Um estudo de caso sobre avaliação de uma API de sistema Web baseado em Microsserviços

Samuel Gomes Bezerra Araújo

CENTRO DE INFORMÁTICA
UNIVERSIDADE FEDERAL DA PARAÍBA

João Pessoa, 2021

Samuel Gomes Bezerra Araújo

Um estudo de caso sobre avaliação de uma API de sistema Web baseado em Microsserviços

Monografia apresentada ao curso Ciência da Computação do Centro de Informática, da Universidade Federal da Paraíba, como requisito para a obtenção do grau de Bacharel em em Ciência da Computação

Orientador: Professor Dr. Raoni Kulesza

Julho de 2021

Catalogação na publicação
Seção de Catalogação e Classificação

A663e Araújo, Samuel Gomes Bezerra.
Um estudo de caso sobre avaliação de uma API de sistema Web baseado em Microsserviços / Samuel Gomes Bezerra Araújo. - João Pessoa, 2021.
72 f. : il.

Orientação: Raoni Kulesza.
TCC (Graduação) - UFPB/CI.

1. Software. 2. Arquitetura de software. 3. API. 4. Avaliação arquitetural. I. Kulesza, Raoni. II. Título.

UFPB/CI CDU 004.4

Elaborado por ROGERIO FERREIRA MARQUES - CRB-15/690

UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE INFORMÁTICA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO
COORDENAÇÃO DO CURSO

Ata da Sessão Pública de Defesa de Trabalho de Conclusão de Curso de **Ciência da Computação**, realizada em **23** de **julho** de **2021**.

Aos **23** dias do mês de **julho**, do ano de **2021**, às **10:30** horas, por meio de videoconferência, reuniram-se os membros da Banca Examinadora constituída para julgar o Trabalho de Conclusão de Curso da Sr. **Samuel Gomes Bezerra Araújo**, matrícula nº **20160163773**, discente do Curso de Bacharelado em Ciência da Computação da Universidade Federal da Paraíba. A comissão examinadora foi composta pelo professor **Raoni Kulesza** (UFPB), orientador e presidente da banca, e pelos membros **Derzu Omaia** (UFPB), examinador interno, **Eduardo de Santana Medeiros Alexandre** (Overmediacast), examinador externo. Iniciando os trabalhos, o presidente da banca cumprimentou os presentes, comunicou-os da finalidade da reunião e passou a palavra ao candidato para que fizesse a exposição oral da monografia intitulada "**Um estudo de caso sobre avaliação de uma API de sistema Web baseado em Microsserviço**". Concluída a exposição, o candidato foi arguido pela Banca Examinadora que, em seguida, emitiu o seguinte parecer: "**APROVADO**", com conceito 9,0 (0,0 a 10,0). Do ocorrido, eu, Gustavo Henrique Matos Bezerra Motta, Coordenador do Curso de Bacharelado em Ciência da Computação, lavrei a presente ata 15 que vai assinada por mim e pelos membros da banca examinadora.

João Pessoa, 23 de julho de 2021

Prof. Gustavo Henrique Matos Bezerra Motta

gov.br Documento assinado digitalmente
Gustavo Henrique Matos Bezerra Motta
Data: 09/11/2021 12:05:25-0300
Verifique em https://verificador.iti.br

Raoni Kulesza
Orientador (UFPB)

Derzu Omaia
Examinador Interno (UFPB)

Eduardo de Santana Medeiros Alexandre
Examinador Externo (Overmediacast)

AGRADECIMENTOS

Dedico esse trabalho a toda minha família, por todo o suporte que me forneceram ao longo dos anos, para que eu pudesse alcançar este objetivo. Agradeço também aos meus professores pela paciência e pelo conhecimento passado; nada disso seria possível sem eles. Em especial, agradeço aos professores Raoni Kulesza, pelo apoio e suporte para com este trabalho, e pelas oportunidades de aprendizado, e Gustavo Henrique, por toda ajuda que ele me forneceu durante esta trajetória.

Por último, mas não menos importante, agradeço aos colegas de curso que tive a oportunidade de conhecer e aos funcionários da instituição, que me ajudaram de alguma forma durante esta caminhada que, não se encerra, mas agora se inicia em uma jornada profissional.

RESUMO

Nos últimos tempos, o padrão arquitetural de microsserviços se consolidou como uma boa alternativa para sistemas Web. Nesse contexto, é importante a presença de um módulo que centralize e controle o acesso aos demais módulos, tal elemento normalmente é chamado de API (do inglês, *Application Programming Interface*). Uma API é um conjunto de código de programação que fazem parte de uma interface e que permitem a criação de diferentes softwares, aplicativos, programas e plataformas de maneira que facilite a vida dos desenvolvedores. Este trabalho tem como principal objetivo realizar uma avaliação de uma API de um sistema de vídeos interativos que utiliza o padrão arquitetural de microsserviços. Para isso, foi elaborada uma documentação da arquitetura atual do objeto de análise do sistema e em seguida foram realizadas avaliações estáticas e dinâmica do código da solução. A partir do estudo foi possível propor mudanças no sistema que fornecem melhorias em relação aos atributos de qualidade do projeto de software do sistema analisado.

Palavras-chave: Arquitetura de software, API, Estudo de caso, Avaliação arquitetural.

ABSTRACT

In recent times, the architectural pattern of microservices has established itself as a good alternative for Web systems. In this context, it is important to have a module that centralizes and controls access to other modules, such an element is usually called an API. An API is a set of programming code that is part of an interface and that allows the creation of different softwares, applications, programs and platforms in a way that makes life easier for developers. This work has as main objective to carry out an evaluation of an API of an interactive video system that uses the microservices architectural pattern. For this, a documentation of the current architecture of the system's object of analysis was elaborated and then static and dynamic evaluation of source code were carried out. From the study, it was possible to propose changes to the system that provide improvements in terms of safety and maintenance attributes.

Key-words: Software architecture, API, Case study, Architectural evaluation.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS

AJAX	–	Asynchronous Javascript and eXtensible Markup Language
API	–	Application Programming Interface
AWS	–	Amazon Web Services
CDN	–	Content Delivery Network
CGI	–	Common Gateway Interface
HATEOAS	–	Hypermedia as the Engine of Application State
HTTP	–	Hypertext Transfer Protocol
IEC	–	International Electrotechnical Commission
IEEE	–	Institute of Electrical and Electronics Engineers
ISO	–	International Organization for Standardization
MVC	–	Model View Controller
OWASP	–	Open Web Application Security Project
REST	–	Representational State Transfer
ROR	–	Ruby on Rails
RPC	–	Remote Procedure Call
SMS	–	Short Message Service
SOA	–	Service-Oriented Architecture
SOAP	–	Simple Object Access Protocol
UML	–	Unified Modeling Language
URI	–	Uniform Resource Identifier
URL	–	Uniform Resource Locator
ZAP	–	Zed Attack Proxy

SUMÁRIO

1 INTRODUÇÃO

A partir do momento em que começaram a surgir sistemas de softwares mais complexos, foi necessário um aumento de tempo e custo investidos para sua produção. Essas condições exigiram aperfeiçoar o processo de desenvolvimento, a fim de evitar o problema de produzir uma solução ineficiente, ou seja, que não alcance os objetivos requeridos pelo cliente. Para isso, a Engenharia de Software aderiu a uma prática bastante comum já utilizada por áreas da engenharia e passou a utilizar um projeto para guiar a construção de sistemas complexos (GERMOGLIO, 2012).

A projeção de uma arquitetura de um sistema tem como base não só suas funcionalidades principais, mas também os atributos de qualidades que devem estar presentes (BASS et al, 2001). Apesar de ser possível usar esses atributos para fazer comparações entre arquiteturas distintas ou até mesmo avaliar o efeito de mudanças em uma arquitetura existente, na prática, a arquitetura de software ainda sofre várias alterações em sua composição devido a realidade do processo de desenvolvimento.

Em situações como de startups, onde é comum um ambiente de muitas mudanças com a necessidade de agilidade, o processo se torna bastante limitado por alguns fatores, como tempo, financeiro e até organizacional. Dessa forma, é normal que certas atividades e artefatos do projeto de software sejam descartadas (BABAR et al, 2013). Apesar disso, a comunidade tem conhecimento que a arquitetura de software é um instrumento bastante eficaz quando se pensa em melhoria de qualidade de um software e na redução do custo não só do seu desenvolvimento mas também nas capacidades evolutivas.

1.1 Tema

O padrão arquitetural de microsserviços possibilita o desenvolvimento de sistemas complexos a partir da inclusão de pequenos serviços que fornecem funcionalidades específicas. Esse padrão oferece benefícios em relação aos atributos de qualidade, como a escalabilidade e manutenibilidade. Já o uso do módulo de API (do inglês, *Application Programming Interface*) permite que haja uma centralização e um controle de acesso aos demais módulos do sistema. É através dessa interface que é possível, por exemplo, liberar ou bloquear o acesso ao conteúdo de um sistema, de forma prática. A partir disso, a necessidade desse módulo não se faz apenas na sua presença, mas também na sua viabilidade. Para tal, é necessário uma avaliação da atual situação da API implementada, em busca de erros e melhorias. Nesse contexto, a avaliação de uma API de um sistema Web que utiliza arquitetura de microsserviços permite caracterizar esse sistema e avaliar a sua capacidade de atender aos objetivos para qual foi projetado.

1.2 Problema

Uma API é um conjunto de código de programação que possibilita a transmissão de dados entre diferentes softwares, além de integrar serviços, empresas, negócios e parceiros. Uma API bem construída facilita esse processo, mas para isso, é necessário ter uma documentação apropriada. Sem documentar a API de maneira adequada, quem for utilizá-la perderá tempo tentando entender o seu funcionamento, o que cria barreiras para a adoção do seu serviço (JÚNIOR MANOEL, 2019).

Antes de iniciar de fato a documentação, é importante ter uma modelagem gráfica da arquitetura do sistema para entender como foi projetada e o seu fluxo de execução. Para isto, é comumente usado o modelo C4, que foi desenvolvido com intuito de auxiliar desenvolvedores na comunicação e descrição da arquitetura de um

software (BROWN, 2018), que são divididos em quatro níveis de abstração, onde cada um possui um nível de detalhamento diferente.

A documentação de uma API precisa ser bem completa, tendo como foco principal os recursos e endpoints disponíveis da mesma (JÚNIOR MANOEL, 2019). Existem alguns passos que devem estar presentes na documentação, como:

1. A descrição da funcionalidade fornecida;
2. Parâmetros de entrada, de forma que especifique se é obrigatório ou não, o tipo de valor esperado e como o valor é recebido;
3. Formato da resposta;
4. Se é necessário ou não ter autenticação;
5. Limitações de uso;
6. Se for uma API baseada no protocolo HTTP (do inglês, *Hypertext Transfer Protocol*), é necessário especificar quais verbos (*GET, POST, PUT, DELETE, PATCH*) são aceitos pelo *endpoint*;
7. Descrição dos possíveis retornos da funcionalidade, tanto de sucesso quanto de erro, esclarecendo o significado desse último quando ocorrer;
8. Procurar mostrar exemplos de uso.

Nesse contexto, este trabalho procura documentar e avaliar uma determinada API que foi criada a partir de uma arquitetura de sistemas Web baseada em microsserviços. Para tal fim, foi selecionado como objeto de estudo de caso um sistema de uma startup, onde é comum que a implementação seja realizada sem um projeto de software adequado e, dessa forma, dificultando a avaliação dos atributos de qualidade para testes, manutenção e até evolução do sistema. Devido ao escopo deste estudo, a pesquisa ficou restrita a documentação, avaliação e sugestão de mudanças de uma API.

1.2.1 Objetivo geral

Este trabalho tem como objetivo principal realizar uma avaliação da arquitetura de uma API de um sistema Web que utiliza o padrão arquitetural de microsserviços.

1.2.2 Objetivos específicos

No que diz respeito aos objetivos específicos, pretende-se:

1. Estudar assuntos relacionados a documentação e a avaliação de arquitetura de software;
2. Estudar e documentar a arquitetura de uma API de um sistema Web utilizado como estudo de caso;
3. Executar avaliações baseadas em análise estática e dinâmica da API de um sistema Web para coletar métricas sobre a sua qualidade e identificar vulnerabilidades;
4. Propor melhorias na arquitetura do sistema do estudo de caso com base nas informações levantadas pelas análises.

1.3 Estrutura da monografia

O Capítulo 2 contém a fundamentação sobre a arquitetura de software, arquitetura de sistemas Web e sobre a linguagem utilizada no código alvo do estudo, de forma que apresenta os conceitos e definições que as englobam. Em seguida, o Capítulo 3 apresenta a metodologia de avaliação de arquitetura que foi aplicada neste trabalho. A avaliação e os resultados foram juntados e apresentados no Capítulo 4. E,

por fim, o Capítulo 5 apresenta uma conclusão com uma visão geral do estudo com sugestão de trabalhos futuros.

2 CONCEITOS GERAIS E REVISÃO DA LITERATURA

Esse tópico irá descrever os principais conceitos que envolvem a arquitetura de software, que é a base do trabalho. Em seguida, é discutido sobre a arquitetura de Sistemas Web, com foco em camadas e microsserviços. Logo depois, serão apresentados os principais conceitos de uma API (do inglês, *Application Programming Interface*) que trabalha com REST (do inglês, *Representational State Transfer*). Por fim, será discutido sobre a linguagem de programação que foi estudada e utilizada.

2.1 Arquitetura de Software

De acordo com Martin Fowler (2019), apesar de difícil definição, arquitetura de software pode ser considerada como coisas importantes, o que quer que seja. Isso significa que pensar em arquitetura de software é decidir o que é importante e procurar manter esses elementos arquiteturais em boas condições. Outras definições surgiram com o tempo, como a de Perry e Wolf (1992), a de Garlan e Shaw (1994), a de Bass et al (2003) e a do padrão IEEE 1741 (2000), que apresentaram a importância de conhecer as etapas compõem o projeto da arquitetura de software.

Segundo Germoglio (2010), existem algumas etapas que propõem um esqueleto de uma arquitetura de software, começando pelo design de software, depois pela definição dos aspectos da arquitetura, os atributos de qualidade, as técnicas de design arquitetural e finalmente, a documentação da arquitetura.

2.1.1 Design de Software

Na primeira etapa da arquitetura de software se encontra o Design de Software, que pode ser definido como a projeção de um sistema, que vai desde a definição da arquitetura até o resultado do processo de design de software. Este processo de design pode ser descrito como o processo de escolha da representação de

uma solução a partir de várias alternativas, dadas as restrições que um conjunto de objetivos envolve (GERMOGLIO, 2010).

Ainda de acordo com Germoglio (2010), apesar do produto gerado pelo processo do design de software ser sempre a solução do design, é necessário saber o nível do detalhamento da solução, que, definidas como o nível de design, podem ser classificadas como:

- Design de alto nível, ou design arquitetural, que descreve como o software é decomposto e organizado em módulos e suas relações;
- Design detalhado, que procura descrever detalhadamente os módulos do design arquitetural.

2.1.2 Fundamentos da Arquitetura de Software

Os fundamentos da arquitetura são compostos de conceitos divididos em partes que definem os aspectos existentes da arquitetura utilizados atualmente, derivados das definições da arquitetura de software. Existem dois aspectos principais que são usados para alcançar os atributos de qualidade, sendo o primeiro o de elementos arquiteturais, que definem como o software é particionado em pedaços menores para um melhor entendimento, podendo estes elementos serem estáticos ou dinâmicos, e o segundo aspecto são as decisões arquiteturais, que são as decisões de estrutura do projeto em meio às alternativas de design arquitetural para alcançar um ou mais atributos de qualidade (GERMOGLIO, 2010).

Outras características importantes que envolvem a arquitetura de software se resumem em:

1. Visão arquitetural, onde este propõe que haja uma representação do sistema adequado para cada stakeholder, uma vez que podem existir várias pessoas interessadas na arquitetura que possuem conhecimentos e interesses diferentes;
2. Rastreabilidade, ou seja, a ligação dos requisitos do sistema com as estruturas arquiteturais;

3. A evolução do sistema, que se faz a partir de um conjunto de regras de forma que o sistema possa crescer enquanto se mantém fiel ao seu objetivo.

2.1.3 Atributos de Qualidade

Durante o desenvolvimento de um projeto de software se encontram os atributos de qualidade, que são propriedades de qualidade do software ou do seu ciclo de desenvolvimento que se relacionam com os objetivos do projeto que já foram alcançados, diferente dos requisitos de software, que garantem que o sistema tenha as funcionalidades desejadas pelo cliente e que funcione adequadamente. Portanto, um software deve exibir atributos de qualidade que atendam aos seus requisitos. (GERMOGLIO, 2010).

Dado que o processo de desenvolvimento de software possui tempo e recursos finitos, é que se originaram os modelos de qualidade, que facilitam essas avaliações durante o processo. A partir deste conceito, surgiram alguns modelos que se tornaram bastante significativos na comunidade, que são conhecidos como o modelo de Boehm, o de McCall e principalmente o do padrão ISO/IEC 9126-1:200, onde este definiu que os atributos que um software deve possuir para que seja considerado de qualidade são: funcionalidade, confiabilidade, usabilidade, eficiência, manutenibilidade e portabilidade.

2.1.4 Técnicas de Design Arquitetural

O penúltimo conceito da arquitetura envolve as técnicas de design arquitetural, que, de acordo com Germoglio (2010), quando aplicados, geralmente resultam em boas soluções de design. Dentre essas técnicas, as mais utilizadas são: divisão e conquista, abstração, encapsulamento, modularização, separação de preocupações, acoplamento e coesão, e separação de interfaces de suas implementações.

Existe outra forma de reuso de experiência de design e que não é propriamente definida como um padrão, conhecida como táticas de design, que normalmente contém ideias e dicas de projeto que ajudam na implementação de atributos de qualidade, mas, caso utilizadas, é necessário que haja uma atenção especial com os trade-offs existentes, pois melhorar um atributo pode afetar negativamente outro. Ainda de acordo com Germoglio (2010), algumas táticas de design se relacionam diretamente com os atributos de qualidade, como por exemplo, táticas como não manter o estado de um elemento da arquitetura, partição de dados e caching podem melhorar o desempenho, mas aumenta a complexidade da arquitetura.

2.1.5 Documentação da Arquitetura

O último processo da arquitetura de software trata da documentação da arquitetura, que proporciona vários benefícios, como integridade conceitual entre os stakeholders e da integridade entre visões da arquitetura, forma de comunicação entre os diferentes stakeholders e um auxílio no processo de design. No entanto, existem também dificuldades em documentar, como o fato do tamanho do documento refletir o tamanho da solução e a sua complexidade, além de ser custoso manter o documento consistente com o design atual (GERMOGLIO, 2010).

Para que seja possível documentar o design de arquitetura é necessário que alguns conceitos sejam definidos previamente, como quem são os stakeholders, que são os que definirão quais os pontos de vista que vão ser usados no documento, e então registrar as decisões arquiteturais, que descrevem o design em visões derivadas a partir dos pontos de vista escolhidos (GERMOGLIO, 2010). Apesar de ser mencionado como último processo, é comum a documentação ser feita em paralelo com o design da arquitetura, uma vez que ambos têm objetivos em comum, e são dependentes do processo e da equipe de desenvolvimento.

2.2 Arquitetura de Sistemas Web

A arquitetura de sistemas Web evoluiu consideravelmente desde o começo da internet (ROESLER et al, 2020). De acordo com Hunter e Crawford (2001), mencionado por Roesler et al (2020), no começo, os sistemas eram desenvolvidos usando arquitetura CGI (do inglês, *Common Gateway Interface*), no qual deu muito poder aos servidores, já que começou a oferecer a funcionalidade de executar scripts quando processando requisições HTTP, fazendo com que os sistemas Webs fossem capaz de processar as requisições de forma mais dinâmica.

No entanto, havia ainda nessa época, problemas que dificultavam a vida dos desenvolvedores, como produzir código de interface e de lógica de negócios de aplicações Webs, de formas separadas (ROESLER et al, 2020). A partir dessa necessidade, começaram a surgir várias arquiteturas, como a arquitetura de 'n' camadas, arquitetura orientada a serviço, monolítica, e de microsserviços.

2.2.1 Arquitetura de N camadas

Tendo em vista o problema anterior, apresentado segundo Roesler et al (2020), a primeira arquitetura a aparecer como solução, foi a de 2 camadas, também conhecido como arquitetura cliente - servidor. Na arquitetura cliente - servidor, uma aplicação é modelada como um conjunto de serviços que são fornecidos pelo servidor, onde clientes podem acessar estes serviços e apresentar resultados para os usuários, além de ter conhecimento dos servidores, mas não podem acessar outros clientes, pois clientes e servidores são processos separados (SOMMERVILLE, 2016).

Ainda de acordo com Sommerville (2016), a arquitetura de 2 camadas é a versão simplificada da arquitetura cliente - servidor: o sistema é implementado como um único servidor lógico mais um número indefinido de clientes que usam o servidor. Atualmente existem duas formas desse modelo arquitetural, como mostra a Figura 2.1:

- Modelo cliente - magro: Nesse modelo, apenas a camada de apresentação é

implementada no cliente, e todas as outras camadas permanecem no servidor;

- Modelo cliente - gordo: Nesse modelo alguns ou todo o processamento da aplicação é realizado no cliente, enquanto o gerenciamento e o banco de dados são implementados no servidor.

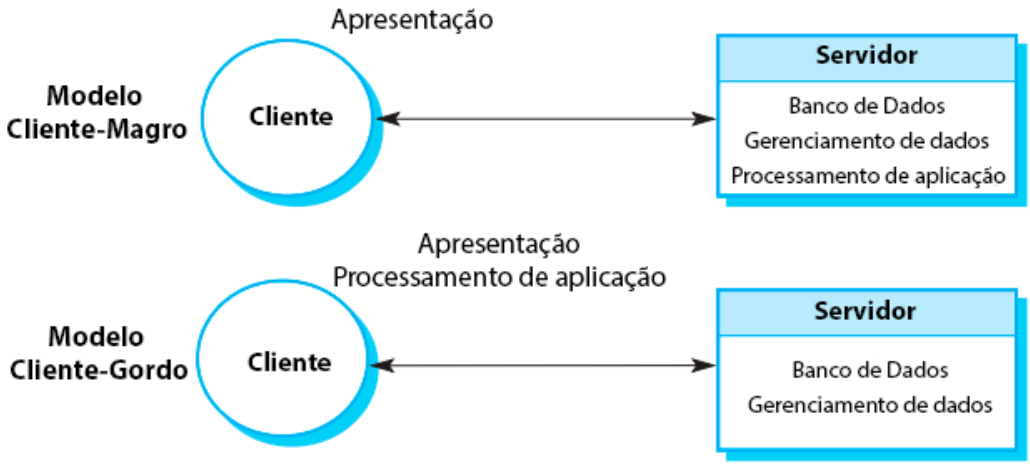


Figura 2.1: Ilustração dos modelos arquiteturais de 2 camadas

Fonte: Sommerville (2016)

O problema fundamental com a arquitetura de 2 camadas é que as camadas lógicas do sistema devem ser mapeadas tanto no cliente como no servidor, e isto pode levar a problemas de escalabilidade e performance ou até problemas de gerenciamento de sistema. De acordo com Sommerville (2016), para evitar essas situações, é possível usar uma arquitetura de 3 camadas, como mostrado na Figura 2.2, onde as camadas do sistema dentro desta arquitetura são processos separados que podem ser executados em diferentes processadores. O ponto mais importante desse modelo é a divisão da lógica de apresentação entre sua própria camada lógica e física. Essa separação em camadas lógicas permite sistemas mais flexíveis, de forma que é possível alterar as partes de forma independente. Esse modelo de 3 camadas pode ainda ser estendido a uma variação de N camadas ou multi camadas, onde novos servidores são adicionados ao sistema (SOMMERVILLE, 2016).

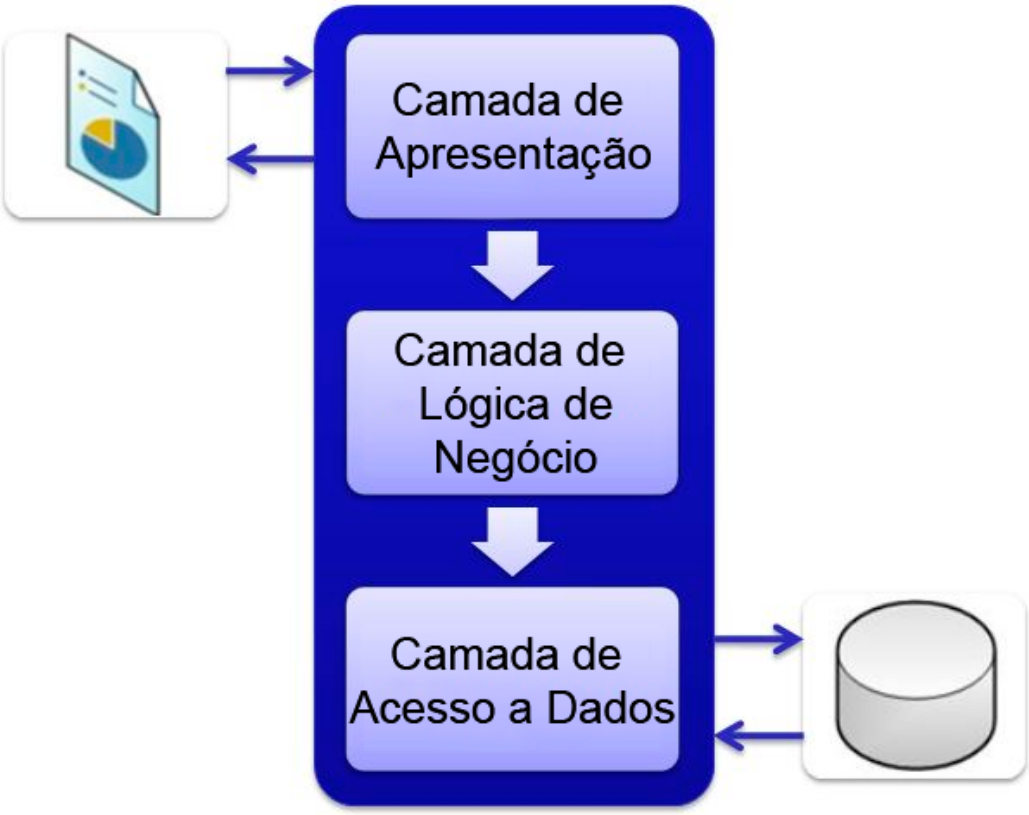


Figura 2.2: Ilustração do modelo arquitetural de 3 camadas

Fonte: <https://www.codeproject.com/Articles/36847/Three-Layer-Architecture-in-C-NET-2>

2.2.4 Arquitetura Orientada a Serviço e de Microsserviços

Uma aplicação monolítica, de acordo com Finnigan (2018), é uma aplicação que tem todos os seus componentes dentro de um único implantável, normalmente não respeita os limites das funcionalidades e tem uma frequência de liberação de 3 a 18 meses. Esse tipo de aplicação tem muitos problemas de performance e escalabilidade quando muitos usuários estão a usá-lo, e, devido ao seu conceito de agrupamento de dependências entre componentes, em algum momento ocorrerá o problema da necessidade de atualizar um único componente (ROESLER et al, 2020). A solução foi encontrada em uma arquitetura menos monolítica e mais distribuída: a

SOA (do inglês, *Service-Oriented Architecture*).

A arquitetura orientada a serviço é um estilo de arquitetura baseado na ideia de que serviços executáveis podem ser incluídos em aplicações (SOMMERVILLE, 2016). A Figura 2.3 demonstra a estrutura de uma SOA: provedores de serviços projetam e implementam serviços, além de especificar suas interfaces, e também publicam informações sobre os mesmos em um registro acessível. Os requisitores (ou clientes) dos serviços descobrem as especificações dos serviços que pretendem usar, localizam o fornecedor e então podem ou não prender a sua aplicação a um serviço específico, normalmente utilizando protocolos de serviço, que permitem uma comunicação entre eles.

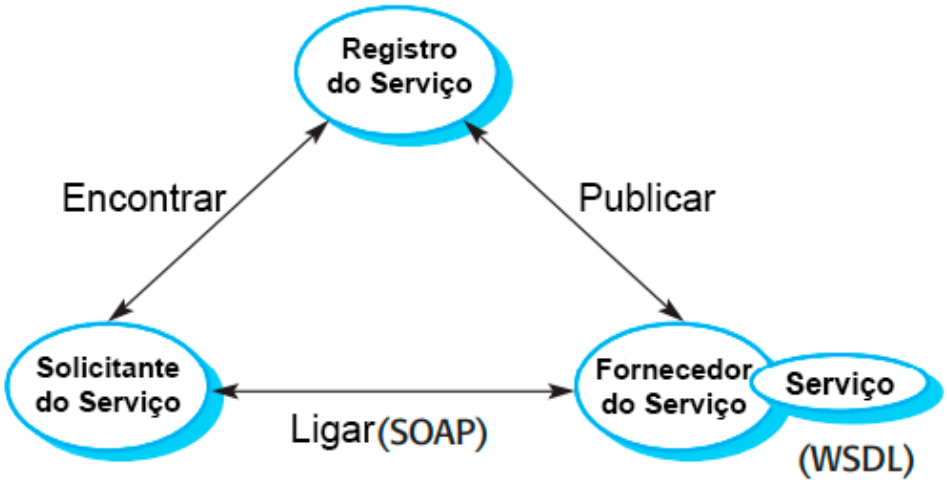


Figura 2.3: Ilustração da arquitetura SOA

Fonte: Sommerville (2016)

Um microsserviço consiste de uma única implantação em execução dentro de um único processo, isolado das outras implementações e processos que apoiam o cumprimento de uma parte específica da funcionalidade do sistema, ou seja, cada microsserviço foca em realizar a única tarefa que lhe foi instruída dentro de um

contexto limitado (FINNIGAN, 2018). Apesar de não ter uma definição precisa, a arquitetura de microsserviços, como demonstrado na Figura 2.4, é uma abordagem para desenvolver uma única aplicação como um pacote de pequenos serviços, onde cada serviço funciona no seu próprio processo e se comunicam utilizando mecanismos leves, onde geralmente é uma API (FOWLER, 2014).

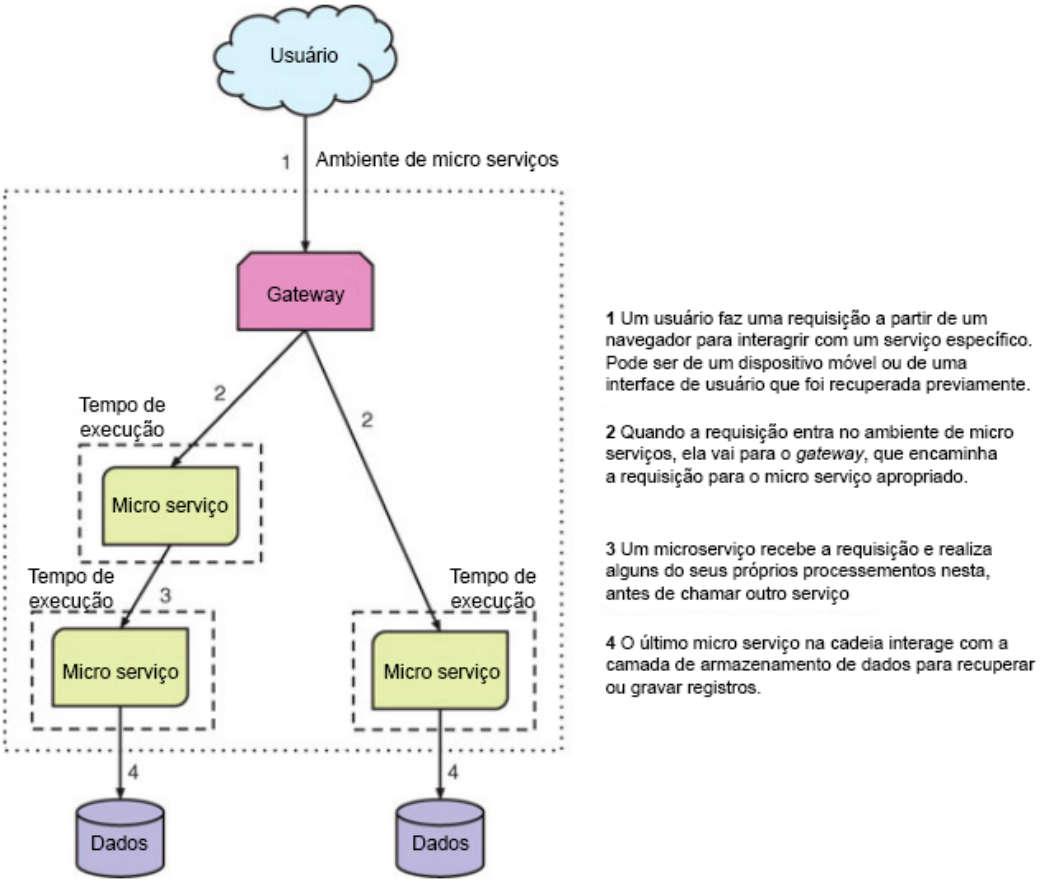


Figura 2.4: Representação de uma arquitetura de microsserviços

Fonte: Finnigan Ken (2018)

Como não existe uma definição formal para a arquitetura de microsserviços, é apropriado se basear nas características que são denominadas comuns pela comunidade, de forma que se enquadram a esse tipo de arquitetura, como:

1. Permitir construir sistemas ao juntar componentes por meio de serviços;
2. Ser divido em serviços organizados em torno de capacidades de negócios;

3. Procurar fazer produtos ao invés de projetos;
4. *Endpoints* mais inteligentes;
5. Controles e gerenciamento de dados descentralizados;
6. Infraestrutura automatizada;
7. Tolerar falhas de serviços;
8. Design evolucionário.

2.3 API

APIs são um conjunto de código de programação que permite com que haja transmissão de dados entre softwares diferentes, e que também contém os termos necessários para que possa ser realizada essa troca (ALTEXSOFT, 2019). Como apresentado na Figura 2.5, quando um cliente precisa acessar uma informação ou funcionalidade que se encontra em um servidor, ele faz uma requisição para uma API, que funciona como uma interface de comunicação entre os dois sistemas, enquanto especifica como os dados devem ser passados, e então o servidor retorna essas informações solicitadas de forma apropriada.

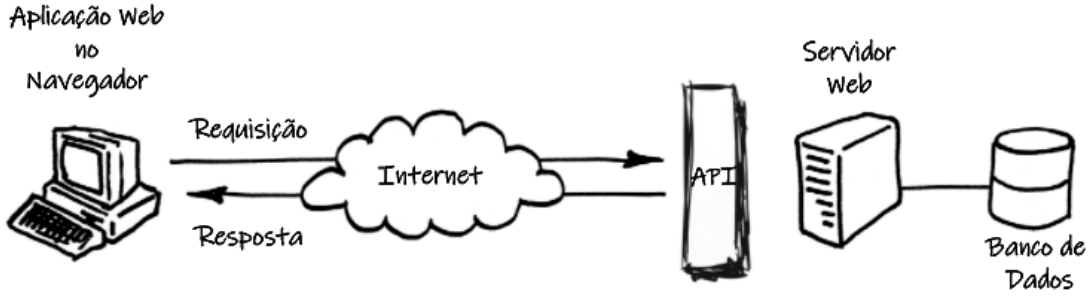


Figura 2.5: Demonstração de como funciona uma API

Fonte:
<https://www.altexsoft.com/blog/engineering/what-is-api-definition-types-specifications-documentation/>

De acordo com a Altexsoft (2019), APIs atendem a vários propósitos, como

simplificar e acelerar o desenvolvimento de um software, adicionar funcionalidades extras em soluções já existentes, ou criar novas aplicações usando serviços fornecidos por terceiros. Em todo caso, não há necessidade de lidar com o código fonte para tentar entender como a solução funciona, bastando apenas conectar uma aplicação a outra.

Existem também as especificações de uma API, que tem como objetivo padronizar a troca de dados e informações entre serviços Web, garantindo que sistemas diferentes com tecnologias diferentes possam se comunicar (ALTEXSOFT, 2019). Dentre os tipos de Web API existentes, alguns são mais conhecidos e utilizados pela comunidade, como o RPC (do inglês, *Remote Procedure Call*), SOAP (do inglês, *Simple Object Access Protocol*), GraphQL e principalmente o REST, na qual foi utilizado neste trabalho.

2.3.1 REST

A API REST é um conjunto de princípios que definem como os *Web Standards* (do inglês, padrões Web) como HTTP e URI (do inglês, *Uniform Resource Identifier*) devem ser usados, ou seja, usar REST em um projeto permite ter um sistema que explora a arquitetura da Web em seu benefício (TILKOV, 2008). Ela consiste de seis restrições arquiteturais escolhidas pelas propriedades que induzem nas arquiteturas candidatas (ROY T. FIELDING, 2000), que são conhecidas como: Cliente-Servidor, sem estado (*stateless*), cache, interface uniforme, sistema em camadas e *code-on-demand* (do inglês, código sob demanda).

O REST é uma abstração dos elementos arquitetônicos dentro de um sistema de hipermídia distribuído (ROY T. FIELDING, 2000). Ele é considerado um estilo de arquitetura de alto nível que poderá ser implementado utilizando várias tecnologias diferentes, e instanciado utilizando diferentes valores para suas propriedades abstratas (TILKOV, 2008), como por exemplo, o HTTP, que "instancia" a interface uniforme do

REST com uma interface especial, consistindo nos verbos HTTP.

Às vezes confundido com REST e usado erroneamente, o termo RESTful é muito usado na comunidade Web. Para que um sistema seja reconhecido como RESTful, é necessário que ele seja capaz de implementar os princípios de REST, ou seja, precisa conter obrigatoriamente as seis restrições arquiteturais, e também é necessário que implemente o HATEOAS (do inglês, *Hypermedia as the Engine of Application State*), que é uma restrição que ajuda os clientes a utilizarem a API sem a necessidade de ter um conhecimento aprofundado sobre a mesma. Ao implementar o HATEOAS, a API passa a disponibilizar links que mostrarão aos clientes como navegar através dos seus recursos, ou seja, basta que ele tenha conhecimento da URL(do inglês, *Uniform Resource Locator*) inicial e, partir dos links oferecidos, poderá acessar todos os recursos da API, se guiando através das requisições realizadas.

2.4 Linguagem Ruby

A linguagem Ruby foi criada por Yukihiro Matsumoto, no ano de 1995, no Japão. Ela tinha como objetivo ser uma linguagem mais legível e agradável de se programar, mas, além das características orientada a objetos, Ruby também foi criada para ser uma linguagem funcional, tendo recursos poderosos e essenciais desse paradigma, como lambdas e *closures* (SOUZA LUCAS 2020), e além disso, é possível citar que Ruby é uma linguagem de tipagem dinâmica, fortemente tipada, orientada a objetos, e é uma linguagem interpretada.

Nos últimos anos a linguagem progrediu bastante, principalmente com a origem do *rubygems*, um serviço de hospedagem de *gems*, que são trechos de código como uma biblioteca, criados por terceiros para serem reutilizados, e que auxilia a comunidade dos desenvolvedores ruby.

De acordo com Souza Lucas (2020), é muito comum encontrar a linguagem

Ruby sendo usada para a criação de scripts para ler e processar arquivos, automatizar *builds* e *deploys*, e fazer *crawling* de sites. Mas, o maior destaque da linguagem Ruby está ligado à aplicações Web, onde o caso de maior sucesso, e que também é a linguagem utilizada neste trabalho, é o Ruby on Rails.

2.4.1 Ruby on Rails

Ruby on Rails (ROR, ou apenas "Rails", como é conhecido) é um framework open source para desenvolvimento de aplicações Web, criado por David Heinemeier Hansson. O framework, que foi escrito na linguagem Ruby, foi extraído de um produto de sua empresa, o Basecamp, em 2003. Desde então, a fama desta linguagem cresceu bastante, levando também a linguagem Ruby, anteriormente apenas conhecida no Japão e em poucos lugares dos Estados Unidos, ao mundo todo (FUENTES VINICIUS 2017).

O framework do ROR usa a estrutura MVC - *Model View Controller* (do inglês, modelo visão controlador), padrão de projeto de software que foca no reuso de código, além de separar o front do back-end. Ele também já é usado por muitos desenvolvedores há bastante tempo, e já foi testado em várias situações, comprovando que aguenta alta carga e grande número de usuários.

Existem vários componentes dentro do framework relacionados ao MVC que facilitam estruturar a arquitetura de uma aplicação ROR, como: *ActiveRecord, ActiveModel, ActionController, ActionView* e *Asset Pipeline*, onde cada componente tem funções específicas para separar e organizar os dados da aplicação. Além dessas características, a linguagem contém um conjunto pilares que constituem as doutrinas do Rails. De acordo com as doutrinas apresentadas no site oficial do ROR[1], existem nove pilares que são considerados mais importantes:

1. Otimize para felicidade do programador: Esta doutrina está ligada diretamente

[1] https://rubyonrails.org/doctrine/

com a linguagem Ruby, que valorizou coisas diferentes de outras linguagens, pensando em acomodar e elevar os sentimentos do programador, ou seja, garantir a felicidade em programar. O princípio desta doutrina é de fazer o criador da linguagem, e a quem concordar com ele, feliz em usá-la.

2. Convenção sobre configuração: Esta segunda doutrina afirma que ao abrir mão de individualidades vãs, é possível desviar da fadiga das decisões mundanas, de forma que seja possível fazer um progresso mais rápido na área que realmente importa, como por exemplo, entender como utilizar melhor um framework pessoal.

3. O menu é *omakase*: Derivado da doutrina 2, esta doutrina se preocupa em fornecer um bom conteúdo ao programador sem que ele tenha conhecimento vasto e também que não dependa da sorte; Rails procura utilizar *frameworks* e bibliotecas que podem ser alteradas, mas que não é necessário, de forma que seja mantido um padrão para todos os programadores, o que facilita no aprendizado e na experiência.

4. Nenhum paradigma: Rails é uma composição de muitas ideias diferentes e até mesmo de paradigmas, mas sem que entrem em conflito, o que permite com que a linguagem seja bem mais flexível, ou seja, um framework muito melhor que qualquer outro paradigma individual. No entanto, é necessário conhecer não só programação orientada a objeto, mas também ter experiência com programação imperativa e funcional.

5. Exalte um código bonito: Essa doutrina afirma que a beleza do código não se encontra apenas em manter um código curto e poderoso, mas procurar manter um fluxo de declarações, de forma que mantenha um sentido mais preciso e mais claro para quem está lendo.

6. Fornecer facas afiadas: Essa sexta doutrina afirma que tanto na linguagem Ruby como em RoR possuem várias ferramentas à disposição do programador, para que ele possa ter uma liberdade maior e se aprofundar mais na linguagem,

mesmo que essa liberdade possa ser perigosa ao código, dependendo de como serão usadas.

7. Sistemas integrados de valor: A sétima doutrina afirma que Rails foi projetado inicialmente para fazer sistemas integrados, os monolíticos, de forma que endereça um sistema inteiro para resolver todo o problema. A ideia é fazer sistemas integrados, que contém mais vantagens do que deixar tudo distribuído, pois, dessa forma, teremos todo o poder das aplicações individuais e distribuídas com um sistema único de fácil uso e de fácil entendimento.

8. Progresso em vez de estabilidade: A ideia desta doutrina é de encarar mudanças para que o Rails continue a crescer e se manter de pé, com atualizações do *framework*, mesmo que aconteça problemas com versões anteriores, pois é assim que os desenvolvedores procuram aprender mais para resolver novos problemas e seguir em frente.

9. Subir uma grande tenda: A última doutrina informa que Rails está sempre disposto a receber novas pessoas, de forma que a proposta inicial a qual foi originada permaneça a mesma, o que não impede que haja novas funcionalidades ou alterações, mas, ainda assim, é necessário trabalhar para que seja bem receptiva para os novos integrantes.

2.5 Modelo C4

O modelo C4 foi desenvolvido com intuito de auxiliar desenvolvedores na comunicação e descrição da arquitetura de um software, seja durante o início do projeto como quando estiver documentando o código, pois é criado um mapa deste, em vários níveis de detalhe (BROWN, 2018). O nome C4 refere-se aos quatro diagramas utilizados dentro da hierarquia do modelo, respectivamente:

1. Diagrama de Contexto;
2. Diagrama de Container;
3. Diagrama de Componente;
4. Código.

Esse modelo usa uma abordagem de abstração, de forma que cada diagrama amplia e detalha algum elemento do diagrama anterior, como é ilustrado na Figura 2.6.

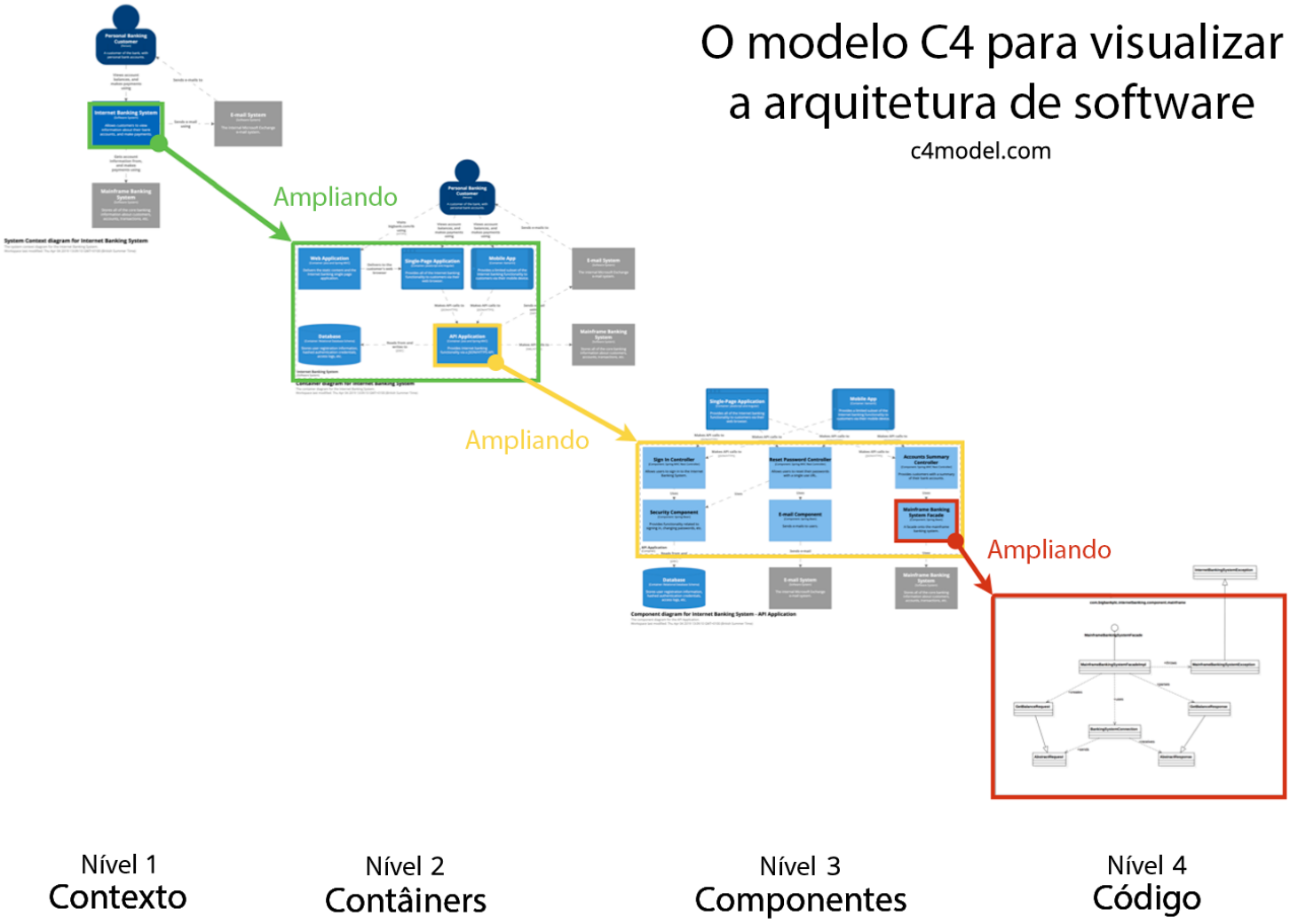


Figura 2.6: Representação dos diagramas do modelo C4

Fonte: <https://c4model.com/>

2.3.1 Diagrama de contexto do sistema

O diagrama de contexto do sistema possui o maior nível de abstração que está disponível no modelo. Aqui é representado o ambiente em que o sistema é presente. Dessa forma, é colocado um retângulo no meio representando o sistema em questão, cercado pelos outros sistemas e usuários com quais ele interage. A ideia nesse nível é focar não no sistema principal, mas em quem usa e com o que é usado. Pelo seu baixo nível de detalhe, esse diagrama pode ser compreendido por todos interessados.

2.3.2 Diagrama de container

No segundo nível do modelo, o diagrama apresenta containers, que definem as fronteiras nas quais códigos são executados ou representam algum tipo de armazenamento de dados. Cada container funciona como uma unidade que é executada e implementada separadamente, como, por exemplo, uma aplicação Web server-side, um microsserviço ou uma base de dados.

O diagrama de container amplia o sistema e mostra, em alto nível, a organização da arquitetura, de forma que é possível reconhecer como estão distribuídas as responsabilidades dentro do sistema, além de identificar as principais tecnologias utilizadas e o modo de comunicação entre containers. Por conta disso, é necessário um conhecimento técnico para compreender o diagrama, que atinge um público dentro e fora do time de desenvolvimento.

2.3.3 Diagrama de componente

No terceiro nível, cada container pode ser decomposto em um conjunto de

componentes, onde cada componente pode ser entendido como um agrupamento de código relacionado a uma mesma funcionalidade que é encapsulado por uma única interface. O foco desse diagrama é atender as necessidades do time de desenvolvimento, uma vez que é possível ver o número de componentes, o que são, qual tecnologia é usada, como se relacionam e detalhes da implementação.

2.3.4 Nível de código

No último nível da hierarquia deste modelo, pode-se ampliar um componente para observar detalhes de implementação do código, como as classes, interfaces, funções e tabelas do banco de dados. Este nível é considerado opcional, pois, o fato do código estar em constante mudança faz com que um diagrama anteriormente criado se torne facilmente obsoleto em pouco tempo, logo, só é recomendado usá-lo para os componentes mais complexos. Normalmente essa etapa se torna mais fácil ao utilizar ferramentas que automatizam o trabalho, como diagramas de classe UML (do inglês, *Unified Modeling Language*) e diagramas de entidade relacionamento baseado no código fonte.

3 METODOLOGIA

Esse trabalho é uma continuação da monografia realizada por Rodolffo (2020), ou seja, a pesquisa é oriunda do mesmo princípio. Em sua monografia, ele estudou sobre dois microsserviços de uma empresa, enquanto neste trabalho será feito uma avaliação de uma API da mesma empresa. Logo, este trabalho também segue a mesma metodologia que o Rodolffo utilizou em sua proposta, que é apresentada nas linhas a seguir.

3.1 Metodologia de Avaliação

Neste trabalho foi utilizado uma metodologia baseada nos conceitos propostos por Guimarães e Nogueira (2008). Ela é dividida em cinco fases principais:

1. A apresentação do sistema;
2. Definição de objetivos;
3. Avaliação da arquitetura;
4. Projeto de manutenção;
5. Implementação de mudanças.

Durante o processo de aplicação dessa metodologia, é necessário que haja repetições de ciclos de atividades até chegar em uma arquitetura adequada, como mostrado na Figura 3.1. Devido a limitação do escopo, o foco do estudo de caso deste trabalho foi apenas nas atividades das fases 3) e 4).

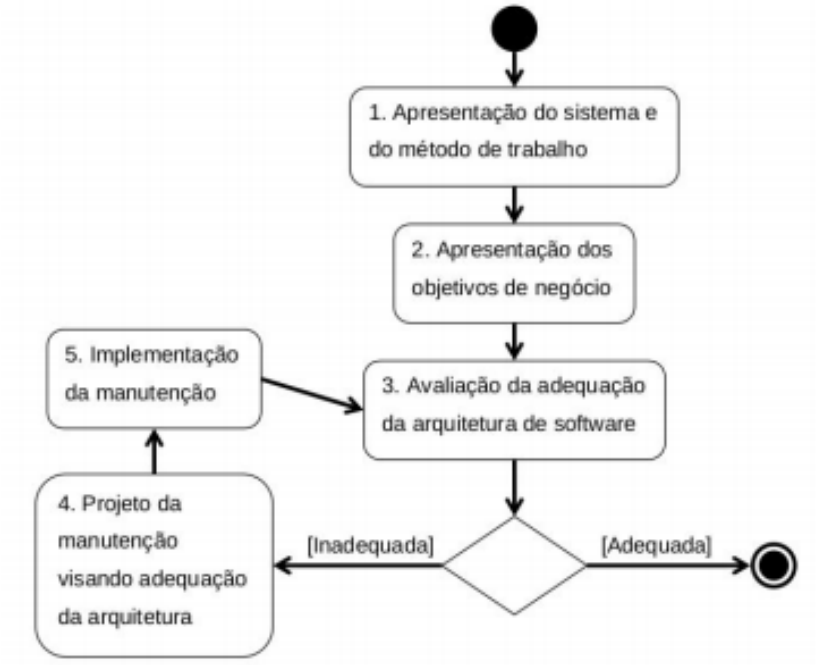


Figura 3.1: Diagrama do processo do método de manutenção

Fonte: Guimarães e Nogueira (2008)

3.1.1 Fase 1: Apresentação do sistema e do método de avaliação

Na primeira fase, a equipe que fará a avaliação obtém as primeiras impressões do sistema que será analisado. A apresentação do sistema é conduzida por alguém já experiente com a arquitetura implementada e que pode repassar artefatos, como a documentação do sistema, e também faz uma demonstração do sistema em funcionamento. A ideia principal nesta fase é que a equipe conheça as funcionalidades que o sistema oferece e o ambiente a qual ele pertence.

Além dessa apresentação, nesta etapa também é repassado o método de avaliação que será utilizado. A equipe de avaliação detalha como ocorrerá o processo, separando as atividades que serão realizadas em cada fase e alegando quais artefatos serão gerados a partir dessa avaliação.

3.1.2 Fase 2: Apresentação dos objetivos de negócios

Nesta fase, a equipe de avaliação toma conhecimento dos objetivos de negócios que foram usados para a criação do sistema. É a partir desses objetivos que são definidos quais os atributos de qualidade que irão formar a arquitetura do sistema. Essa etapa é essencial, uma vez que o objetivo principal da avaliação do sistema é verificar se a arquitetura atual implementada é capaz de atender a todos os atributos de qualidade necessários.

Dentre várias técnicas existentes para descrever os objetivos de negócios, a mais comum é o uso de cenários. Nesse contexto, são concedidos para cada atributo de qualidade um conjunto de cenários, podendo ser uma descrição de uma situação com uma persona, ou usando o diagrama de casos de uso UML, de forma que demonstram situações onde possíveis clientes usariam o sistema. O uso dos cenários permite identificar e definir problemas a serem resolvidos e novas funcionalidades a serem implementadas.

3.1.3 Fase 3: Avaliação de adequação da arquitetura

O foco desta fase é realizar a avaliação da arquitetura e verificar se ela está atendendo aos atributos de qualidade definidos anteriormente. Para alcançar este objetivo, é muito importante utilizar métodos de avaliações estáticas e dinâmicas no software em análise para levantar métricas que são utilizadas para caracterizar o sistema e identificar problemas existentes no código.

3.1.4 Fase 4: Projeto de manutenção

Após a avaliação, caso seja identificado que a arquitetura do sistema não atende aos objetivos de negócios, será necessário preparar uma especificação com as alterações que precisarão ser feitas no sistema para que seja possível realizar as correções da arquitetura, de forma que o sistema atenda a todos os objetivos.

Nesta quarta fase, a equipe de avaliação discute a forma que a arquitetura deve ter e, levando em conta os impactos positivos e negativos das alterações a serem implementadas, define qual será a melhor solução para a situação. É importante ter conhecimento dos padrões arquiteturais durante este processo, pois, facilita a escolha da solução, uma vez que esses padrões são bastante conhecidos.

3.1.5 Fase 5: Implementação da manutenção

Na quinta e última fase é onde ocorre a implementação das mudanças propostas na fase 4. Para garantir um bom resultado, é necessário que a implementação seja rigorosa com as especificações. Além disso, é importante também aderir a boas práticas de programação e procurar utilizar testes com frequência. Ao término dessa fase, o sistema terá uma nova arquitetura, e assim será necessário que seja reavaliada, voltando para a fase 3, e repetindo este ciclo até que seja alcançado o resultado adequado.

4 ESTUDO DE CASO

Como esse trabalho é uma continuação da monografia de Rodolffo (2020), este capítulo é composto de forma semelhante a que se encontra em sua proposta, apresentando diferentes níveis da arquitetura do sistema do objeto de estudo, as métricas e vulnerabilidades coletadas através da análise estática e dinâmica e discutindo as propostas de mudanças que teriam impacto positivo na manutenibilidade e segurança do sistema.

4.1 Sistema Overmedia

A Overmedia[2] é uma startup brasileira da cidade de São Paulo que trabalha com vídeos que permitem interação do usuário para que ele possa fazer determinadas escolhas em tempo real. Essa tecnologia denominada "video.bot" é a junção da inteligência de um bot com voz e apelo visual de um vídeo, que tem o intuito de melhorar a experiência digital de clientes e colaboradores que a utilizarem. O sistema da Overmedia segue, arquiteturalmente, o padrão de microsserviços, além de usar APIs envolvendo sistemas Web, logo, é um excelente candidato como objeto de estudo.

4.1.1 Visão geral do sistema (Overmedia) - C1

Como é possível observar no diagrama de contexto na Figura 4.1, o sistema Overmedia atende a dois tipos diferentes de clientes, sendo um o usuário comum ou usuário final, no qual se refere às pessoas para qual o sistema foi projetado, que consegue acessar e interagir com os "video.bot", e o outro usuário é o gerente, que tem acesso as configurações do "video.bot", onde este possui permissões para baixar os dados que foram coletados, gerar emails e SMS (do inglês, *Short Message Service*) para determinados usuários e obter vários tipos de relatórios que ajudam a entender a

[2] Overmedia: Disponível em: https://overmediacast.com/

forma como os usuários estão interagindo com o "video.bot".

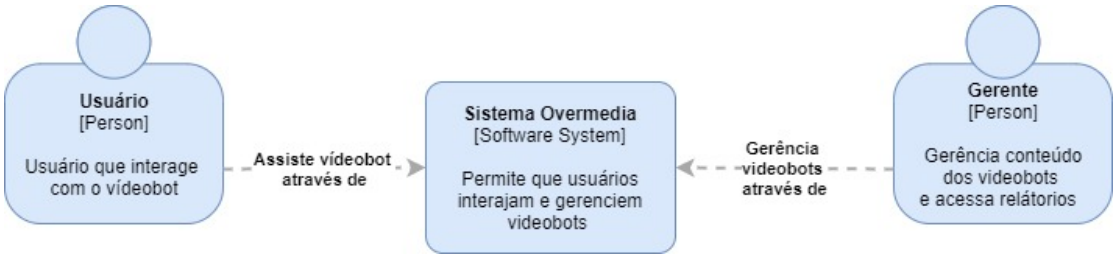


Figura 4.1: Diagrama de contexto do sistema Overmedia

Fonte: Imagem adaptada de Rodolffo (2020)

4.1.2 Visão de alto nível - C2

O sistema da Overmedia pode ser decomposto em diversos containers, como demonstrado no diagrama da Figura 4.2. O James Browser é um player de vídeo utilizado pela empresa que suporta interações com um “video.bot” feitas pelos usuários. Essas interações só são possíveis devido ao uso da `“API Overmedia”`, que é quem define como um “video.bot” deve responder as requisições feitas. Cada interação feita, independente de qual seja, é codificada em um evento e transmitida ao `“Tracking Server”` para ser registrada nas bases de dados existentes. Esses dados são, em seguida, utilizados pelo `“Tracking Report”` que gera relatórios com informações apropriadas, relacionadas aos acessos realizados aos “video.bot”. Por fim, um gerente consegue acessar esses relatórios através da aplicação Web `“Plataforma de Gerência”`.

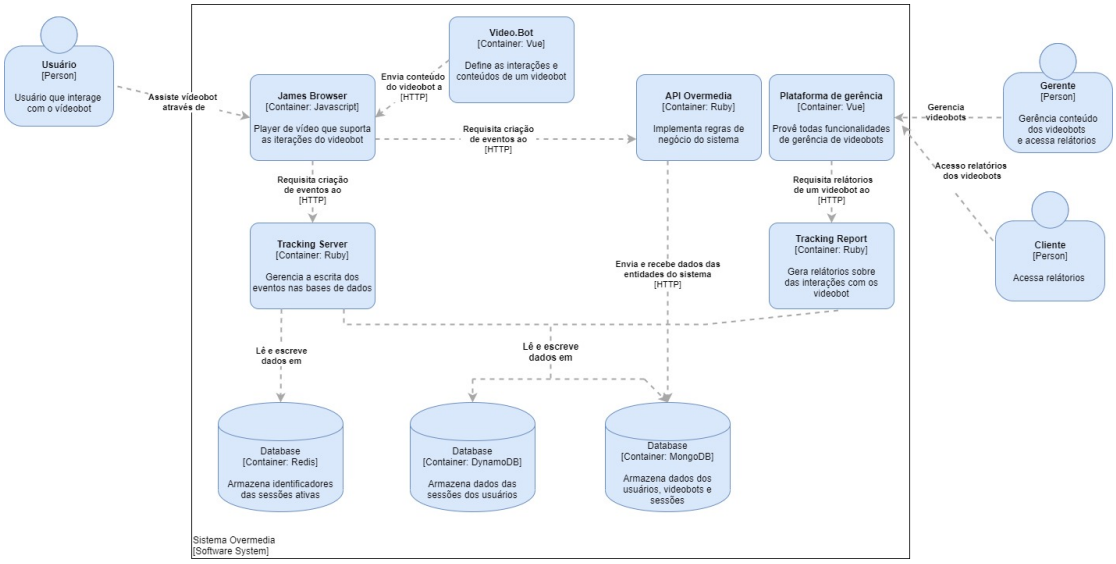


Figura 4.2: Diagrama de container do sistema Overmedia

Fonte: Imagem adaptada de Rodolffo (2020) e modificada pelo autor

4.1.3 Visão de baixo nível - C3 (Overmedia API)

Nessa seção é apresentado o diagrama de arquitetura da API que foi o foco do estudo de caso. Essa visão de baixo nível permite compreender a lógica e fluxo de execução desta API, além de esclarecer como é feita a distribuição de responsabilidades.

O container "`Overmedia API`" apresenta as regras de negócios no sistema, ou seja, a implementação dos comportamentos relacionados ao "vídeo.bot" e das marcas ou empresas envolvidas. Na Figura 4.3 é possível observar a organização geral dos componentes que compõem a API e, nos parágrafos seguintes, serão detalhadas as responsabilidades de cada um deles.

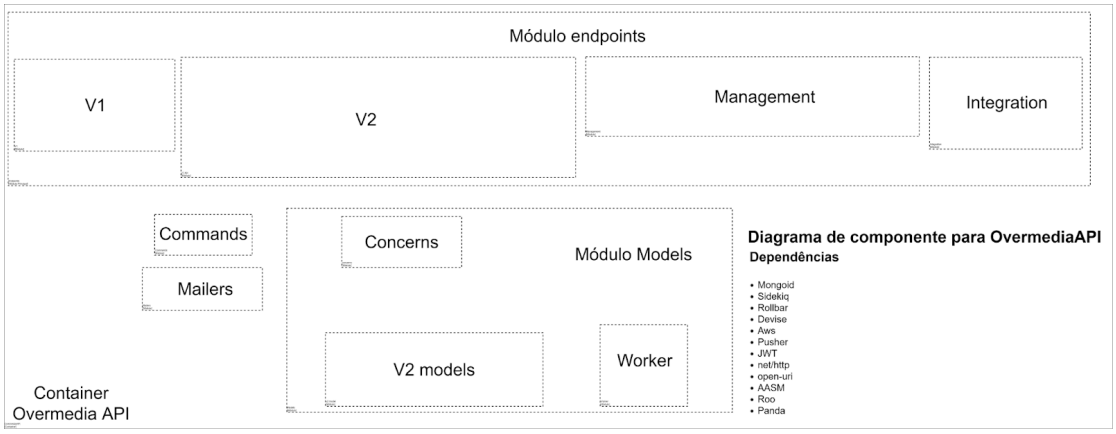


Figura 4.3: Diagrama de componentes do sistema Overmedia

Fonte: Elaborada pelo autor

Como é conhecido, uma API é analisada inicialmente pelos seus *endpoints*, que são as URLs onde o serviço pode ser acessado por uma aplicação cliente. Nessa API da Overmedia, existem cinco sub módulos dentro do módulo principal `"endpoint"`:

1. `"v1"`;
2. `"v2"`;
3. `"management"`;
4. `"integration"`;
5. `"session"`.

Um detalhe importante é que este último encontra-se totalmente comentado no arquivo que determina as rotas, significando que não está sendo usado, e logo, também não está sendo representado aqui.

O módulo `"v1"`, como mostra a Figura 4.4, pode ser considerado como o início da API, pois é aqui que contém o componente `"Home Controller"`, que faz a verificação inicial para saber se o usuário tem autorização para fazer as requisições das funcionalidades que aqui se encontram. Esse componente usa o `"Authorize API Request"` do módulo `"commands"`, que está sendo representado na Figura 4.5, para procurar o token decodificado do usuário que está tentando fazer as requisições, que deve ser encontrado na coleção de usuários autorizados do componente `"Auth`

Client Model".

Também se encontra no módulo "v1" o componente "Authentication Controller", que permite a autenticação de um usuário, de forma que usa o componente "Authenticate User" do módulo "commands" para codificar o id do usuário por um dia, que é a duração da autenticação, caso o usuário em questão seja encontrado na coleção de usuários autorizados do componente "Auth Client", que se encontra no módulo "models", como demonstrado na Figura 4.6.

Ainda no módulo "v1", existem outros três componentes que só são acessíveis caso o usuário seja autorizado previamente:

- Um componente para tratar envio de emails "Email Sendings Controller", que usa o componente "Mailer Base" do módulo "concerns", representado pela Figura 4.7, que gerencia os dados e as informações a serem passadas no email, e então envia o email usando o componente "Email Sending Mailer", do módulo "mailers", que está sendo apresentado pela Figura 4.8;
- Um componente para tratar envio de mensagens (torpedos) "SMS Sendings Controller", que usa o componente "SMS Base" do módulo "concerns", que gerencia os dados e as informações relacionadas ao torpedo, e que permite enviá-lo usando a API da AWS (do inglês, *Amazon Web Services*);
- Um componente para criar URLs curtas, que é a forma usada para referenciar as sessões de uma marca. Esse componente usa o "Url Generation Body Base", que gerencia os dados usados para criar a URL, sendo necessário usar alguns modelos que contém as informações referentes a um vídeo e de um canal de uma marca, "Video Model" e "Brand Channel Session Model", respectivamente.

O segundo módulo "v2", como mostrado na Figura 4.9, compõe, em sua maioria, de funcionalidades referente a um vídeo, como:

- "Moods Controller", que permite visualizar os sentimentos (estado de

espírito) de um vídeo;

- "Reasons Controller", que permite visualizar as reações (causas) de um vídeo;
- "Duplication Controller", que possibilita a duplicação de um vídeo;
- "Users to Exclude Controller" e "Sessions to Exclude Controller", que gerenciam os usuários e as sessões a serem excluídas, respectivamente;
- "People Grid Meta Controller", que permite visualizar o cabeçalho da tabela que exibe dos dados de navegação de usuário para um determinado vídeo;
- "Categories Controller", que gerencia as categorias de um vídeo;
- "Content Controller", que gerencia os conteúdos relacionados a uma categoria e que são acessados pelo "Category Model", *model* este que pertence ao módulo "v2 model", como mostrado na Figura 4.10;
- "Posts Controller", gerenciamento de cadastro de posts relacionados ao "Post Model";
- "Mailing Template Controller" que gerencia os modelos de emails usando as informações encontradas no "Mailing Template Model";
- "Conversion Configs Controller" que gerencia as configurações de conversão de vídeo, determinados em "Video Conversion Configs Model";
- "Unique Url Publishings Controller", que gerencia os canais de marca que tem uma URL única para seu acesso. Esse, por sua vez, obtém as informações através do componente "Brand Channel Session Unique Model".
- "Videobot Email Sending", que possibilita que um "video.bot" envie um

email para um determinado canal de marca, onde o gerenciamento dos dados e a liberação de envio são realizados a partir do componente "Videobot Mailer" do módulo "mailers";

- "Videobot Content Controller", que permite a visualização dos conteúdos de um "video.bot" adquiridos a partir do componente "Videobot Content Model";

- "Mailings Controller", que gerencia as toda a parte de envios relacionados a um vídeo, isso inclui características pertencentes a mailings, como arquivar mailings, pré-visualização, envios de emails e SMS, agendamento de emails e exportação de mailings para planilhas. Ele usa os componentes "Mailing Model" para obter as informações de um mailing e "URL Generation Model" para gerar uma URL para um mailing. Durante o processo de criação da URL, é usado o componente "Campaign Sending Model" para preparar os dados de envio da campanha, e então é usado o "Start Short Urls Generation" do módulo "workers" (Figura 4.11) para dar início a processo da geração das URLs curtas. Este componente, por sua vez, vai usar outros dois componentes do mesmo módulo:

    - "Create Brand Channel Session Worker" para criar uma nova sessão de uma determinada marca;

    - "Finish Short Urls Generation Worker" que conclui o processo de gerar URLs, utilizando o "Mailing Model" para alterar o status de um determinado mailing para "pronto".

Pertence ainda ao módulo "v2", mas não necessariamente é uma característica direta de um vídeo, o componente "Brand Channel Session Discovery", que define métodos para procurar por uma sessão de uma determinada marca, utilizando os modelos "Brand Channel Session Model" e "Brand Channel Session Unique Model", e caso encontre a sessão, os dados são recuperados através do "Videobot Tape Model". Todos os componentes deste módulo "v2" que estão

relacionados a um vídeo utilizam o componente "Base Controller", que é responsável por encontrar um vídeo, para que seja possível ter acesso a essas características. Esse controlador, por sua vez, usa o "Base Controller" do módulo "management", que libera o acesso às requisições através de um token específico de um usuário.

O módulo "management", como mostrado na Figura 4.12, lida com o gerenciamento de requisições feitas ao sistema. Um dos componentes principais deste módulo é o "Videos Controller", que gerencia as características e funcionalidades específicas de um vídeo, além de contemplar um conjunto de gerenciadores individuais, como gerenciamento de aprovação de vídeos, de manifestos, de criação de canal de marca, dos passos de engajamento, das *thumbnails,* dos envios e dos passos do fluxo de trabalho. Esse controlador de vídeos usa os modelos "Campaign Model" para obter as informações de uma campanha, "Campaign Sending Model" para obter os dados e as interações possíveis relacionados a uma campanha, e o "Video Config Model", para obter as informações das configurações de um vídeo.

Existe um componente neste módulo denominado "Campaigns Controller", que gerencia, de forma geral, as campanhas, incluindo os filtros e os eventos de funil, ou seja, o alcance, as interações e as reproduções que um determinado vídeo recebeu em certo período, de forma que podem ser classificadas por campanha, grupo ou por vídeo. Esse componente usa o "Account Model" para obter informações de uma conta de um cliente para quando for necessário adicionar uma nova campanha, e o "Campaign Model" para obter informações de uma campanha.

É possível obter dados dos dispositivos usados pelos clientes para assistir aos vídeos através do componente "Devices Controller", que usa o "Devices Model" para adquirir essas informações. O componente "Backlog Controller" fornece o gerenciamento de itens *backlog* do banco Itaú, que usa o componente "Backlog Item" para adquirir as informações relacionadas a um item do *product backlog.*

O componente "Users Controller" gerencia os usuários do sistema, incluindo funcionalidades como reiniciar a senha do usuário e o login de dois fatores.

Esse componente utiliza o "User Model" para obter informações de um determinado usuário, e este por sua vez, usa o componente "User Mailers" do módulo "mailers", para quando é necessário fazer envios de emails para um usuário.

O componente "Panda Controller" fornece funcionalidades de vídeo relacionadas ao serviço da *Pandastream* (*telestream cloud*), como fazer upload de vídeos na nuvem, e esse componente utiliza o "Video Model" para obter as informações de um determinado vídeo. Outro componente que funciona de forma semelhante é o "CDN Controller", onde há um gerenciamento de funcionalidades de CDN (do inglês, *Content Delivery Network*) da AWS relacionadas a um vídeo.

Esse módulo possui ainda outros dois componentes:

- "Mailer Controller", que permite envios de emails específicos de uma determinada marca/empresa, a partir de um "video.bot";
- "Brand Channel Session Controller", que gerencia as sessões de um "video.bot" de uma determinada marca, utilizando o "Brand Channel Session Model" para adquirir tais informações.

De forma semelhante ao módulo "v2", os componentes do módulo "management", com exceção do "Brand Channel Session Controller", utilizam o componente "Base Controller" para que seja possível fazer as requisições de gerenciamento.

O quarto módulo é o de integração, denominado de "integration", como mostrado na Figura 4.13, que lida com a automação de outros sistemas do cliente. Neste módulo, são encontrados quatro componentes principais. O primeiro é o "Campaigns Controller", que permite obter informações dos vídeos de campanhas ativas no momento. O segundo componente é o "Videos Controller" , que possibilita a obtenção de dados específicos de um vídeo, como os passos de engajamento e os usuários e sessões a serem excluídos. Esse controlador usa o "Video Model" para conseguir essas informações. Terceiro componente trata-se de "Devices Controller", que serve para obter conhecimento de todos os dispositivos que um

cliente usa para acessar os vídeos. O penúltimo componente é o "Brand Channel Sessions Controller", que permite visualizar as informações de uma sessão de alguma marca, onde vai ser procurado no sistema através de "Brand Channel Session Model", tanto pelo id da sessão como pelo id do cliente. Todos esses quatro componentes necessitam de uma autorização para serem requisitados, que é obtido através do "Base Controller", que faz a autenticação através do token de integração de uma conta do cliente.

O container "Overmedia API" como mostrado na Figura 4.3, possui também várias dependências, como:

- "MongoID": *Framework* para o Ruby que permite mapear os dados de algum módulo para a base de dados MongoDB. A maioria dos componentes do módulo "models" utiliza essa dependência;
- "Sidekiq": Processamento em segundo plano para ruby que permite utilizar *threads* para lidar com vários trabalhos ao mesmo tempo em um único processo. Muito utilizado pelo módulo "workers";
- "Rollbar": Serviço de relatório de exceção em tempo real para linguagem Ruby e outras linguagens. Usado comumente por controladores;
- "Devise": Solução flexível de autenticação para o RoR. Usado por componentes que necessitam de autenticação;
- "AWS": Fornece à linguagem Ruby fácil acesso aos serviços da Amazon, como a Amazon S3, Amazon EC2 e DynamoDB;
- "JWT": Permite implementar na linguagem Ruby o padrão RFC 7519 OAuth JSON Web Token (JWT)[3]. Usado pelos componentes "Authenticate User" e "Authorize API Request";
- "net/http": Fornece à linguagem Ruby uma biblioteca que pode ser usada para criar agentes de usuário HTTP. Projetada para trabalhar em conjunto com URI;

[3] link da documentação do padrão RFC 7519: https://datatracker.ietf.org/doc/html/rfc7519

- “open-uri”: Biblioteca que engloba e provê fácil uso do serviços HTTP, HTTPS e FTP;
- “AASM”: Fornece à linguagem Ruby uma biblioteca para adicionar estados de máquina finitos em suas classes. Está sendo usado pelo componente `“Mailing Model”`;
- “Roo”: Fornece à linguagem Ruby a possibilidade de leitura de vários tipos de planilhas;
- “Panda”: Fornece uma interface à linguagem Ruby para acessar a API do serviço *Pandastream*.

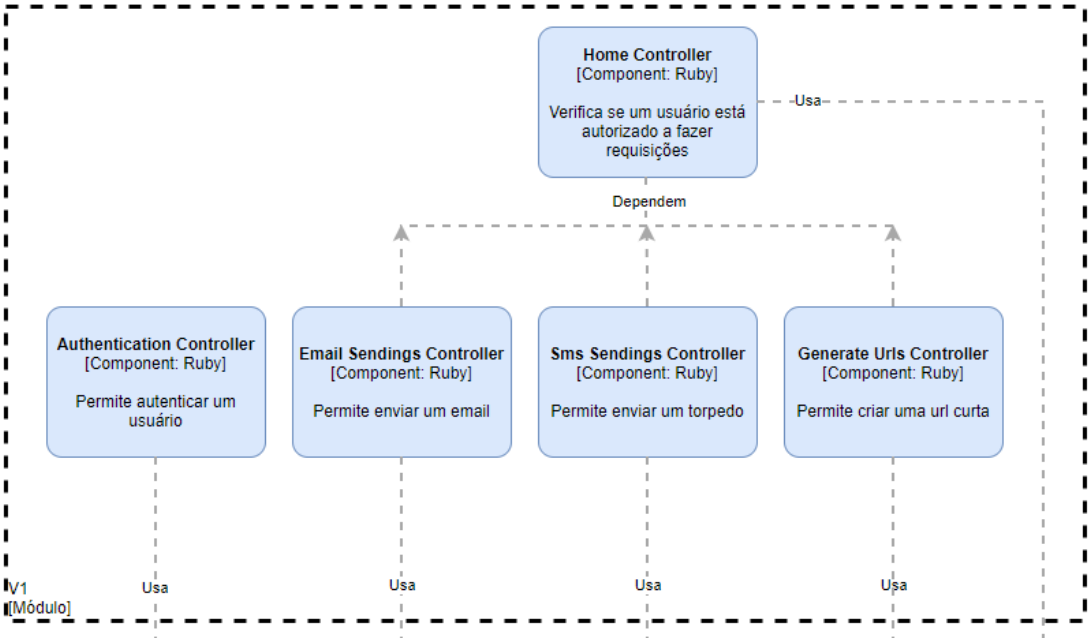


Figura 4.4: Diagrama do módulo v1

Fonte: Elaborada pelo autor

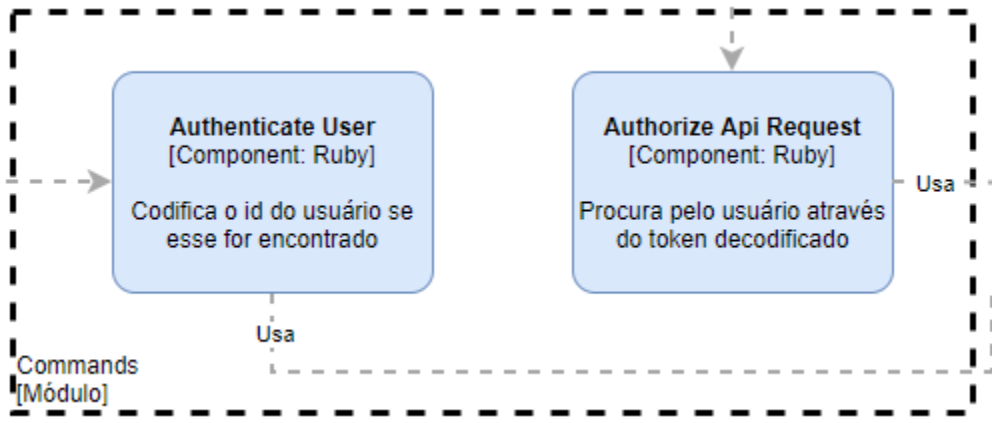


Figura 4.5: Diagrama do módulo commands

Fonte: Elaborada pelo autor

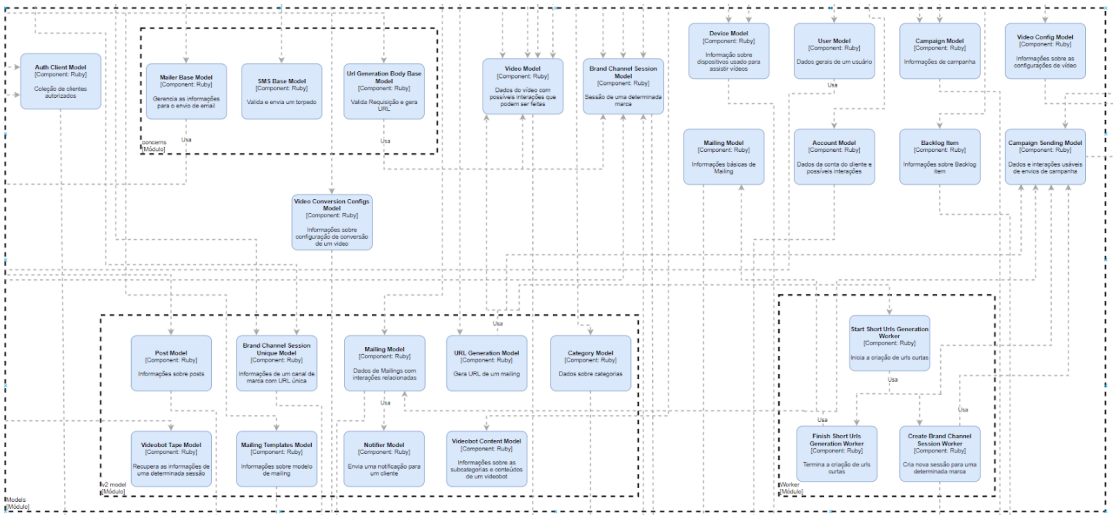


Figura 4.6: Diagrama geral do módulo models

Fonte: Elaborada pelo autor

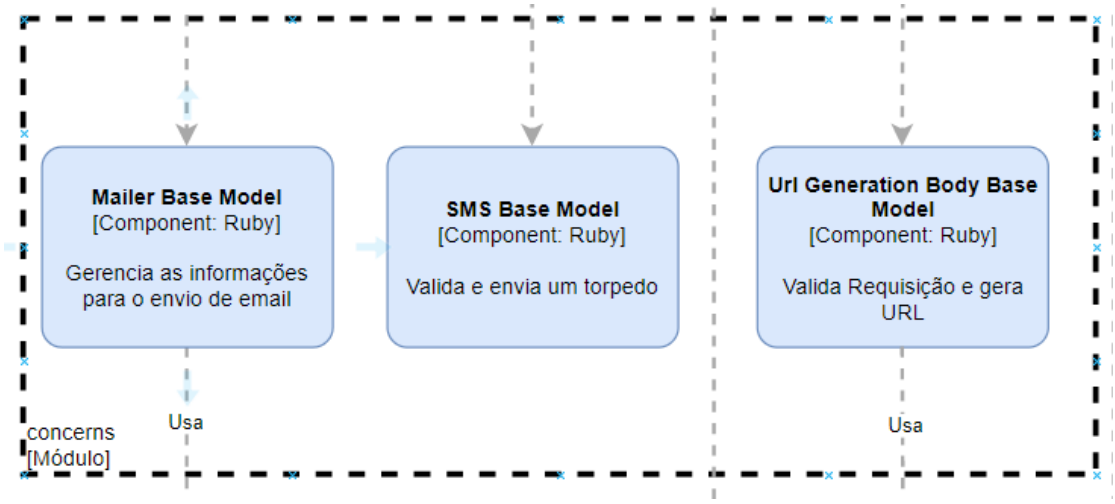


Figura 4.7: Diagrama do módulo concerns

Fonte: Elaborada pelo autor

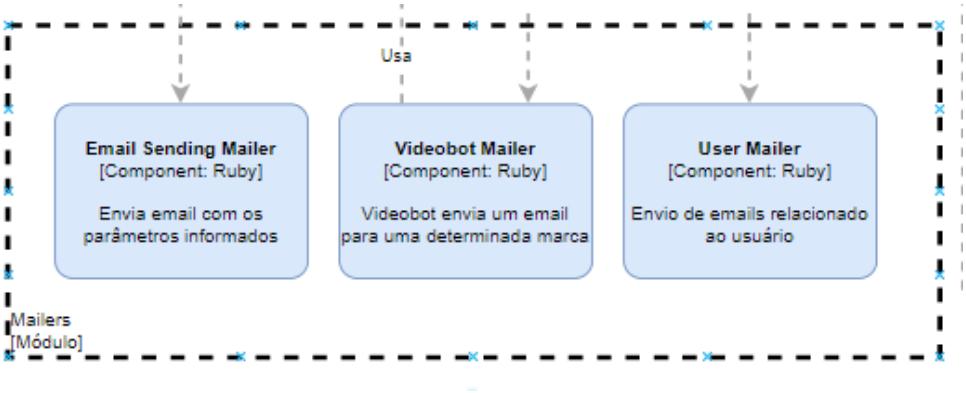


Figura 4.8: Diagrama do módulo mailers

Fonte: Elaborada pelo autor

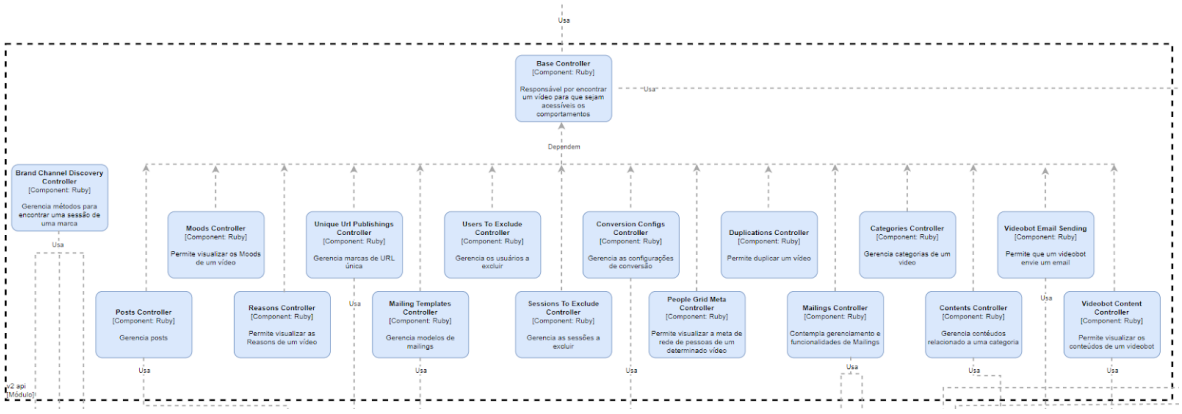


Figura 4.9: Diagrama do módulo v2

Fonte: Elaborada pelo autor

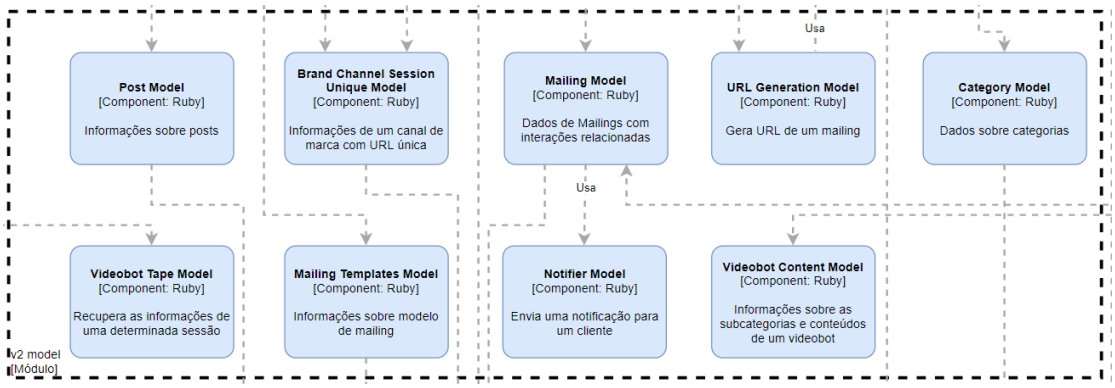


Figura 4.10: Diagrama do módulo v2 model

Fonte: Elaborada pelo autor

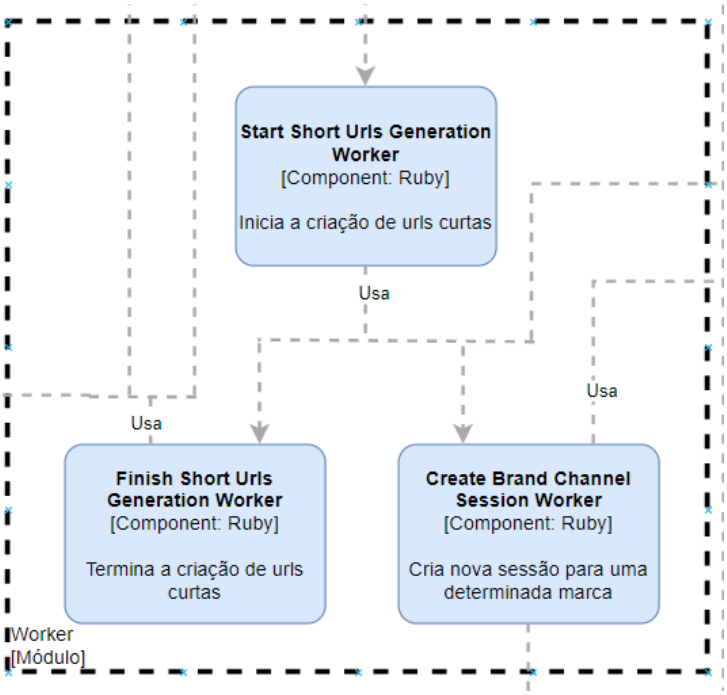


Figura 4.11: Diagrama do módulo worker

Fonte: Elaborada pelo autor

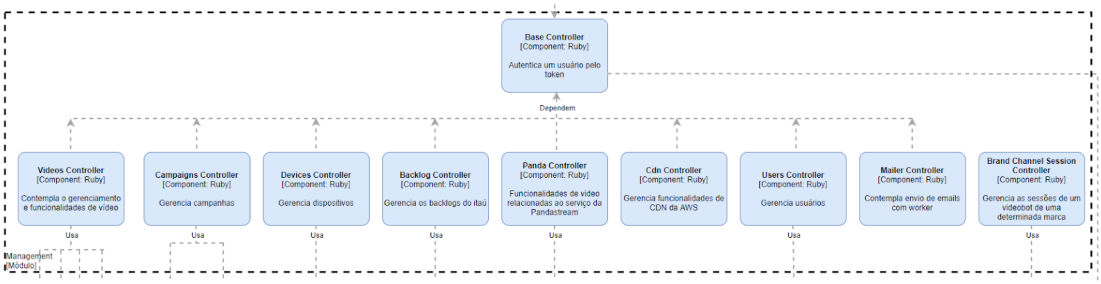


Figura 4.12: Diagrama do módulo management

Fonte: Elaborada pelo autor

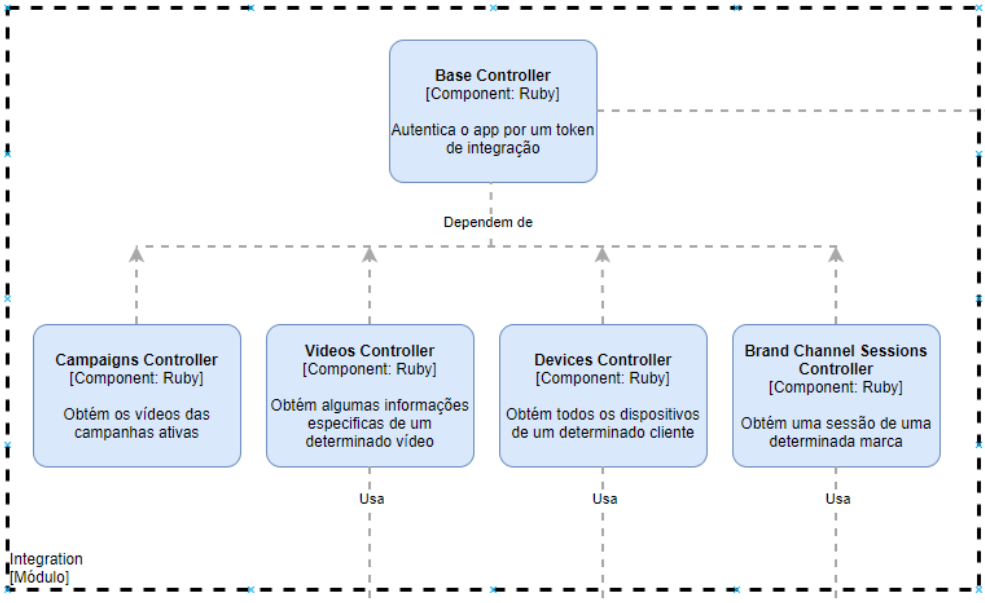


Figura 4.13: Diagrama do módulo integration

Fonte: Elaborada pelo autor

4.2 Resultados das avaliações

Esta seção apresenta o resultado da análise estática e dinâmica que foi aplicada na API Overmedia do sistema, que foi apresentado anteriormente. Estes resultados são importantes para obter informações e entender os problemas existentes atualmente no sistema.

4.2.1 Análise estática

Para obter dados sobre a qualidade do código fonte atual do sistema foi utilizado a ferramenta *Rubycritic*, que é uma *gem* que engloba *gems* de análise estática e fornece um relatório geral sobre o código analisado. Esta etapa busca identificar

violações sintáticas e falhas no design que causam um impacto negativo na qualidade da API. Esta qualidade é avaliada através de notas e numerações que definem a situação de um determinado componente. A Figura 4.14, que foi gerada através do uso da *Rubycritic*, apresenta uma visão geral da qualidade da API.

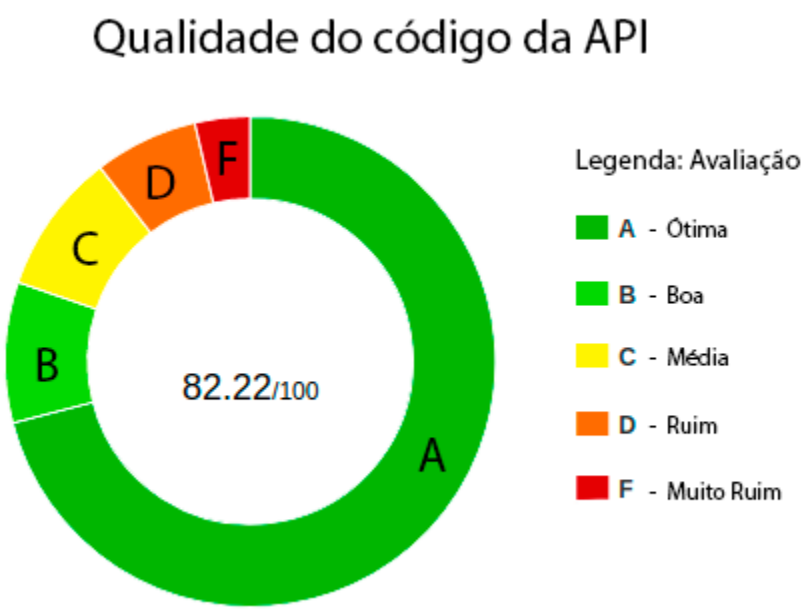


Figura 4.14: Visão geral da qualidade da Overmedia API

Fonte: Elaborada pelo autor

Dentro da `"Overmedia API"`, os arquivos com pior avaliação são os `"Recommendation Controller"` e o `"Videobot Mailer"`, como mostra a Figura 4.15. O primeiro componente pertence ao módulo `"session"`, que, como explicado anteriormente, não está sendo usado, se tornando um código legado, e que nesse caso, não deveria estar mais presente no código fonte, mas, no arquivo das rotas é informado que não foi removido pois o *James Browser* que está rodando em produção para o banco Itaú possui um método que redireciona para este módulo, ou seja, se remover agora iria quebrar o sistema. O segundo componente possui uma grande quantidade de código duplicado e um alto nível de complexidade relacionados a sua composição.

Em relação aos outros arquivos da lista, que possuem nome "`spec`", esses são arquivos originados pela *gem* "rspec-rails", que é um framework de testes onde gera

arquivos que contém explicações detalhadas de como a aplicação deve ser comportar, logo não serão analisados pois não é o alvo do estudo.

Um caso à parte é o componente `"Video Model"`, como mostrado na Figura 4.16, que apresenta a maior complexidade dos arquivos da API, com exceções dos arquivos specs, possuindo um valor de 317.28 pelo fato de possuir muitos métodos, e alguns com mais de 50 linhas. Também é possível observar pela Figura 4.17, que esse arquivo de vídeo possui um *churn* alto, o que significa que ele recebe alterações com frequência. Além disso, observando a lista de smells desses componentes, verifica-se a presença da *Feature Envy,* que indica um número maior de referências a objetos terceiros do que a si próprio.

Rating	Name	Churn	Complexity	Duplication	Smells
F	SessionApi::Public::RecomendationsController	66	140.64	161	23
F	VideobotMailer	41	308.61	362	46
F	LoginSpec	9	441.01	40	5
F	AbstractUserSpec	3	1097.73	404	13
F	CenarioUmItauSpec	37	314.5	37	6
F	DocsignsSpec	11	702.71	185	11
F	OcrControllerSpec	5	423.71	81	8
F	BugfixesSpec	1	190.51	3225	3
F	SmsSendingSpec	6	220.9	2150	7

Figura 4.15: Arquivos com pior avaliação da Overmedia API

Fonte: Elaborada pelo autor

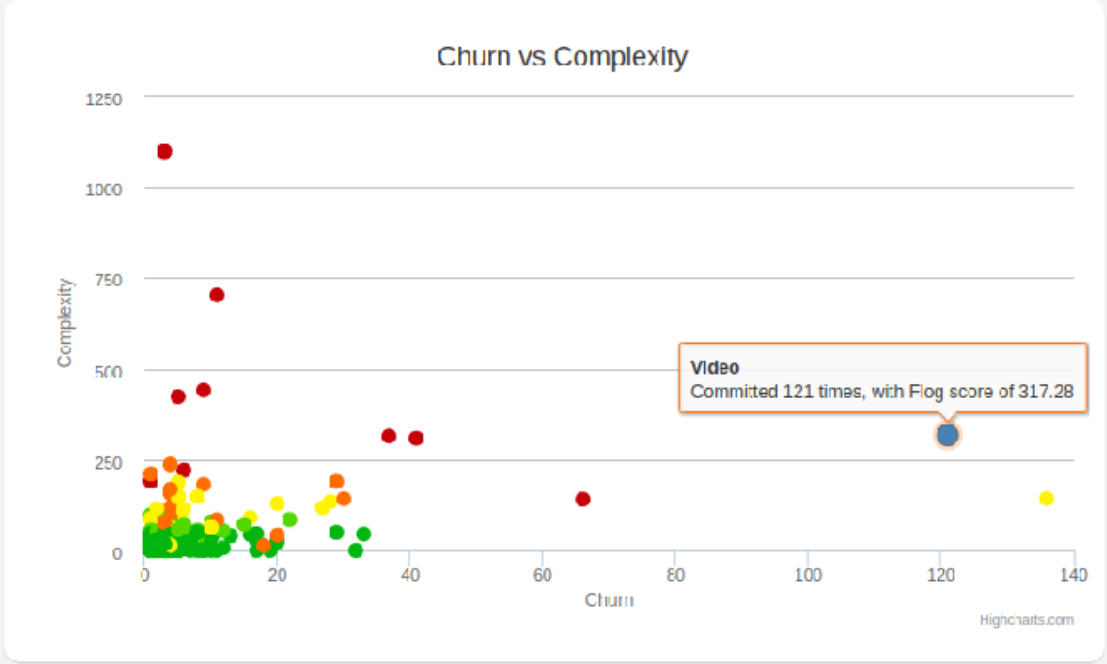


Figura 4.16: Gráfico churn vs complexidade da Overmedia API

Fonte: Elaborada pelo autor

Rating	Name	Churn ↓	Complexity	Duplication	Smells
C	Routes	136	142.53	0	1
D	Video	121	317.28	0	23

Figura 4.17: Arquivos com maior Churn

Fonte: Elaborada pelo autor

4.2.2 Análise dinâmica

Para a realização da análise dinâmica da API foram utilizadas as ferramentas:

- OWASP ZAP[4] (do inglês, *Open Web Application Security Project - Zed Attack Proxy*)

[4] OWASP ZAP: https://www.zaproxy.org/

- Postman API Platform[5].

O ZAP é uma ferramenta feita em Java que atua como um scanner de segurança de aplicativos da Web de código aberto, além de ser um dos projetos mais ativos da OWASP atualmente. Ela contém vários recursos embutidos, como interceptação de servidor proxy, rastreadores da Web (também conhecido como *spiders*) tradicionais e AJAX (do inglês, *Asynchronous Javascript and eXtensible Markup Language*), scan automatizado e passivo, forçar browser, *fuzzer*, entre outros. Já o Postman é um software gratuito do tipo API *Client*, que proporciona aos desenvolvedores a possibilidade de criar, compartilhar, testar e documentar uma API, através das solicitações HTTP e HTTPS feitas pelos usuários, podendo essas chamadas serem simples ou complexas.

O objetivo dessa etapa era simular ataques à API para identificar suas vulnerabilidades. Para alcançar esse objetivo, foi necessário fazer ataques em as rotas públicas da API, pois não é possível acessar todos os endpoints de um sistema externo.

A API avaliada possui uma documentação interna elaborada pela ferramenta Insomnia[6], como exibido na Figura 4.18, e é dividida nos seguintes módulos:

- `Authenticate`: Este módulo possui dois *endpoints*: um para obter o token de autenticação e outro para verificar se a autenticação foi feita com sucesso;
- `URL`: O endpoint deste módulo é usado para gerar URL dos "video.bot";
- `SMS`: O endpoint deste módulo é usado para enviar um SMS para um ou mais números de celular;
- `Email`: Este módulo possui dois *endpoints*, sendo um para enviar emails e o outro para envios de email relacionados a uma `"Brand Channel Session"`, podendo conter ou não os dados de um cliente;
- `Mailing Template`: Este módulo contém *endpoints* de um gerenciamento de

[5] Postman: https://www.postman.com/
[6] Insomnia: https://insomnia.rest/

cadastro relacionados ao "`Mailing Template`";

- `Users`: Este módulo possui *endpoints* para visualizar um usuário específico ou todos os usuários, além de poder atualizar os dados ou desativar um usuário;

- `PeopleGridMeta`: O endpoint deste módulo é usado para visualizar o cabeçalho da tabela que exibe dos dados de navegação de usuário;

- `Brand Channel Sessions`: Este módulo possui vários *endpoints* relacionados a busca de sessões de uma marca por diferentes métodos, além de poder enviar emails para e atualizar uma determinada sessão de uma marca;

- `Videobot Content`: Este módulo possui dois *endpoints*: um para obter os passos de engajamento e as sessões e usuários a serem excluídos de um determinado vídeo, e o outro para obter os conteúdos de um "video.bot";

- `Categories`: Este módulo contém *endpoints* de um gerenciamento de cadastro relacionados a categoria de um vídeo;

- `Category Content`: Este módulo possui *endpoints* para criar e visualizar os conteúdos relacionados a uma categoria.

Também foi necessário a autorização para ter acesso às informações dessas rotas, de forma que foi criado um ambiente de testes para evitar falsos alarmes.

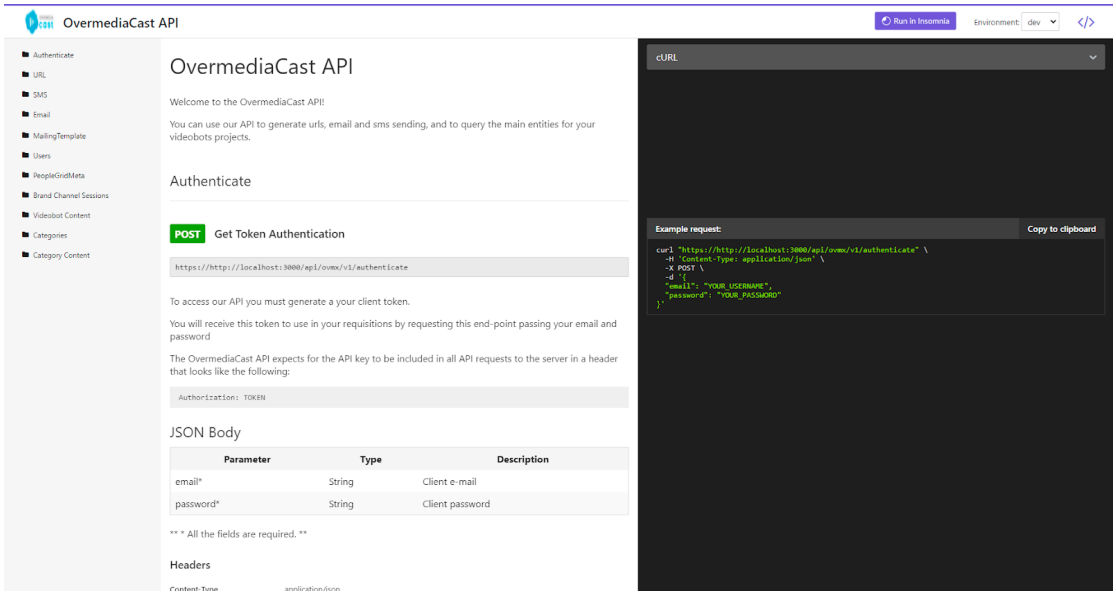


Figura 4.18: Documentação da API da Overmedia

Fonte: Elaborada pelo autor

Para alcançar o objetivo final dos testes de penetração, foram precisos primeiro mapear todas as rotas da API no Postman, manualmente, uma por uma, para poder analisar corretamente cada *endpoint*, onde se fez necessário ter os dados completos das chamadas, para testar os casos de sucesso e de erro. Depois foi preciso alterar as configurações do Postman para usar o ZAP como proxy (Figura 4.19): quando executado uma solicitação pelo Postman em uma das rotas, o ZAP consegue captar essa chamada e já faz a análise daquela rota (Figura 4.20), reportando possíveis falhas em forma de alertas.

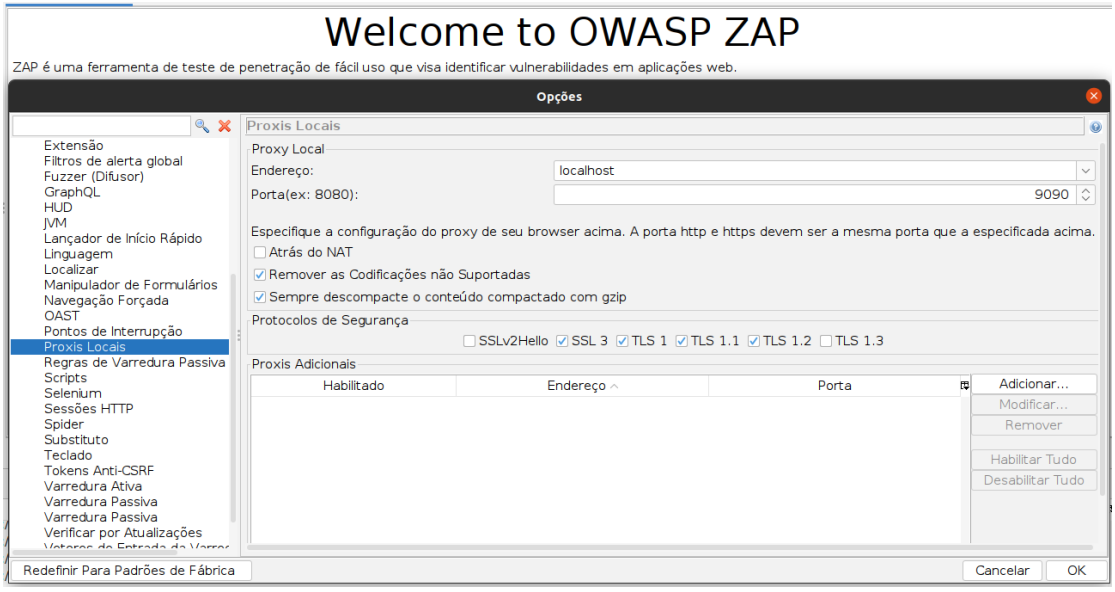


Figura 4.19a: Configurando o ZAP para ser utilizado como proxy

Fonte: Elaborada pelo autor

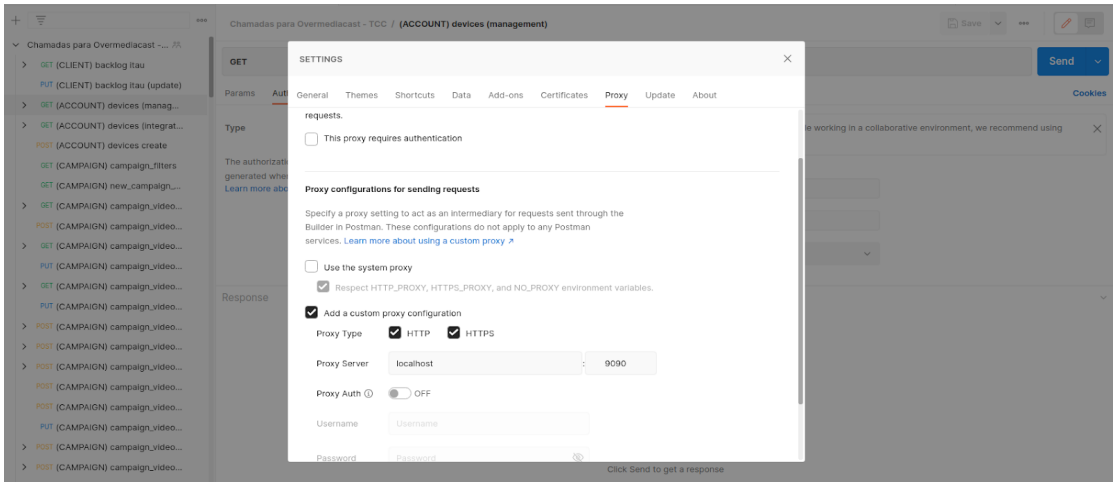
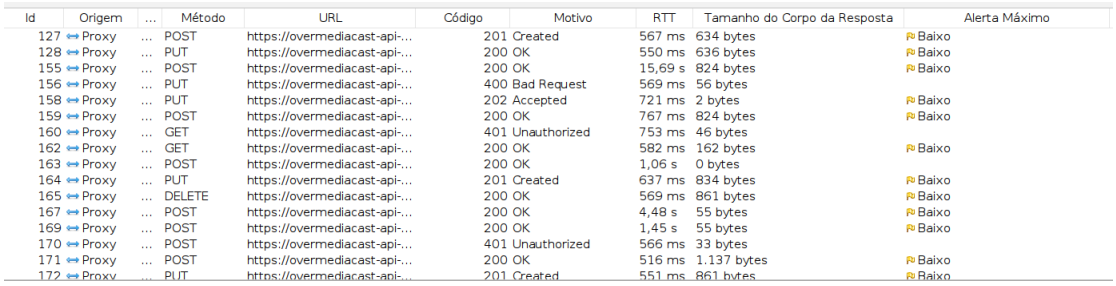


Figura 4.19b: Configurando o Postman para utilizar o ZAP como proxy

Fonte: Elaborada pelo autor



Id	Origem	...	Método	URL	Código	Motivo	RTT	Tamanho do Corpo da Resposta	Alerta Máximo
127	Proxy	...	POST	https://overmediacast-api-...	201	Created	567 ms	634 bytes	Baixo
128	Proxy	...	PUT	https://overmediacast-api-...	200	OK	550 ms	636 bytes	Baixo
155	Proxy	...	POST	https://overmediacast-api-...	200	OK	15,69 s	824 bytes	Baixo
156	Proxy	...	PUT	https://overmediacast-api-...	400	Bad Request	569 ms	56 bytes	
158	Proxy	...	PUT	https://overmediacast-api-...	202	Accepted	721 ms	2 bytes	Baixo
159	Proxy	...	POST	https://overmediacast-api-...	200	OK	767 ms	824 bytes	Baixo
160	Proxy	...	GET	https://overmediacast-api-...	401	Unauthorized	753 ms	46 bytes	
162	Proxy	...	GET	https://overmediacast-api-...	200	OK	582 ms	162 bytes	Baixo
163	Proxy	...	POST	https://overmediacast-api-...	200	OK	1,06 s	0 bytes	
164	Proxy	...	PUT	https://overmediacast-api-...	201	Created	637 ms	834 bytes	Baixo
165	Proxy	...	DELETE	https://overmediacast-api-...	200	OK	569 ms	861 bytes	Baixo
167	Proxy	...	POST	https://overmediacast-api-...	200	OK	4,48 s	55 bytes	Baixo
169	Proxy	...	POST	https://overmediacast-api-...	200	OK	1,45 s	55 bytes	Baixo
170	Proxy	...	POST	https://overmediacast-api-...	401	Unauthorized	566 ms	33 bytes	
171	Proxy	...	POST	https://overmediacast-api-...	200	OK	516 ms	1.137 bytes	Baixo
172	Proxy	...	PUT	https://overmediacast-api-...	201	Created	551 ms	861 bytes	Baixo

Figura 4.20: Exemplo de captura feita pelo ZAP

Fonte: Elaborada pelo autor

Durante os testes foi constatado que algumas das rotas presentes na API não foram executadas corretamente, devido a diferentes motivos: apesar de permanecerem no arquivo de rotas da API, a ação equivalente a determinadas rotas já não existe mais, ou foi alterada, logo é retornado um erro durante a sua chamada. Outras rotas necessitam de configurações adicionais para que a chamada seja completada, e que não estava devidamente configurado no ambiente de testes, portanto, não foi possível analisá-las.

Ao final dos testes, foi gerado um relatório pela própria ferramenta do ZAP, onde possui informações referentes aos alertas encontrados e possíveis tratamentos. Os alertas são divididos em quatro diferentes níveis de riscos: alto, médio, baixo e informativo, como mostrado na Figura 4.21. Seguindo esses nomes, é possível entender o quão inseguro está o código atualmente, onde os riscos de alto nível são mais problemáticos, necessitando de ajustes imediatos, enquanto os de baixo nível são menos preocupantes, mas que ainda precisam ser observados. Neste relatório, foi possível observar dois tipos diferentes de alertas, apresentados na Figura 4.22:

- Divulgação de Data e Hora - Unix: Este alerta informa que um carimbo de data/hora foi divulgado pelo aplicativo/servidor da web. A preocupação do desenvolvedor aqui é para que haja uma confirmação de que esses dados não sejam confidenciais, e que não sejam agregados, para evitar uma possível exploração de falha.

- *Incomplete or No Cache-control Header Set*: Este alerta informa que o

cache-control header não foi configurado corretamente ou está faltando, permitindo que o navegador e proxies consigam colocar algum conteúdo diferente para a cache, e assim alterando os resultados da chamada.

Como verificado, os alertas são de baixo nível de risco, o que não apresentam vulnerabilidades para a API, mas ainda é importante verificar e confirmar os dados apresentados nesse relatório, de forma que seja possível refinar o código para deixá-lo mais seguro.

Alert counts by risk and confidence

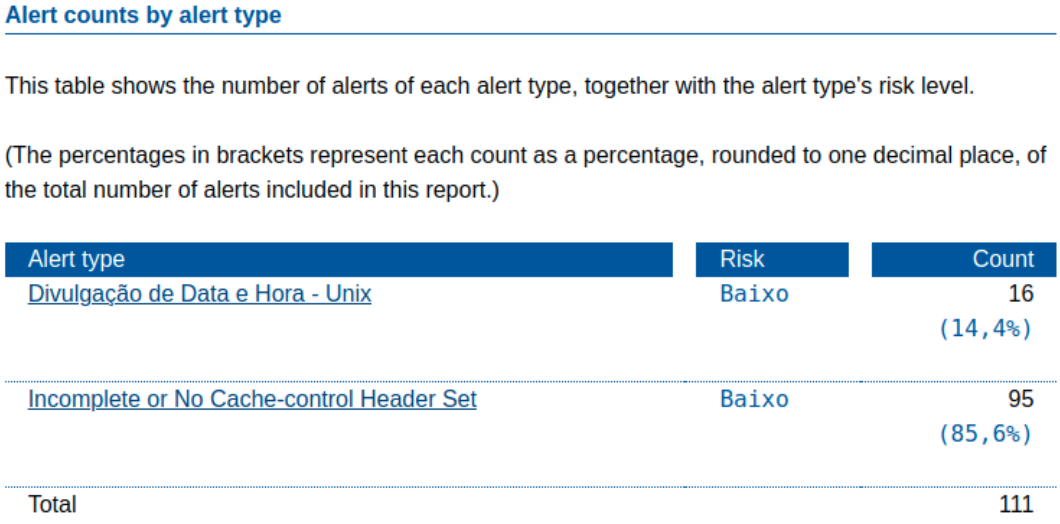
This table shows the number of alerts for each level of risk and confidence included in the report.

(The percentages in brackets represent the count as a percentage of the total number of alerts included in the report, rounded to one decimal place.)

		Confidence				
		User Confirmed	Alto	Médio	Baixo	Total
Risk	Alto	0 (0,0%)	0 (0,0%)	0 (0,0%)	0 (0,0%)	0 (0,0%)
	Médio	0 (0,0%)	0 (0,0%)	0 (0,0%)	0 (0,0%)	0 (0,0%)
	Baixo	0 (0,0%)	0 (0,0%)	95 (85,6%)	16 (14,4%)	111 (100,0%)
	Informativo	0 (0,0%)	0 (0,0%)	0 (0,0%)	0 (0,0%)	0 (0,0%)
	Total	0 (0,0%)	0 (0,0%)	95 (85,6%)	16 (14,4%)	111 (100%)

Figura 4.21: Resultado dos testes feitos na API

Fonte: Elaborada pelo autor



Alert counts by alert type

This table shows the number of alerts of each alert type, together with the alert type's risk level.

(The percentages in brackets represent each count as a percentage, rounded to one decimal place, of the total number of alerts included in this report.)

Alert type	Risk	Count
Divulgação de Data e Hora - Unix	Baixo	16 (14,4%)
Incomplete or No Cache-control Header Set	Baixo	95 (85,6%)
Total		111

Figura 4.22: Tipos de alerta encontrados nos testes da API

Fonte: Elaborada pelo autor

4.3 Mudanças sugeridas e implementadas

Observou-se através das métricas coletadas pela análise estática que o principal obstáculo em relação a manutenção dessa API é o alto grau de complexidade que existem em determinados componentes e o possível código legado.

O componente `"Videobot Mailer"` possui em sua composição a definição de envio de emails para uma determinada marca, onde cada marca possui um modelo próprio para esse envio, o que está causando a alta complexidade nesse caso. Uma possível solução seria separar esses métodos por arquivos, ou criar um único método de envio que consiga reconhecer a adaptar sua funcionalidade para a marca especificada.

Já o componente `"Video Model"` tem muita informação contida, uma vez que esse arquivo pode ser considerado um dos principais da API, que é onde se concentra as informações relacionadas a um vídeo, e quase todos os outros componentes acabam necessitando em algum momento deste componente. Existe uma variável denominada "fingerprints" que possui um dicionário com mais de 100 linhas de código com dados

alfanuméricos, que podem ser transferidas para outro arquivo. Também possui muitos métodos, onde alguns poderiam ser realocados em outros componentes e outros encurtados.

Um detalhe importante é na questão do código legado, que já devia ter sido removido da API. Apesar de informado que uma funcionalidade ainda direciona para o módulo `"session"`, é possível arrumar esta funcionalidade para que ela seja direcionada para o lugar correto, ou então voltar a usar esse módulo, em vez de deixar apenas como código morto.

Já as informações coletadas pela análise dinâmica demonstram que o código atual foi bem implementado, mas precisa de revisões, tanto para resolver os problemas apresentados como alerta no relatório do teste de penetração, como para retirar rotas não mais usadas, proporcionando uma API mais limpa.

5 CONCLUSÕES E TRABALHOS FUTUROS

A arquitetura de software é uma ferramenta de extrema importância para que um sistema alcance a expectativa dos clientes, ou seja, forneça uma solução adequada para qual foi projetado. A produção de uma boa arquitetura está relacionada a um processo de evolução contínua que está sempre se adaptando ao ambiente designado. Nesse âmbito, a avaliação da arquitetura é uma atividade indispensável, pois através dela é possível identificar problemas e coletar dados sobre o sistema de forma que se faça mais prático planejar como o software deve evoluir.

Este trabalho documentou uma API de um sistema de vídeos interativos, onde, a partir disso, gerou-se um documento acessível que contém informações relevantes sobre os métodos e suas funcionalidades que, até então, não eram tão evidentes para a equipe de desenvolvimento. O estudo de caso apresentado neste trabalho também averiguou que a arquitetura atual do sistema possui alguns pontos fracos que podem prejudicar o desempenho como um todo, e que pode ser melhorado. As métricas coletadas na avaliação estática demonstraram a presença de componentes que possuem grande quantidade de lógica em código, enquanto as coletadas na avaliação dinâmica, demonstraram informações importantes que não são facilmente perceptíveis, mas que, se não tratados, podem causar problemas como vazamento de dados sensíveis ou até falha de segurança.

Para os trabalhos futuros, é aconselhado a utilização de outras metodologias de avaliação, com o intuito de detalhar mais o sistema de modo que a análise da sua arquitetura atual seja feita de forma mais ampla e mais precisa, além de buscar novas propostas de evolução para a API estudada. Mediante os dados apresentados nas avaliações, seria interessante fazer um estudo dos arquivos "`.spec`" da API, pois, apesar de serem gerados por terceiros, alguns são modificados pela empresa, além de revisar manualmente o código implementado, de forma que seja verificado se as informações apresentadas nos alertas do relatório dos testes de penetração não sejam um problema.

Por último, mas não menos importante, é recomendado que, além da expansão da documentação da API, as futuras avaliações sejam trabalhadas em cima de um escopo maior, de forma que envolva também outros módulos do sistema que não fizeram parte do estudo de caso apresentado neste trabalho e, se possível, utilizar metodologias de avaliação mais específicas para esses outros módulos.

REFERÊNCIAS

ALTEXSOFT. **What is API: Definition, Types, Specifications, Documentation**. United States, 2019. Disponível em <https://www.altexsoft.com/blog/engineering/what-is-api-definition-types-specifications-documentation/>. Acesso em: 28 abr. 2021

BABAR, Muhammed et al. Agile Software Architecture: Aligning Agile Processes and Software Architectures. New York: Morgan Kaufmann, 2013.

BASS, Len et al. **Software Architecture In Practice.** Boston: Addison-Wesley, 2003.

Boehm, B.W. Software Risk Management: Principles and Practices. **IEEE Software**, 8, p. 32-41, jan. 1991

BROWN, Simon. O modelo C4 de documentação para Arquitetura de Software. InfoQ, 2018. Disponível em: <https://www.infoq.com/br/articles/C4-architecture-model/>. Acesso em: 4 jun. 2021.

FINNIGAN, Ken. **Enterprise Java Microservices**. USA. Manning Publications, 2018.

FOWLER, Martin. **Microsservices**, 2014. Disponível em: <https://martinfowler.com/articles/microservices.html>. Acesso em 19 mai. 2021.

FOWLER, Martin. **Software Architecture Guide**, 2019. Disponível em: <https://martinfowler.com/architecture/>. Acesso em 25 mai. 2021.

GARLAN, David; SHAW, Mary. **An introduction to software architecture**. Pennsylvania: Carnegie Mellon University, 1994.

GERMOGLIO, Guilherme. **Arquitetura de Software**. 2010. Disponível em: <http://cnx.org/contents/8e1f00c8-89da-4330-95dd-ead2890ea645@9.1>. Acesso em: 24 mar. 2021.

GUIMARÃES, Oliveira; NOGUEIRA, João. **Método para manutenção de sistema de software utilizando técnicas arquiteturais.** Dissertação - Escola politécnica, Universidade de São Paulo - USP, São Paulo, 2008.

IEEE. IEEE Recommended Practice for Architectural Description for Software-Intensive Systems. **IEEE Std 1471-2000**, p. 1–30, out. 2000

JÚNIOR, Manoel. **Como documentar API: passo a passo para otimizar integrações**, 2019. Disponível em: <https://www.take.net/blog/tecnologia/documentar-api/>. Acesso em 18 jul. 2021.

LIMA, Victor. **Um estudo de caso de um sistema de gestão de redes escolares.** Monografia - Centro de Informática, Universidade Federal da Paraíba - UFPB, João Pessoa, 2020.

MCCALL, J. et al. **Factors in software quality: concept and definitions of software quality**. nov. 1977

NUNES, Rodolffo. **Um estudo de caso sobre avaliação de arquiteturas de software para microsserviços.** Monografia - Centro de Informática, Universidade Federal da Paraíba - UFPB, João Pessoa, 2020.

NUNES, Thiago. **Projeto e Implementação de um Sistema de Software Distribuído Baseado em Micro-serviços.** Monografia - Centro de Informática, Universidade Federal da Paraíba - UFPB, João Pessoa, 2020.

PERRY, Dewayne E.; WOLF, Alexander L. Foundations for the Study of Software Architecture. **ACM SIGSOFT Software Engineering Notes**, v. 17, n. 14, p. 40–52, out. 1992.

ROESLER, Valter et al. **Special Topics in Multimedia, IoT and Web Technologies**. Springer Nature Switzerland AG, 2020.

FIELDING, Roy Thomas. **Architectural Styles and the Design of Network-based Software Architectures.** Dissertation, University of California, Irvine, 2000.

FUENTES, Vinícius. **Ruby on Rails: Coloque sua aplicação web nos trilhos**. São Paulo: Casa do Código, 2017.

TILKOV, Stefan. **Uma rápida Introdução ao REST**, 2008. Disponível em: <https://www.infoq.com/br/articles/rest-introduction/?itm_source=infoq_en&itm_medium=link_on_en_item&itm_campaign=item_in_other_langs>. Acesso em 9 mai. 2021.

SOMMERVILLE, Ian. **Software Engineering**. Boston: Pearson Education Limited, 2016.

SOUZA, Lucas. **Ruby: Aprenda a programar na linguagem mais divertida**. São Paulo: Casa do Código, 2020.