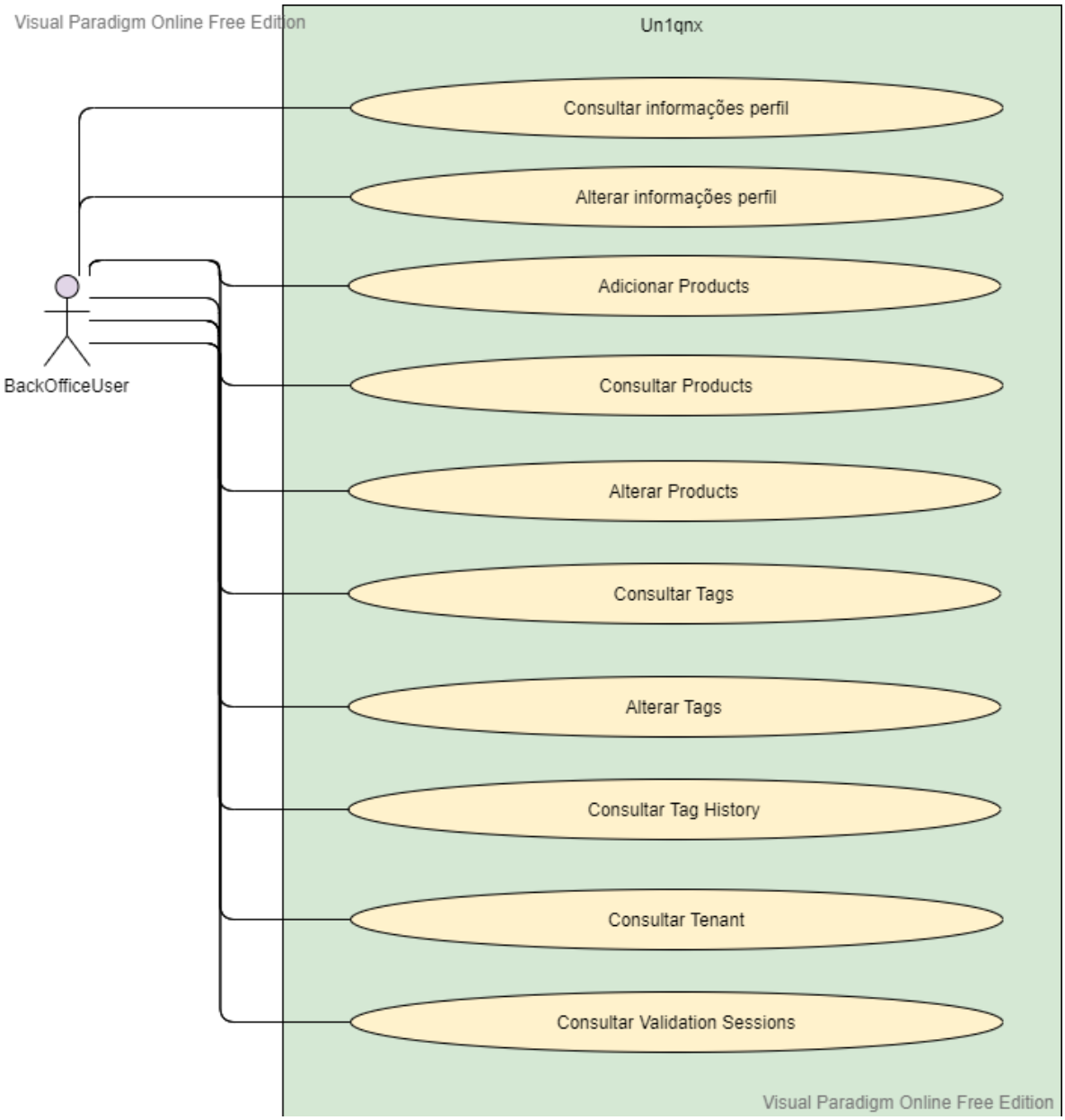


Figura 32: Use Cases do BackOfficeUser

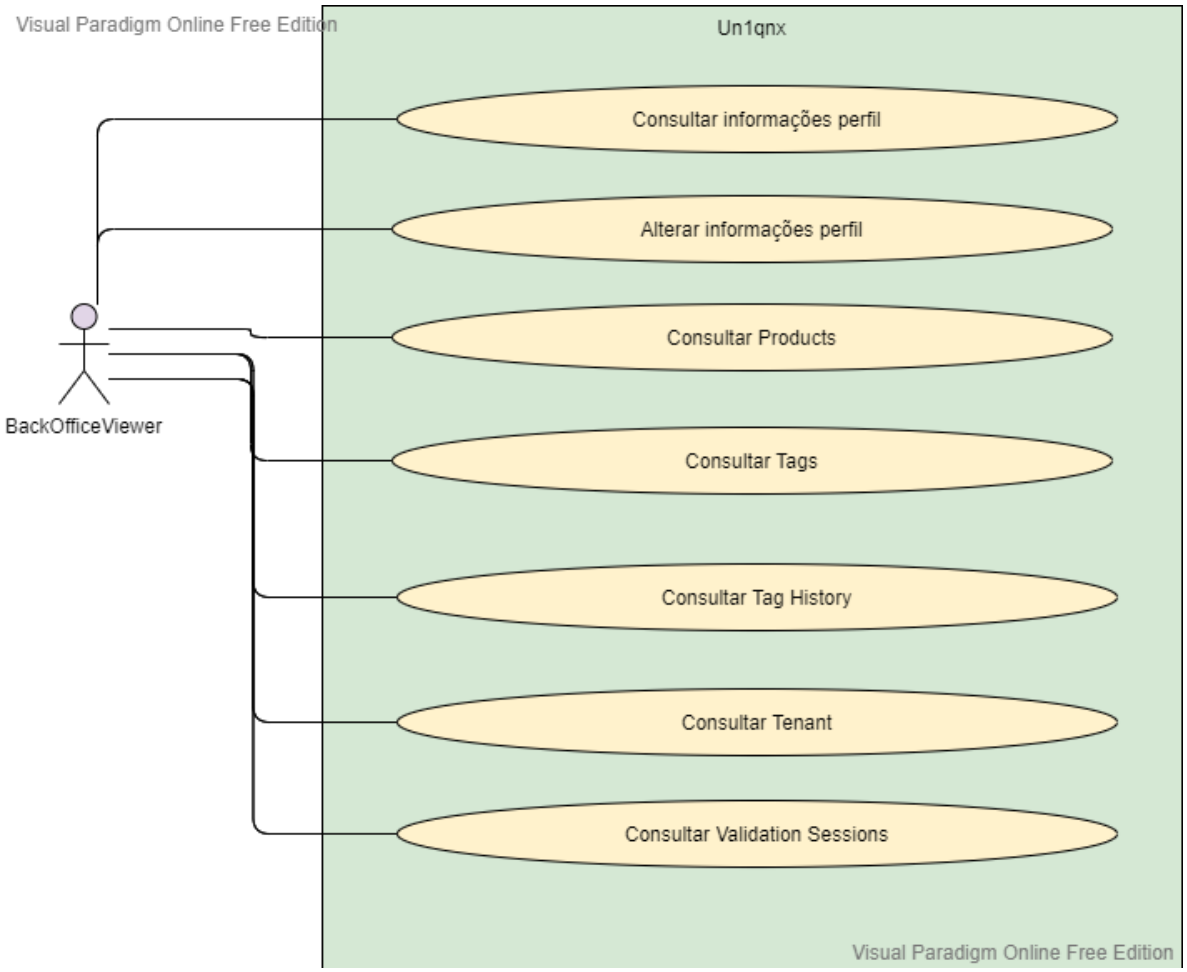


Figura 33: Use Cases do BackOfficeViewer

B

DIAGRAMA DA BASE DE DADOS

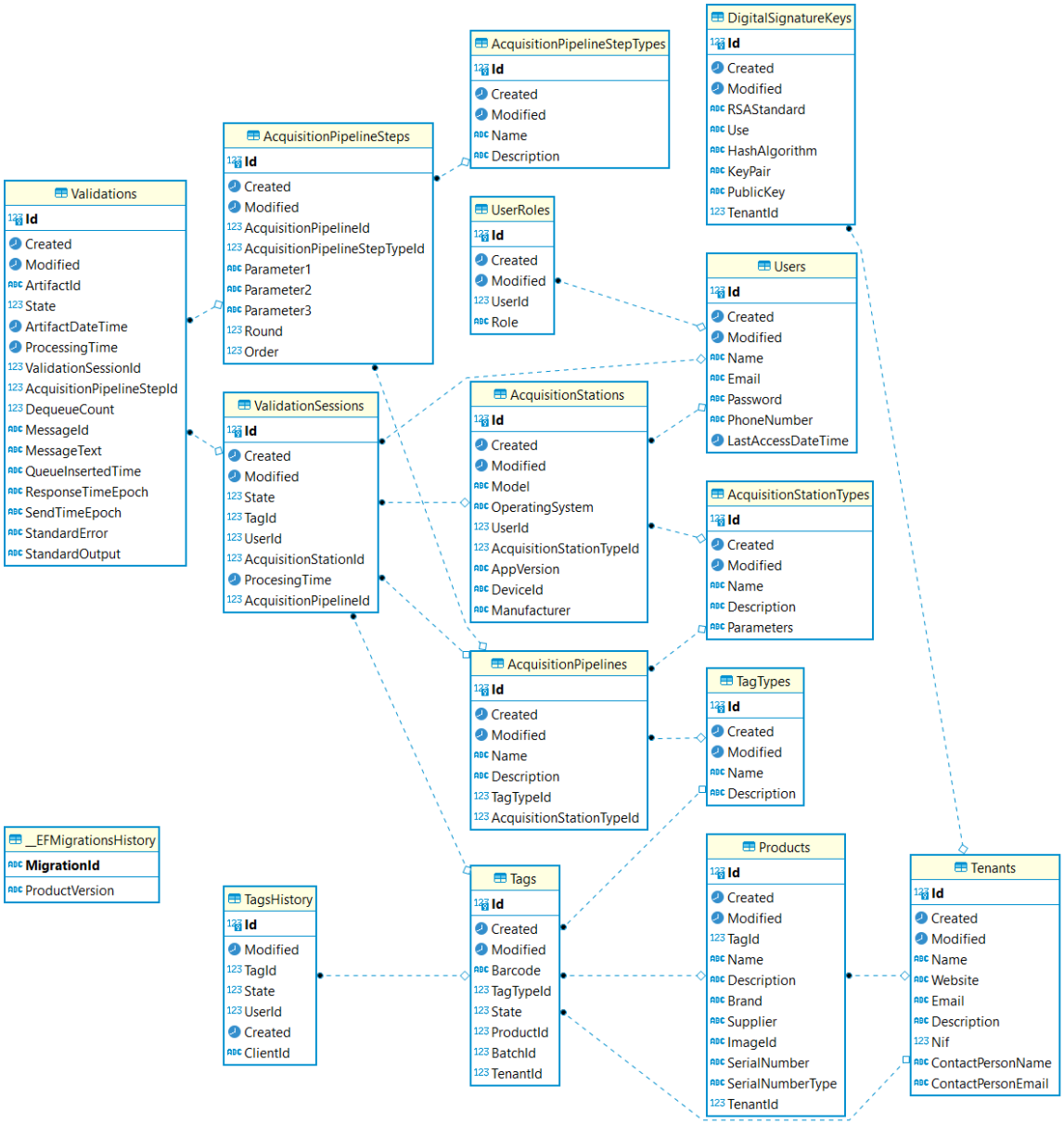
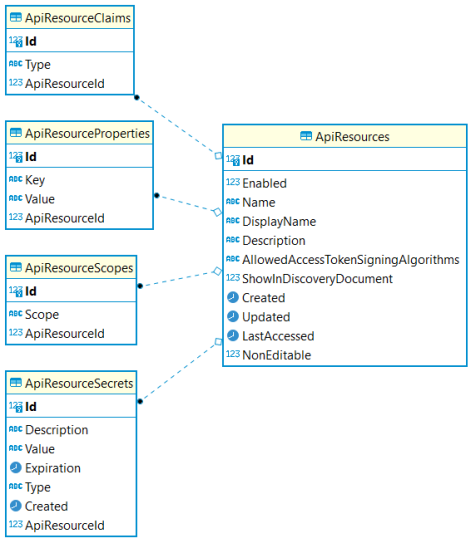
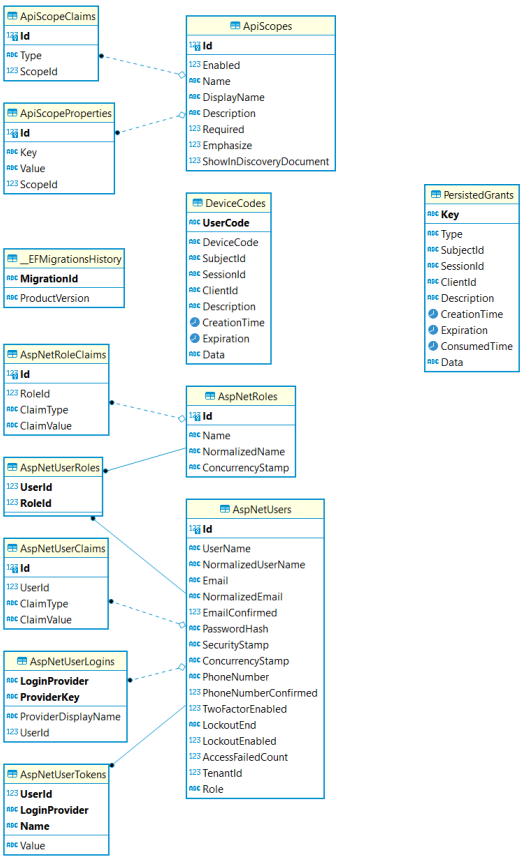


Figura 34: Diagrama da base de dados principal do sistema UN1Qnx


ApiResourceClaims
Id
Type
ApiResourceId
ApiResourceProperties
Id
Key
Value
ApiResourceId
ApiResourceScopes
Id
Scope
ApiResourceId
ApiResourceSecrets
Id
Description
Value
Expiration
Type
Created
ApiResourceId
ApiResources
Id
Enabled
Name
DisplayName
Description
AllowedAccessTokenSigningAlgorithms
ShowInDiscoveryDocument
Created
Updated
LastAccessed
NonEditable


ApiScopeClaims
Id
Type
ScopeId
ApiScopeProperties
Id
Key
Value
ScopeId
ApiScopes
Id
Enabled
Name
DisplayName
Description
Required
Emphasize
ShowInDiscoveryDocument
DeviceCodes
UserCode
DeviceCode
SubjectId
SessionId
ClientId
Description
CreationTime
Expiration
Data
PersistedGrants
Key
Type
SubjectId
SessionId
ClientId
Description
CreationTime
Expiration
ConsumedTime
Data
__EFMigrationsHistory
MigrationId
ProductVersion
AspNetRoleClaims
Id
RoleId
ClaimType
ClaimValue
AspNetRoles
Id
Name
NormalizedName
ConcurrencyStamp
AspNetUserRoles
UserId
RoleId
AspNetUsers
Id
UserName
NormalizedUserName
Email
NormalizedEmail
EmailConfirmed
PasswordHash
SecurityStamp
ConcurrencyStamp
PhoneNumber
PhoneNumberConfirmed
TwoFactorEnabled
LockoutEnd
LockoutEnabled
AccessFailedCount
TenantId
Role
AspNetUserClaims
Id
UserId
ClaimType
ClaimValue
AspNetUserLogins
LoginProvider
ProviderKey
ProviderDisplayName
UserId
AspNetUserTokens
UserId
LoginProvider
Name
Value

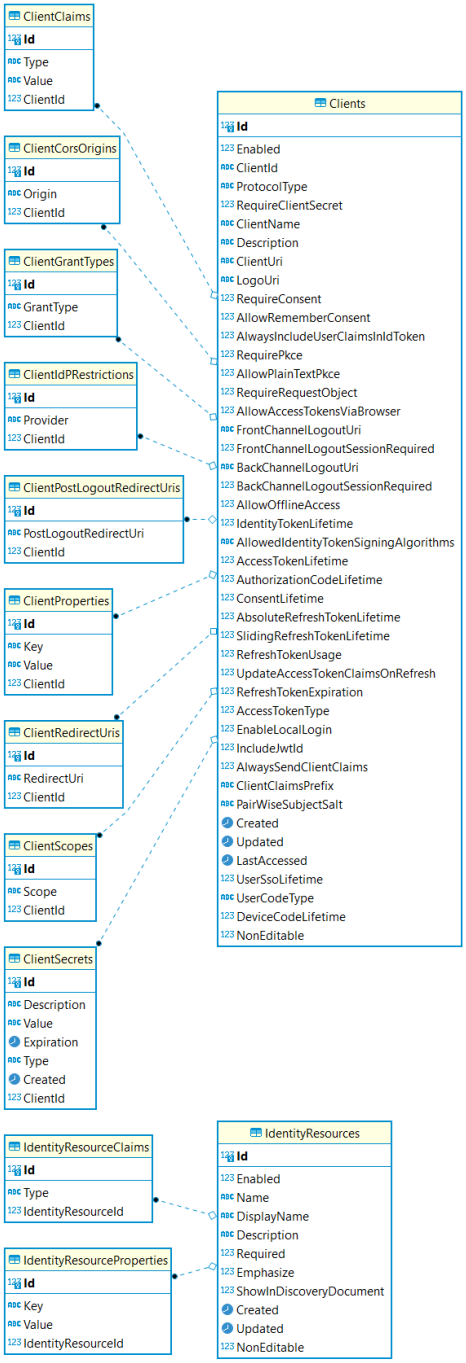
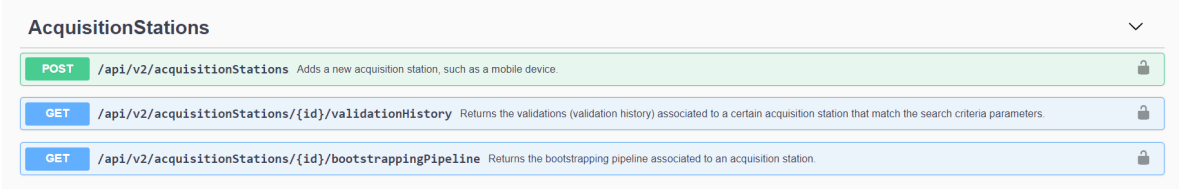
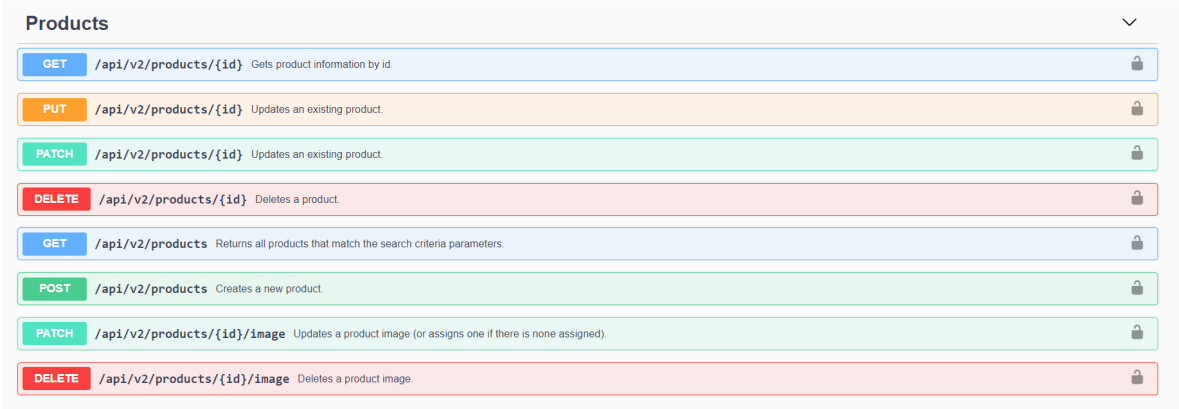


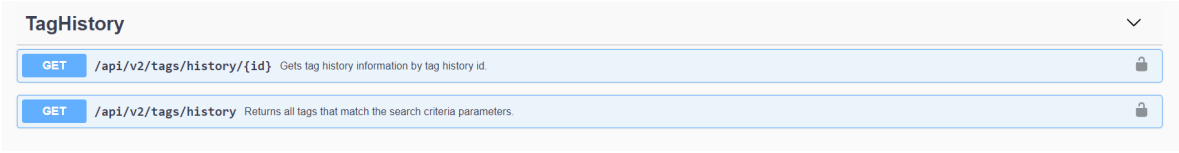
Figura 35: Diagrama da base de dados do Identity Server

C

API RESTFUL







Tags

GET	/api/v2/tags/{id}	Gets tag information by id.
PUT	/api/v2/tags/{id}	Updates an existing tag by id.
PATCH	/api/v2/tags/{id}	Updates an existing tag by id.
DELETE	/api/v2/tags/{id}	Deletes a tag by id.
GET	/api/v2/tags/byCode/{tagCode}	Gets tag information by tagCode.
PUT	/api/v2/tags/byCode/{tagCode}	Updates an existing tag by tagCode.
PATCH	/api/v2/tags/byCode/{tagCode}	Updates an existing tag by tagCode.
DELETE	/api/v2/tags/byCode/{tagCode}	Deletes a tag by tagCode.
GET	/api/v2/tags	Returns all tags that match the search criteria parameters.
POST	/api/v2/tags	Creates a new tag.
GET	/api/v2/tags/{id}/history	Gets tag history information by id.
GET	/api/v2/tags/byCode/{tagCode}/history	Gets tag history information by tagCode.
PATCH	/api/v2/tags/{id}/partial	Updates an existing tag by id.
PATCH	/api/v2/tags/byCode/{tagCode}/partial	Updates an existing tag by tagCode.
PATCH	/api/v2/tags/bulk/partial	Updates tags in bulk.
PATCH	/api/v2/tags/byBatch/{batchId}	Updates tag state by batch id.
POST	/api/v2/tags/bulk	Creates tags in bulk.
PUT	/api/v2/tags/bulk	Updates tags in bulk.
PATCH	/api/v2/tags/bulk	Updates tags in bulk.
DELETE	/api/v2/tags/bulk	Deletes tags in bulk (by barcode).

TagTypes

GET	/api/v2/tags/types	Returns all tag types.
POST	/api/v2/tags/types	Creates a new tag type.
GET	/api/v2/tags/types/{id}	Gets tag type information by id.
PUT	/api/v2/tags/types/{id}	Updates an existing tag type.
PATCH	/api/v2/tags/types/{id}	Updates an existing tag type.
DELETE	/api/v2/tags/types/{id}	Deletes a tag type.

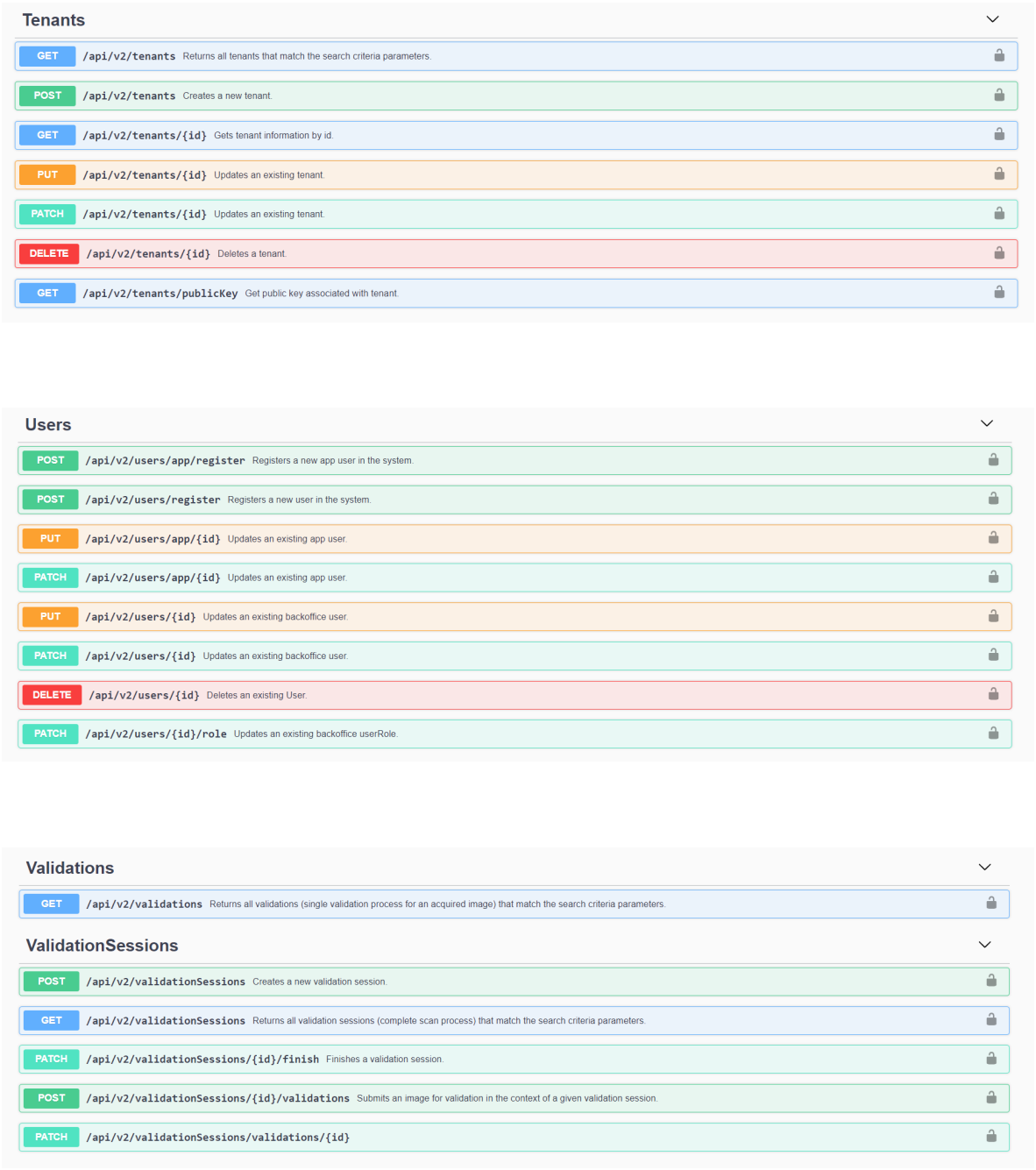


Figura 36: Documentação de todos os métodos da API

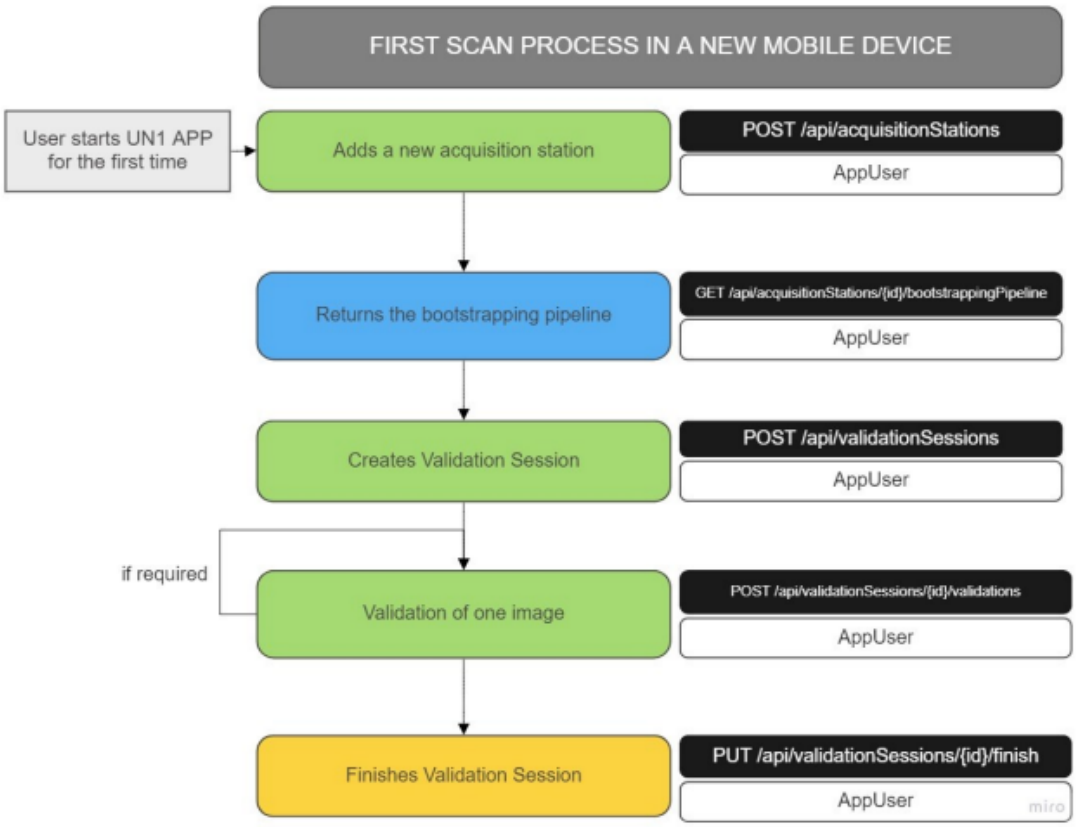


Figura 37: Fluxo de um scan de validação na app UN1Qnx

D

REQUISITOS

Users

- login e registo de utilizadores;

- Alterar informações do utilizador;

Products

- Adicionar/criar um novo produto;

- Obter a lista de todos os produtos;

- Obter um produto através do seu id;

- Alterar ou apagar um produto através do seu id;

- Alterar ou apagar a imagem de um produto através do seu id;

Tags

- Adicionar/criar uma nova tag;

- Obter a lista de todas as tags;

- Obter uma tag através do seu id;

- Alterar ou apagar uma tag através do seu id;

- Adicionar um conjunto de novas tags;

- Alterar ou apagar um conjunto de novas tags;

TagsTypes

- Adicionar uma nova tag type;

- Obter a lista de todas as tags types;

- Obter uma tag type através do seu id;

- Alterar ou apagar uma tag type através do seu id;

ValidationSessions

- Adicionar uma nova validation session;

- Obter a lista de todas as validation sessions;

- POST /api/validationSessions/{id}/validations – ato de validar um determinado step do pipeline;

- PUT /api/validationSessions/{id}/finish – terminar validacao: ver quantos steps foram aprovados e concluir a validação.

Validations

- Obter a lista de todas as validations efetuadas;

Requisitos adicionais	Por iniciar	Em desenvolvimento	Concluído	Aprovado
• Fazer análise de vulnerabilidades e de segurança				X
• Alterar apenas o estado de uma tag				X
• Alterar vários estados de uma tag de uma vez só. Para tal, adicionar uma coluna BatchId (Lote) à tabela Tags				X
• Obter todo o histórico de alterações a uma tag; (nova tabela)				X
• Implementar updates parciais aos recursos da API				X
• Atualizar UserRoles: Un1qnxAdmin, BackofficeAdmin, BackofficeUser, BackofficeViewer e AppUser				X
• Utilizadores devem aceder apenas a tags/produtos, validações,etc relacionados consigo.				X
• Criar uma tabela Tenant para guardar informações relativas a cada empresa				X
• Adicionar à tabela Products a coluna tenantId				X
• Adicionar à tabela Users a coluna tenantId				X
• Adicionar métodos CRUD para backoffice users				X
• Definir conforme os UserRoles as autorizações para cada método				X
• Adicionar método para alterar userRoles				X
• Distinguir login e registo entre App users e Backoffice users				X
• Adicionar mecanismo do IdentityServer				X
• Adicionar versionamento da API				X
• Atualizar o flow de Validação				X
• Adicionar security headers middleware				X
• Adicionar Cross Site Scripting Middleware				X
• Adicionar HATEOAS				X

Figura 38: Lista de requisitos da API