

**UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
CENTRO TECNOLÓGICO
DEPARTAMENTO DE ENGENHARIA ELÉTRICA
PROJETO DE GRADUAÇÃO**



JOSÉ HENRIQUE MERLO DE SOUSA

**DESENVOLVIMENTO DE APLICAÇÃO
MULTIPLATAFORMA PARA GERENCIAMENTO DAS
OPERAÇÕES DO PROJETO SOLIDARIEDADE DIGITAL**

VITÓRIA-ES

DEZEMBRO/2023

José Henrique Merlo de Sousa

DESENVOLVIMENTO DE APLICAÇÃO MULTIPLATAFORMA PARA GERENCIAMENTO DAS OPERAÇÕES DO PROJETO SOLIDARIEDADE DIGITAL

Parte manuscrita do Projeto de Graduação do aluno José Henrique Merlo de Sousa, apresentado ao Departamento de Engenharia Elétrica do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do grau de Engenheiro Eletricista.

Vitória-ES

Dezembro/2023

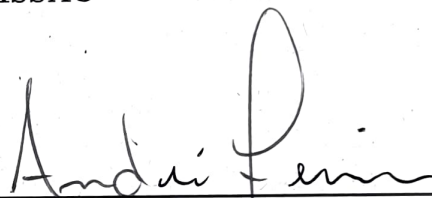
José Henrique Merlo de Sousa

DESENVOLVIMENTO DE APLICAÇÃO MULTIPLATAFORMA PARA GERENCIAMENTO DAS OPERAÇÕES DO PROJETO SOLIDARIEDADE DIGITAL

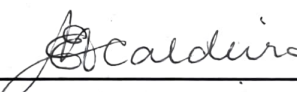
Parte manuscrita do Projeto de Graduação do aluno José Henrique Merlo de Sousa, apresentado ao Departamento de Engenharia Elétrica do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do grau de Engenheiro Eletricista.

Aprovado em 14 de dezembro de 2023.

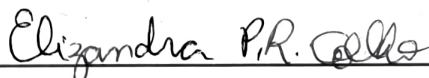
COMISSÃO EXAMINADORA:



Prof. Dr. André Ferreira
Universidade Federal do Espírito Santo
Orientador



**Prof. Dra. Eliete Maria de Oliveira
Caldeira**
Universidade Federal do Espírito Santo
Examinador



**Prof. Dra. Elizandra Pereira Roque
Coelho**
Universidade Federal do Espírito Santo
Examinador

Vitória-ES

Dezembro/2023

Aos meus pais pelo amor e apoio irrestrito.

AGRADECIMENTOS

Expresso meu profundo agradecimento a todas essas pessoas especiais que surgiram em minha vida e que permanecem ao meu lado, possibilitando que eu concluísse mais uma etapa.

Primeiramente, gostaria de agradecer aos meus pais, Danielle e Evelar. Suas palavras de encorajamento e confiança foram a luz que iluminou os dias mais desafiadores. Sem o suporte de vocês, esta jornada teria sido muito mais difícil.

À Laís, pelo incentivo, presença constante e compreensão nos momentos mais difíceis.

Aos amigos que caminharam comigo nesta trajetória, compartilhando risos, desafios e aprendizados.

Ao meu orientador, André, expresso minha sincera gratidão por toda assistência, orientação e dedicação ao longo do desenvolvimento deste trabalho.

À banca examinadora pelo aceite do convite e pelo tempo investido para leitura e avaliação desse trabalho.

Finalmente, agradeço à Universidade Federal do Espírito Santo pela formação pública, gratuita e de qualidade que me foi proporcionada.

RESUMO

A leitura, o registro e a atualização de dados são parte fundamental de muitos processos em que estão inseridos. Em um processo de manutenção de dispositivos eletrônicos, é boa prática realizar uma série de registros de componentes, peças e especificações, com devida consulta e atualização quando necessário. Muitas vezes, o volume de dados gerado é expressivo e pode se tornar confuso. Deste modo, a utilização de planilhas eletrônicas com propósito de um banco de dados pode trazer uma série de desvantagens, como a ausência ou mal aplicação da validação de dados, duplicidade de registros, informações mal dispostas e desempenho inferior em relacionamentos entre tabelas. Este trabalho descreve uma aplicação baseada no estilo de arquitetura REST, que possibilita o gerenciamento da manutenção dos dispositivos do projeto Solidariedade Digital. A proposta de aplicação busca uma interface multiplataforma que garanta melhor tratamento dos registros, conectada a uma API e um banco de dados, onde são definidas as lógicas e validações das transações. Para desdobramento do projeto, foram empregadas técnicas de modelagem e análise de requisitos, conceitos de arquitetura de *software* e aplicação de testes unitários com integração contínua. Em termos gerais, é obtida uma aplicação responsiva, eficiente, escalável e desacoplada, que implementa o fluxo de manutenção do projeto com foco na integridade das informações, aplicando validações e restrições nos lados de cliente e servidor.

Palavras-chave: Interface de Programação de Aplicações; Transferência Representacional de Estado; Linguagem de Modelagem Unificada.

ABSTRACT

The reading, recording, and updating of data are fundamental aspects of many processes in which they are embedded. In the maintenance process of electronic devices, it is a good practice to carry out a series of records of components, parts, and specifications, with proper consultation and updating when necessary. Often, the volume of generated data is significant and can become confusing. Thus, the use of spreadsheets with the purpose of a database can bring a series of disadvantages, such as the absence or improper application of data validation, duplicate records, poorly arranged information, and inferior performance in table relationships. This work describes an application based on the REST architecture style, enabling the management of maintenance for devices in the Solidariedade Digital project. The proposed application seeks a cross-platform interface that ensures better handling of records, connected to an API and a database, where the logic and validation of transactions are defined. For the project's development, modeling and requirements analysis techniques, software architecture concepts, and the application of unit tests with continuous integration were employed. In general terms, a responsive, efficient, scalable, and decoupled application is obtained, implementing the project's maintenance flow with a focus on information integrity, applying validations and constraints on both the client and server sides.

Keywords: Application Programming Interface; Representational State Transfer; Unified Modeling Language.

LISTA DE FIGURAS

Figura 1 – Modelo cliente-servidor.	19
Figura 2 – Representação de requisições cliente-servidor em um sistema CCSS. . .	20
Figura 3 – Componentes de uma mensagem HTTP.	24
Figura 4 – Arquitetura do sistema.	35
Figura 5 – Diagrama das fases de manutenção.	36
Figura 6 – Diagrama de casos de uso: computadores e componentes.	39
Figura 7 – Diagrama de casos de uso: empréstimos e tomadores de empréstimos. .	40
Figura 8 – Diagrama de casos de uso: usuários.	41
Figura 9 – Diagrama de classes: computadores e componentes.	42
Figura 10 – Diagrama de classes: empréstimos, comentários e históricos.	43
Figura 11 – Consulta ao banco de dados que recupera os nomes dos usuários. . . .	55
Figura 12 – Teste de autenticação utilizando credenciais inválidas.	56
Figura 13 – Teste de autenticação utilizando credenciais válidas.	56
Figura 14 – Teste de requisição para uma rota protegida sem a presença do cabeçalho de autorização.	57
Figura 15 – Teste de requisição para uma rota protegida com o cabeçalho de autori- zação.	57
Figura 16 – Teste com falha de validação da requisição de criação de um processador.	58
Figura 17 – Teste de sucesso de validação da requisição de criação de um processador.	58
Figura 18 – Tela principal de autenticação.	64
Figura 19 – <i>E-mail</i> de verificação recebido pelo usuário.	65
Figura 20 – Tela de visualização de indicadores.	66
Figura 21 – Gráfico de empréstimos e cartão de computadores sob responsabilidade.	66
Figura 22 – Tela de listagem de computadores.	67
Figura 23 – Tela do processo de manutenção na etapa de Rede e Periféricos.	68
Figura 24 – Tela de listagem de componentes.	69
Figura 25 – Tela de listagem de empréstimos.	70
Figura 26 – Tela de listagem de tomadores de empréstimos.	70
Figura 27 – Tela de listagem de usuários.	71
Figura 28 – Tela do painel inicial e listagem de computadores em dispositivo móvel.	72
Figura 29 – Proposta de integração do sistema com a biblioteca.	74

LISTA DE TABELAS

Tabela 1 – Principais métodos e respectivas funcionalidades do protocolo HTTP.	25
Tabela 2 – Natureza e significado das classes de respostas do protocolo HTTP. . .	26
Tabela 3 – Classes CSS da <i>framework</i> Quasar relacionadas à largura da tela que executa a aplicação.	33

LISTA DE ABREVIATURAS E SIGLAS

API	<i>Application Programming Interface</i>
ASCII	<i>American Standard Code for Information Interchange</i>
CCSS	<i>Client-Cache-Stateless-Server</i>
CLI	<i>Command-Line Interface</i>
CRUD	<i>Create, Read, Update and Delete</i>
CRLF	<i>Carriage Return Line Feed</i>
CSS	<i>Cascading Style Sheets</i>
CT	Centro Tecnológico
DVI	<i>Digital Video Interface</i>
EARTE	Ensino-Aprendizagem Remoto Temporário e Emergencial
FEST	Fundação Espírito Santense de Tecnologia
HDMI	<i>High-Definition Multimedia Interface</i>
HTML	<i>Hypertext Markup Language</i>
HTTP	<i>Hypertext Transfer Protocol</i>
JSON	<i>JavaScript Object Notation</i>
LAETI	Laboratório de Adequação de Equipamentos de Tecnologia da Informação
LDAP	<i>Lightweight Directory Access Protocol</i>
MVC	<i>Model-View-Controller</i>
ORM	<i>Object-Relational Mapping</i>
PROAECI	Pró-Reitoria de Assuntos Estudantis e Cidadania
PROEX	Pró-Reitoria de Extensão
PWA	<i>Progressive Web App</i>
QR	<i>Quick Response</i>

REST	<i>Representational State Transfer</i>
RFC	<i>Request for Comments</i>
RPC	<i>Remote Procedure Call</i>
SGBD	Sistema Gerenciador de Banco de Dados
SOAP	<i>Simple Object Access Protocol</i>
SQL	<i>Structured Query Language</i>
SUPEC	Superintendência de Comunicação
UFES	Universidade Federal do Espírito Santo
UML	<i>Unified Modeling Language</i>
URI	<i>Uniform Resource Identifier</i>
URL	<i>Uniform Resource Locator</i>
URN	<i>Uniform Resource Name</i>
VGA	<i>Video Graphics Array</i>
WSL	<i>Windows Subsystem for Linux</i>

SUMÁRIO

1	INTRODUÇÃO	13
1.1	Identificação do Problema e Solução Proposta	14
1.2	Estado da Arte	16
1.3	Objetivos	16
1.4	Estrutura do Texto	17
2	REFERENCIAL TEÓRICO	18
2.1	Estilo de Arquitetura REST	18
2.1.1	Restrições	18
2.1.1.1	Cliente-servidor	18
2.1.1.2	<i>Stateless</i>	19
2.1.1.3	<i>Cache</i>	20
2.1.1.4	Interface Uniforme	20
2.1.1.4.1	Recursos e Identificadores de Recursos	21
2.1.1.5	Sistema em Camadas	22
2.1.1.6	Código sob Demanda	22
2.2	Protocolo de Transferência de Hipertexto	23
2.2.1	Mensagens HTTP	23
2.2.1.1	Cabeçalhos	24
2.2.1.2	Requisições	25
2.2.1.3	Respostas	26
2.3	Lado do Servidor	26
2.3.1	Bancos de Dados	27
2.3.1.1	Modelo Relacional de Bancos de Dados	27
2.3.2	Linguagem de Modelagem Unificada	28
2.3.2.1	Diagrama de Casos de Uso	28
2.3.2.2	Diagrama de Classes	28
2.3.3	Interface de Programação de Aplicações	29
2.3.3.1	Laravel	29
2.3.3.2	Eloquent ORM	30
2.4	Lado do Cliente	30
2.4.1	Vue.js	30
2.4.1.1	Quasar	32
2.4.2	Aplicação Multiplataforma	32
2.4.2.1	Responsividade	32
2.4.2.2	<i>Progressive Web App</i>	33

2.4.2.3	<i>Mobile First</i>	33
3	METODOLOGIA E ETAPAS DE DESENVOLVIMENTO	35
3.1	Arquitetura e Análise de Requisitos	35
3.2	Lado do Servidor	43
3.2.1	Autenticação	44
3.2.2	Computadores e Componentes	48
3.2.3	Modelos Secundários	50
3.2.4	Rotas e <i>Middlewares</i>	51
3.2.5	<i>Cache</i>	53
3.2.6	Testes	53
3.2.6.1	Fábricas e Semeadores	53
3.2.6.2	Testes Unitários	55
3.3	Lado do Cliente	59
4	RESULTADOS	63
4.1	Recursos Computacionais	63
4.2	Módulos da Aplicação	63
4.2.1	Autenticação	64
4.2.2	Painel Inicial	65
4.2.3	Computadores	66
4.2.4	Componentes	68
4.2.5	Empréstimos e Tomadores de Empréstimo	69
4.2.6	Usuários	70
4.3	Responsividade	71
5	CONCLUSÃO E TRABALHOS FUTUROS	73
5.1	Conclusão	73
5.2	Trabalhos Futuros	74
	REFERÊNCIAS	76

1 INTRODUÇÃO

O projeto Solidariedade Digital¹ surgiu durante o período da pandemia de coronavírus, em que o distanciamento social e a adoção de *lockdowns* levaram os docentes e discentes da Universidade Federal do Espírito Santo a uma readaptação no que tange à realização de suas atividades de ensino, pesquisa e extensão.

Dentro deste contexto, a adoção do Ensino-Aprendizagem Remoto Temporário e Emergencial (EARTE) foi pautada em como manter a integridade do ensino e garantir seu acesso a todos. Para a realização de atividades de ensino de maneira remota, existe a necessidade de que as pessoas envolvidas no processo minimamente possuam dispositivos com conectividade à internet.

Como recurso, a Pró-reitoria de Extensão (PROEX) apresentou o projeto Solidariedade Digital, que conta com a participação da Superintendência de Comunicação (SUPEC) da UFES, da Pró-Reitoria de Assuntos Estudantis e Cidadania (PROAECI), do Centro Tecnológico (CT) e com o apoio da Fundação Espírito Santense de Tecnologia (FEST). O projeto é pautado em uma campanha de promover doações de equipamentos eletrônicos aos estudantes assistidos pela UFES, que surgiu como recurso paralelo a outras ações desenvolvidas pela universidade para acelerar o processo de inclusão digital, a fim de minimizar os prejuízos enfrentados por esses estudantes que não possuem as ferramentas básicas de acesso ao apoio pedagógico (UFES, 2021).

O Laboratório de Adequação de Equipamentos de Tecnologia da Informação (LAETI) desempenha um papel importante dentro do projeto Solidariedade Digital. Toda a manutenção dos equipamentos eletrônicos (*desktops*, *notebooks*, *tablets*, monitores, dentre outros) é realizada por discentes que receberam devida capacitação técnica para realizar intervenções nos dispositivos, tanto em *hardware* quanto em *software*. O processo de manutenção também é assistido e coordenado por docentes do Centro Tecnológico da UFES.

Apesar do lapso temporal entre o desenvolvimento deste *software* e o período marcado pela pandemia, torna-se evidente que a sua relevância perdura e se intensifica. O projeto Solidariedade Digital, concebido previamente, revela-se notavelmente útil nos dias de hoje, especialmente ao considerar a sua capacidade de contemplar estudantes que ainda carecem de acesso a computadores. A utilidade dos computadores destaca-se como um apoio indispensável em um cenário em que a virtualidade se tornou uma ferramenta essencial para a educação e comunicação. Este exemplo revela a resiliência e adaptabilidade do

¹ <https://proex.ufes.br/solidariedade-digital>

projeto, demonstrando como a inovação tecnológica pode transcender barreiras temporais e se manter relevante mesmo após desafios inesperados, como os apresentados pela pandemia.

A gestão dos ativos que estão sob manutenção é de suma importância para que o projeto obtenha resultados satisfatórios. O presente trabalho busca desenvolver uma ferramenta que facilite e aprimore a segurança, desempenho e organização dos dados que são gerados durante o processo.

1.1 Identificação do Problema e Solução Proposta

Até a implementação deste projeto, os dados dos dispositivos são dispostos em planilhas do *Google Sheets*², contendo informações como: especificações gerais de *hardware* e *software*, fabricante, número de patrimônio UFES (caso seja um dispositivo oriundo da própria universidade), informações de periféricos e testes. A pasta de trabalho contém várias planilhas, que são divididas em remessas de computadores e componentes avulsos. Por sua vez, cada planilha é dividida em 5 seções: (i) triagem; (ii) testes de *hardware*; (iii) manutenção; (iv) rede e periféricos; (v) testes de usuário.

Uma única planilha dentro da pasta de trabalho possui 114 linhas e 43 colunas. Com um número de colunas expressivo, a visualização e compreensão dos dados é prejudicada. A situação se torna ainda mais crítica quando a pessoa que realiza a manutenção acessa a planilha via aparelho móvel, que usualmente possui dimensões menores do que a de um *notebook* ou *desktop*. Uma otimização na disposição geral de como os dados são apresentados, nesse contexto, pode trazer um ganho significativo de produtividade.

A organização dos dados também parte de uma estrutura passível de aplicação de relacionamentos entre tabelas, visto que as planilhas de computadores e as planilhas de componentes avulsos deveriam se relacionar via chaves primárias. Neste caso, um banco de dados relacional é uma solução mais adequada (GANCHEV, 2022).

Além disso, as planilhas *online* possuem outras desvantagens:

1. A validação dos dados não abrange de forma ampla as necessidades específicas dos usuários e pode ser retirada acidentalmente por algum outro usuário conectado, o que reflete na integridade dos dados;

² <<https://workspace.google.com/google/sheets>>

2. Planilhas mostram tudo, o tempo todo. Isso implica na poluição visual que foi pontuada, além de maior tempo de processamento de dados que muitas vezes não são objetos da consulta;
3. Planilhas possuem uma pior experiência com mais de um usuário conectado ao mesmo tempo, além de consultas e filtros manuais menos otimizados (GANCHEV, 2022).

A partir disso, a proposta do projeto é de desenvolver uma aplicação multiplataforma que visa sanar os problemas apontados nesta seção. O desenvolvimento consiste em: uma interface de usuário (cliente) e uma API em conjunto com um banco de dados (servidor), baseados no estilo de arquitetura de *software* REST (FIELDING, 2000).

O projeto de uma aplicação multiplataforma visa otimizar o acesso aos dados em dispositivos móveis, com a criação de interfaces de usuário responsivas. Além disso, é proposta a implementação de um leitor de *QR Code* que possibilite maior praticidade em identificar um dispositivo, reduzindo as chances de erro humano em uma digitação, por exemplo.

A modelagem de um banco de dados de maneira apropriada também será útil para aprimorar a integridade, desempenho e segurança dos dados. Por meio da API, será possível realizar requisições para apenas os recursos de interesse. As consultas são mais personalizadas se comparadas com as planilhas e melhor dispostas na interface com o usuário.

Em termos de validação de dados, uma aplicação no modelo cliente-servidor possibilita a criação de duas camadas de defesa: uma em cada componente. No cliente, existe a validação dos dados antes de realizar a requisição, por meio de regras e testes de expressões regulares. No servidor, existe a validação dos dados antes de realizar qualquer transação com o banco de dados, em vista de que cada coluna de uma tabela em um banco de dados possui tipagem específica.

A divisão das planilhas em seções de triagem, testes de *hardware*, manutenção, rede e periféricos e testes de usuário refletem uma representação do processo executado pela equipe de manutenção. No espaço físico do projeto, os dispositivos são separados de acordo com a etapa em que se encontram. No desenvolvimento desta aplicação, também é proposta a criação da representação virtual deste cenário, identificando os dispositivos por etapa do processo de manutenção.

1.2 Estado da Arte

O gerenciamento eficiente do fluxo de manutenção do projeto Solidariedade Digital é essencial para assegurar a sua estabilidade, a confiabilidade e evolução contínua. No âmbito do desenvolvimento de *software*, a adoção da arquitetura REST emergiu como uma abordagem proeminente para a implementação de sistemas distribuídos, oferecendo princípios fundamentais como a escalabilidade, a simplicidade e a interoperabilidade. Este estado da arte busca explorar o cenário atual de *softwares* desenvolvidos a partir dos conceitos deste estilo de arquitetura, somado às características de experiência de usuário, modelagem e foco em integridade de dados.

Ao explorar os trabalhos relacionados, destaca-se a relevância de Tarriño e Albaladejo Gemma (2017) na abordagem multiplataforma de uma aplicação para criação de portais e gestão de dados. Tarriño e Albaladejo Gemma (2017) aplica os conceitos do padrão MVC como escolha de projeto, no qual há a separação da camada de negócio, do intermediador de requisições e da interface de usuário. Neste caso, apesar da separação entre camadas, as mesmas se encontram em uma mesma entidade, não caracterizando o cliente como independente. Vale notar que o presente trabalho se distingue ao incorporar os conceitos do estilo de arquitetura REST, adicionando uma camada de integração com ferramentas externas, ampliando o escopo e a utilidade do sistema proposto.

A contribuição de AL-atraqchi (2022) na utilização de APIs REST alinha-se com a tendência identificada neste estado da arte. O autor pontua uma série de aspectos relevantes sobre segurança na criação de uma API com a *framework* Laravel. O presente trabalho se compõe com parte dos conceitos apresentados, porém se diferencia ao introduzir o conceito de redundância na validação de dados e a implementação de uma aplicação multiplataforma. Em outros trabalhos analisados foi possível notar que apesar de denominar a arquitetura como REST, poucos autores citam a implementação de um *cache* de aplicação, restrição da arquitetura.

1.3 Objetivos

Objetivo Geral

- O desenvolvimento de uma aplicação multiplataforma para o gerenciamento dos processos de manutenção do projeto Solidariedade Digital, garantindo integridade

e desempenho. Para isso, pretende-se aplicar os conceitos do estilo de arquitetura REST, modelagem de *software* e experiência de usuário.

Objetivos Específicos

- Modelar o *software* de acordo com a sua análise de requisitos;
- Desenvolver uma API seguindo os conceitos do estilo de arquitetura REST;
- Desenvolver um cliente responsivo e multiplataforma;
- Criar um mecanismo de validação de dados íntegro para uso de múltiplos usuários;
- Introduzir testes unitários à aplicação.

1.4 Estrutura do Texto

O presente trabalho está estruturado conforme os seguintes capítulos:

- **Introdução:** capítulo inicial que estabelece um contexto que orienta o leitor sobre as motivações subjacentes à escolha do tema e aos métodos empregados;
- **Referencial Teórico:** capítulo em que são expostos os fundamentos teóricos que fundamentam a elaboração do projeto;
- **Metodologia e Etapas de Desenvolvimento:** são delineadas as fases de desenvolvimento do projeto, incluindo sua arquitetura, modelagem, demonstrações práticas de aplicação e testes unitários;
- **Resultados:** são apresentados os resultados obtidos através da implementação demonstrada nas etapas de desenvolvimento do projeto;
- **Conclusão e Trabalhos Futuros:** é realizada a conclusão do trabalho com base em seus objetivos, considerações finais e sugestão de trabalhos futuros.

2 REFERENCIAL TEÓRICO

2.1 Estilo de Arquitetura REST

O REST (*Representational State Transfer*) consiste em um conjunto de restrições dentro da arquitetura de *software* que permite a construção de sistemas distribuídos, seguros e escaláveis. O início do REST foi motivado por Roy Fielding, co-fundador da “*Apache HTTP Server Project*”, em sua tese de doutorado. Durante a definição do problema, Fielding (2000) condicionou a escalabilidade da *web* como sendo governada por uma série de restrições, propondo a abordagem pragmática de satisfazê-las uniformemente, garantindo que a *web* continuasse se expandindo.

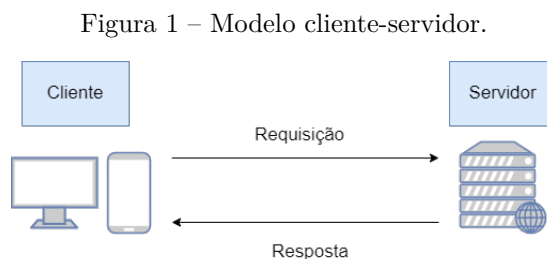
Em comparação com outros estilos, como SOAP (*Simple Object Access Protocol*) e RPC (*Remote Procedure Call*), o estilo de arquitetura REST se destaca devido à sua abordagem simplificada e orientada a recursos. Enquanto o SOAP e o RPC frequentemente envolvem operações complexas e contratos de serviço rigorosos, o REST adota uma abordagem mais leve e baseada em padrões da *web*, utilizando métodos HTTP e representações de recursos (MUNIZ et al., 2023).

2.1.1 Restrições

O conjunto de restrições abordado foi agrupado em seis categorias principais, que em conjunto formam o estilo de arquitetura de *software* REST.

2.1.1.1 Cliente-servidor

A primeira restrição adicionada ao REST é a implementação do modelo cliente-servidor. Neste contexto, um componente de servidor oferece um conjunto de serviços e escuta requisições por esses serviços. O componente do cliente é responsável por realizar esse tipo de requisição visando obter serviços, por meio de um conector. O servidor rejeita ou aceita a requisição, retornando uma resposta ao cliente. A interação cliente-servidor pode ser representada de acordo com a Figura 1.



Fonte: Produção do próprio autor.

Segundo Fielding (2000), a principal vantagem desse modelo de sistema é a permissão do desacoplamento do servidor com o cliente, o que o autor chamou de “separação de preocupações”. Quando separada a atribuição de interface de usuário e o armazenamento de dados, o projeto otimiza sua portabilidade em várias plataformas e permite com que seja mais facilmente escalável, já que os componentes do servidor se tornam mais simples. O mais importante é que os componentes possam o livre arbítrio de evoluir independentemente, exigindo múltiplos domínios organizacionais para atender a escalabilidade da *web*.

2.1.1.2 *Stateless*

A restrição *stateless* define que, por natureza, a comunicação estabelecida entre cliente e servidor não possua um estado definido, de forma que cada requisição enviada pelo cliente ao servidor contenha todas as informações necessárias para que o servidor a compreenda, não podendo o cliente gozar de contextos armazenados no servidor. Dessa forma, o estado de sessão é em sua totalidade responsabilidade do cliente (FIELDING, 2000).

Em termos de vantagens, o servidor *stateless* induz maior visibilidade, confiabilidade e escalabilidade ao sistema.

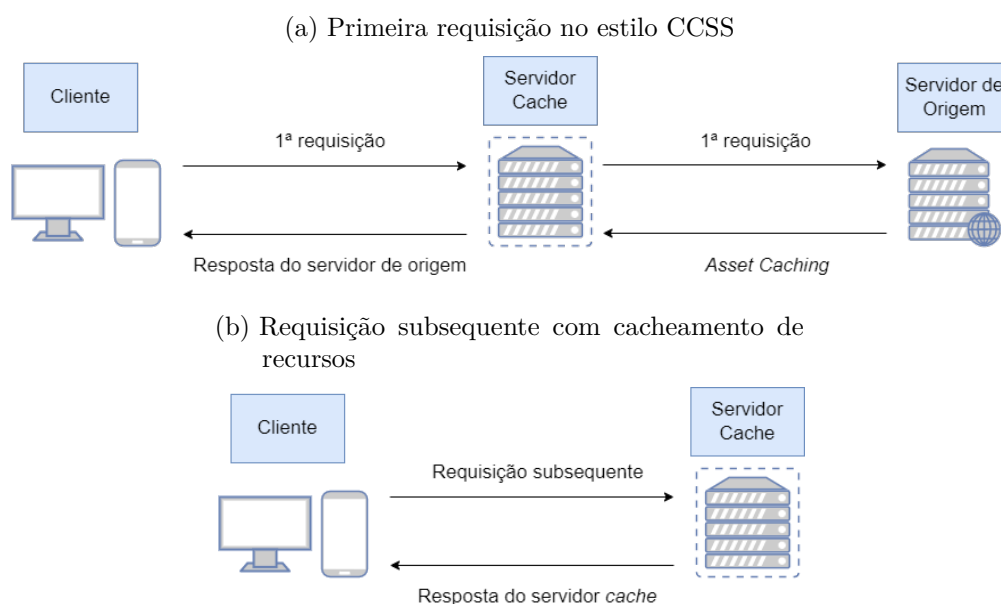
1. A visibilidade é otimizada frente à necessidade de consultar apenas uma requisição para compreender toda a sua natureza, independente de outra sessão;
2. A confiabilidade do sistema é aperfeiçoada, pois há maior facilidade em se recuperar de falhas parciais. Nesse contexto, a melhor visibilidade contribui com a confiabilidade do sistema diretamente;
3. A escalabilidade é potencializada pelo fato de não ser delegada ao servidor a função de gerenciar o uso de recursos entre requisições, simplificando sua implementação.

2.1.1.3 Cache

Em termos de otimização de rede, a restrição de *cache* é implementada unida à restrição *stateless* abordada na Seção 2.1.1.2, configurando o estilo de cliente-servidor sem estado de *cache*, denominado *Client-Cache-Stateless-Server* (CCSS).

Na Figura 2, um *cache* atua como mediador entre cliente e servidor, em que as respostas anteriores às solicitações do cliente podem ser reutilizadas, se dispostas em *cache*, em futuras respostas às solicitações, resultando em muitos dos casos em respostas idênticas.

Figura 2 – Representação de requisições cliente-servidor em um sistema CCSS.



Fonte: Produção do próprio autor.

A principal vantagem da implementação da restrição de *cache* é a de eliminar de maneira total ou parcial a necessidade de interações pontuais com o servidor, otimizando o desempenho concebido pelo usuário da aplicação (FIELDING, 2000).

2.1.1.4 Interface Uniforme

A característica mais enfática que difere o estilo de arquitetura REST de outros estilos baseados em rede é a uniformização de interfaces entre seus componentes. Esse padrão possibilita a maior independência entre os serviços que cada componente presta, o que potencializa a sua escalabilidade (FIELDING, 2000).

A aplicação de uma interface uniforme é a chave para que o modelo cliente-servidor possa ser implementado com eficácia. Por meio dela, cliente-servidor se comunicam de maneira padronizada, independente da linguagem em que cada um desses foi desenvolvido. Como exemplo, um cliente desenvolvido na linguagem de programação *Javascript* deve se comunicar, por meio de uma interface uniforme, com um servidor *back-end* desenvolvido na linguagem de programação PHP.

A interface uniforme funciona como um meio padronizado e imutável de comunicação entre os componentes do sistema. No caso da presente proposta de projeto de graduação, será utilizado o protocolo HTTP com recursos identificados por URI's, por meio de operações CRUD (*Create, Read, Update, Delete*) e a aplicação do formato JSON no modelo de requisição-resposta do sistema, sintaxe padronizada de intercâmbio de dados entre componentes desenvolvidos em qualquer linguagem de programação.

2.1.1.4.1 Recursos e Identificadores de Recursos

No estilo de arquitetura REST, a abstração chave de informações é contida em um recurso. Um recurso consiste em qualquer tipo de informação que pode ser representado na *web*, como um mapeamento conceitual para um conjunto de entidades. Segundo Fielding (2000), qualquer informação que possa ser nomeada é considerada um recurso: um documento, imagem, objeto, etc. No caso do presente trabalho, a representação de um computador a ser realizada intervenção é considerada um recurso.

Para que seja realizada a manipulação e leitura das informações que estão englobadas pelos recursos e suas interações, o REST utiliza identificadores uniformes de recursos que podem ser classificados como localizadores, nomes, ou ambos.

1. URI - *Uniform Resource Identifier* - é um identificador que consiste em uma sequência de caracteres que combinam as regras de sintaxe nomeadas para cada recurso (BERNERS-LEE; FIELDING; MASINTER, 2005);
2. URL - *Uniform Resource Locator* - se refere a um subconjunto de URI's que adicionalmente à identificação do recurso engloba a informação de seu endereço, fornecendo um mecanismo de acesso primário (BERNERS-LEE; FIELDING; MASINTER, 2005);
3. URN - *Uniform Resource Name* - possibilita a identificação única de um recurso, de forma persistente e que independe de sua localização (BERNERS-LEE; FIELDING; MASINTER, 2005).

2.1.1.5 Sistema em Camadas

A quinta restrição determinada por Fielding (2000) em sua tese consiste no desenvolvimento do servidor a partir de composições hierárquicas que subdividem o componente em camadas. Sobretudo, essa restrição implica em que o cliente obtenha acesso apenas à camada mais externa do servidor, podendo esta utilizar de camadas inferiores para compor os recursos em suas respostas.

A implementação de um sistema em camadas tem função importante na escalabilidade do sistema, visto que tecnicamente podem ser alocadas diversas funcionalidades como balanceamento de carga e represamento.

A principal desvantagem da representação do sistema em camadas consiste no aumento da latência no intercâmbio de dados, prejuízo que segundo Fielding (2000) pode ser contornado por meio do compartilhamento de *cache* entre camadas intermediárias. A combinação de um sistema em camadas com uma interface uniforme permite com que até mesmo os componentes intermediários transmitam o conteúdo de uma mensagem, pois estas são auto-descritivas e possuem uma semântica padronizada.

2.1.1.6 Código sob Demanda

Por fim, a última proposta de restrição trata-se do *code-on-demand*, que define a possibilidade de que os serviços baseados em REST retornem respostas com a representação de um recurso por meio de um código executável via cliente.

Segundo Fielding (2000), por meio do código sob demanda é possível simplificar o cliente, uma vez que diminui a quantidade de ferramentas necessárias a serem preestabelecidas no seu desenvolvimento. Fielding (2000) ainda categoriza esta restrição como opcional. Neste contexto, o autor pontua que uma “restrição opcional” soa como um oxímoro¹, porém defende que uma restrição opcional garante que haja o comportamento desejado no caso geral, mas com o entendimento de que possa ser desabilitada em contextos específicos.

¹ Oxímoro é uma figura de linguagem que consiste em relacionar em uma mesma expressão palavras que exprimem sentidos contrários.

2.2 Protocolo de Transferência de Hipertexto

O presente projeto propõe a implementação do HTTP como seu protocolo padrão, o que converge com as restrições do estilo de arquitetura REST discutidas na Seção 2.1. Nesta seção serão abordados os principais conceitos do HTTP com relação à sua estrutura e aplicações.

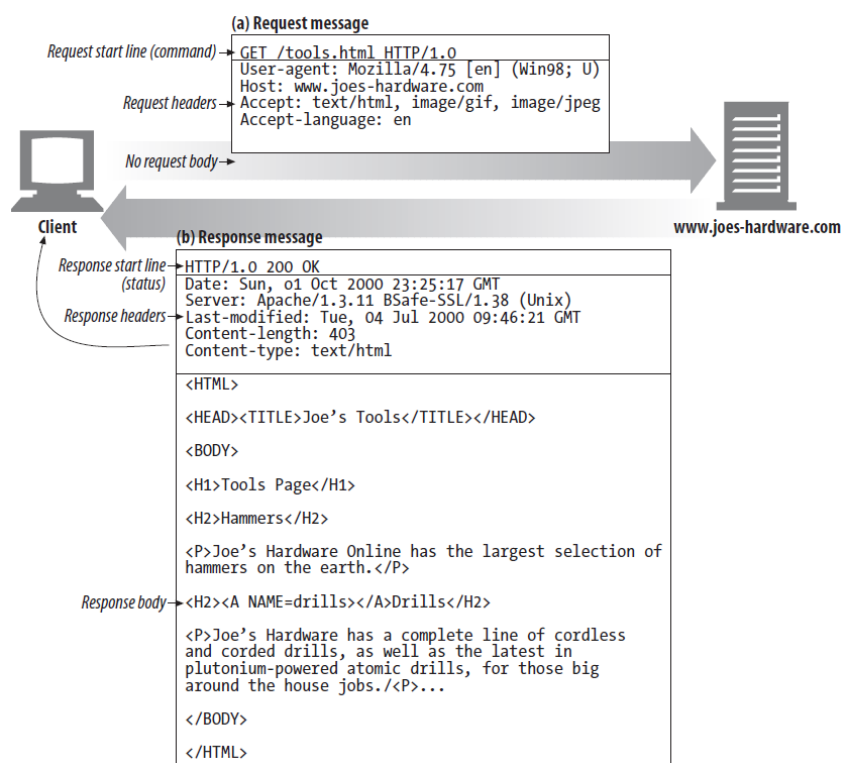
O HTTP é o protocolo de transferência de hipertexto mais utilizado na *web* para intercâmbio de recursos, segundo Gourley et al. (2002). Este protocolo é baseado na comunicação do modelo cliente-servidor através de mensagens individuais que independem entre si. As mensagens são denominadas “requisições” quando são enviadas pelo cliente (geralmente um navegador da *web*) e “respostas” quando enviadas pelo servidor *web*.

2.2.1 Mensagens HTTP

As mensagens utilizadas no protocolo HTTP seguem a estrutura genérica definida por Crocker (1982) na RFC 822² para entidades de transferência, tanto para requisições quanto para respostas. Estas mensagens consistem em campos de cabeçalho e, opcionalmente, um corpo que contém sequências de linhas com caracteres ASCII. A Figura 3 representa a composição de duas mensagens HTTP, de uma requisição e de uma resposta. A linha inicial pode apresentar a informação do método para a requisição ou do status para uma resposta. Abaixo da linha inicial são identificados os *headers* de cada mensagem, seguidos de uma linha em branco que indica o fim desta seção. Por fim, o corpo da mensagem é identificado para a mensagem de resposta, em formato HTML. É importante ressaltar a ausência do corpo da mensagem no exemplo de requisição, destacando a não obrigatoriedade da presença deste componente.

² O RFC é uma série de publicações que documenta padrões e protocolos oficiais da internet que são mantidos pelo IETF (*Internet Engineering Task Force*)

Figura 3 – Componentes de uma mensagem HTTP.



Fonte: Gourley et al. (2002).

2.2.1.1 Cabeçalhos

Os *headers* ou cabeçalhos das mensagens HTTP são importantes semanticamente para as requisições e respostas, principalmente para a troca de informações adicionais entre cliente e servidor. São os cabeçalhos que definem as regras de negócio de uma solicitação ou requisição, trocando informações de autenticação, sessão, tipo de recurso, controle de origem de acesso, dentre outros.

Nielsen et al. (1999) define que cada campo de cabeçalho consiste no nome do campo, seguido de um operador “:” e o seu respectivo valor. Além disso, os cabeçalhos são classificados em categorias de acordo com suas aplicabilidades:

1. *General-headers*: cabeçalhos gerais que possuem aplicações tanto para as requisições quanto para as respostas. Exemplo: cabeçalho *Cache-Control*, que define as diretivas que devem ser obedecidas por todos os mecanismos de *cache* na cadeia de requisição-resposta.

2. *Request-headers*: cabeçalhos que permitem ao cliente transmitir dados adicionais sobre a requisição, bem como informações sobre si mesmo. Um exemplo que será abordado é o do *header Authorization*, que é utilizado com fins de autenticação do cliente, o qual transmite suas credenciais para acessar determinado recurso.
3. *Response-headers*: cabeçalhos que permitem ao servidor transmitir dados adicionais sobre a resposta que não podem ser inseridos no código de *status*. Um exemplo de *response-header* é o cabeçalho *Retry-After*, utilizado em conjunto com o código de *status* 503 (serviço indisponível), que indica o tempo em que o servidor estará indisponível até uma próxima requisição do cliente.

2.2.1.2 Requisições

O formato da mensagem de requisição é composto pelas componentes *request line*, *request headers* e *request body*. Os cabeçalhos da mensagem podem conter tanto cabeçalhos do tipo *general-headers* quanto *request-headers*. O corpo da mensagem, assim como descrito anteriormente, é opcional e é composto por caracteres no formato ASCII.

A linha de requisição é composta por uma componente que indica o método da requisição, o URI do recurso requisitado e a versão do protocolo HTTP utilizado, separados por caracteres de espaço e com um caractere “CRLF” indicando o fim da linha. O método da requisição é *case-sensitive* e indica o método a ser executado em determinado recurso, identificado pelo URI subsequente. Na Tabela 1, são descritas as funcionalidades dos principais métodos HTTP utilizados.

Tabela 1 – Principais métodos e respectivas funcionalidades do protocolo HTTP.

Método	Funcionalidade
GET	Busca qualquer informação que possa ser identificada por uma URI.
HEAD	Retorna metadados de uma informação identificada por uma URI assim como o método GET, porém este não devendo possuir um corpo de mensagem.
POST	Cria um novo recurso.
PUT	Atualiza ou cria um recurso especificado.
DELETE	Remove um registro específico.
OPTIONS	Requisita informações sobre opções disponíveis na cadeia resposta-requisição de um recurso específico.

Fonte: Adaptado de Nielsen et al. (1999).

2.2.1.3 Respostas

O formato da mensagem de resposta se difere da mensagem de requisição apenas com relação à sua linha inicial, denominada *response line*. Os cabeçalhos da mensagem podem conter tanto cabeçalhos do tipo *general-headers* quanto *response-headers*.

A linha de resposta é composta por uma componente que indica a versão HTTP, seguida do código de *status* e um breve texto descritivo do código de *status*, separados por caracteres de espaço e com um caractere “CRLF” indicando o fim da linha.

O código de *status* é um número inteiro de três dígitos utilizado como realimentação geral de uma requisição. O primeiro dígito identifica a classe de resposta, enquanto os outros dois dígitos não possuem uma categoria definida. Na Tabela 2, são descritas as naturezas e seus significados de acordo com a classe de resposta identificada pelo primeiro dígito da coluna “Código de *Status*”. O sufixo “xx” representa os dois algarismos restantes do código que podem assumir um valor no intervalo [0 – 9].

Tabela 2 – Natureza e significado das classes de respostas do protocolo HTTP.

Código de <i>Status</i>	Natureza	Significado
1xx	Informativa	Indica que a requisição foi recebida e segue no processo.
2xx	Sucesso	Indica que a requisição atual foi recebida, entendida e aceita.
3xx	Redirecionamento	Indica que uma ação adicional necessita ser tomada para completar a requisição.
4xx	Erro do Cliente	Indica algum desvio na requisição, como ausência de determinado parâmetro ou sintaxe incorreta.
5xx	Erro do Servidor	Indica falha do servidor em retornar uma resposta de uma requisição aparentemente válida, geralmente por indisponibilidade.

Fonte: Adaptado de Nielsen et al. (1999).

2.3 Lado do Servidor

Nesta seção serão abordados os principais aspectos do referencial teórico para o desenvolvimento *server-side*, com uma breve explanação sobre banco de dados, modelagem, API e

a *framework*³ Laravel.

2.3.1 Bancos de Dados

A modelagem de um banco de dados se faz necessária para atender aos requisitos do presente projeto. Elmasri, Navathe e Pinheiro (2009) definem um banco de dados como uma coleção de dados relacionados, no qual os dados se referem a informações implícitas, inteligíveis. Além disso, os autores pontuam que para ser considerado um banco de dados, deve haver uma coerência lógica entre a coleção de dados. Geralmente esta coerência é projetada com específico interesse para alguma aplicação.

Por sua vez, um sistema gerenciador de bancos de dados (SGBD) é uma ferramenta que fornece maior facilidade para que bancos de dados sejam estruturados, que sejam realizadas consultas aos dados, criados novos registros ou modificados registros existentes. Para este trabalho, é proposta a utilização do SGBD relacional PostgreSQL.

2.3.1.1 Modelo Relacional de Bancos de Dados

O modelo relacional representa o banco de dados como uma coleção de relações, que podem ser entendidas como tabelas. Os cabeçalhos da tabela são chamados de atributos, que por sua vez possuem formatos possíveis, denominados domínios. As linhas da tabela são denominadas tuplas e a tabela em si é a própria relação (ELMASRI; NAVATHE; PINHEIRO, 2009).

As tuplas são identificadas por chaves primárias que não podem assumir o valor nulo, de acordo com a restrição de integridade de entidade. Também não deve haver duplicidade de chaves primárias em uma relação (ELMASRI; NAVATHE; PINHEIRO, 2009).

Em paralelo, a restrição de integridade referencial afirma que uma tupla de uma relação que referencia uma tupla de outra relação deve referenciar uma tupla existente. Para que isso ocorra, é utilizada na relação uma “chave estrangeira”, que se referencia à chave primária de outra relação. A chave estrangeira deve obrigatoriamente possuir o mesmo domínio da chave primária da relação referenciada. Ainda, o valor da chave estrangeira deve ocorrer para a chave primária de alguma tupla da relação referenciada, a não ser que este seja nulo (ELMASRI; NAVATHE; PINHEIRO, 2009).

³ Na Engenharia de *Software*, uma *framework* pode ser definida como uma estrutura que une componentes e bibliotecas com o objetivo de prover uma base para implementação de funcionalidades.

2.3.2 Linguagem de Modelagem Unificada

Segundo Guedes (2018), a UML (*Unified Modeling Language*) é uma linguagem de modelagem padronizada utilizada para modelar e visualizar sistemas de *software* com base no paradigma de orientação a objetos. Baseada na notação gráfica, a UML oferece uma abordagem eficaz para representar visualmente as diferentes perspectivas e elementos de um sistema, desde a estrutura estática até o comportamento dinâmico. Dois dos principais artefatos propostos para este projeto são o diagrama de casos de uso e o diagrama de classes.

2.3.2.1 Diagrama de Casos de Uso

De acordo com Guedes (2018), o diagrama de casos de uso é uma ferramenta de modelagem que ajuda a identificar e documentar as interações entre os atores (usuários, sistemas externos) e o sistema em desenvolvimento. Ele descreve as funcionalidades que o sistema oferecerá do ponto de vista dos usuários e como essas funcionalidades se relacionam entre si. Dentre os principais elementos, estão:

1. **Ator:** representa um papel que interage com o sistema. Isso pode ser um usuário humano, outro sistema ou componente externo;
2. **Caso de Uso:** descreve uma funcionalidade específica do sistema, geralmente nomeada de forma descritiva;
3. **Relação de Inclusão:** indica que um caso de uso inclui outro, necessariamente;
4. **Relação de Extensão:** indica que um caso de uso pode estender outro, não obrigatoriamente.

2.3.2.2 Diagrama de Classes

O diagrama de classes é outra ferramenta essencial para o levantamento de requisitos. Ele representa a estrutura estática do sistema, definindo as classes, seus atributos e os relacionamentos entre elas. Dentre os principais elementos segundo Guedes (2018), estão:

1. **Classe:** representa uma entidade no sistema, como um objeto, uma funcionalidade ou um conjunto de dados. As classes contêm atributos e métodos que definem seu comportamento;
2. **Atributos:** propriedades de uma classe que descrevem seu estado;
3. **Relacionamentos:** representam as associações entre classes. Os relacionamentos podem ser de composição, agregação ou associação, e ajudam a definir como as classes interagem.

O diagrama de classes fornece uma visão estrutural do sistema, auxiliando na identificação de requisitos relacionados à organização dos dados, à lógica de negócios e à representação das entidades envolvidas (GUEDES, 2018).

2.3.3 Interface de Programação de Aplicações

Uma Interface de Programação de Aplicações (API) pode ser concebida como um conjunto de funções que permitem com que haja a comunicação entre aplicações distintas sem a necessidade de compreender aspectos detalhados do serviço, mas apenas utilizá-lo. De acordo com Elmasri, Navathe e Pinheiro (2009), no cenário da comunicação entre um cliente e o banco de dados, a API opera como uma biblioteca de funções que se conecta ao banco de dados e realiza uma série de transações, de acordo com os parâmetros recebidos do cliente e a sua regra de negócio implementada. O cliente não necessita realizar os comandos para acessar um banco de dados, pois é a API que os executa em plano de fundo.

2.3.3.1 Laravel

Neste trabalho, é proposta a utilização da *framework* Laravel para o desenvolvimento da API do sistema. A Laravel consiste em uma *framework* de aplicações *web* que utiliza a linguagem de programação PHP e possui funcionalidades que facilitam o desenvolvimento, como uma camada de abstração de banco de dados. Além disso, é altamente escalável e possibilita a criação de interfaces de usuário (LARAVEL, 2022).

A *framework* apresenta diversas soluções para criação e proteção de rotas, validação de requisições, envio de *e-mails* e notificações, autenticação e autorização, testes HTTP e ainda documentação da API desenvolvida (LARAVEL, 2022).

Como é proposta a utilização do estilo de arquitetura de *software* REST, é imposta a restrição apresentada na Seção 2.1.1.1, que desacopla o cliente do servidor. Assim, a Laravel será utilizada apenas para o desenvolvimento de uma API, provendo autenticação e armazenamento/recuperação de dados para um cliente desenvolvido de maneira desassociada.

2.3.3.2 Eloquent ORM

Um ORM, ou *Object-Relational Mapping*, é uma técnica de programação que permite aos desenvolvedores mapear objetos em seu código para tabelas de um banco de dados relacional. Essa abordagem simplifica a interação entre o código da aplicação e o banco de dados, permitindo que os desenvolvedores realizem operações de leitura e gravação de dados sem precisar escrever consultas SQL manualmente. Em vez disso, eles podem usar métodos e classes orientadas a objetos para manipular dados de forma mais intuitiva (ROEBUCK, 2011).

A Laravel oferece suporte a ORM por meio de sua biblioteca Eloquent ORM. No Eloquent, é possível criar modelos para representar tabelas do banco de dados. Um modelo é uma classe que estende a classe base do Eloquent e define os campos da tabela e suas relações com outras tabelas. Isso torna a definição de estruturas de dados mais organizada e legível. O ORM também facilita a definição e manipulação de relações entre modelos. Adicionalmente, permite definir a estrutura do banco de dados utilizando código PHP. O Eloquent é integrado com as migrações, tornando a criação e manutenção do esquema do banco de dados uma tarefa simples e controlada por versão (LARAVEL, 2022).

2.4 Lado do Cliente

Nesta seção serão abordados os principais aspectos do referencial teórico para o desenvolvimento *client-side*. Dessa forma, explana-se conceitos sobre as *frameworks* utilizadas e propriedades relevantes de uma aplicação multiplataforma.

2.4.1 Vue.js

O Vue.js é uma *framework* JavaScript com o propósito de desenvolver interfaces de usuário. A ideia central do Vue.js é integrar em um único arquivo a estrutura da página, a lógica

e a estilização. Este formato de arquivo com sufixo “*.vue” é denominado *Single File Component* e permite que um único arquivo possua código HTML, JavaScript e CSS (VUE.JS, 2022).

Os componentes de arquivo único são importantes para a escalabilidade da interface, visto que estes possuem característica de reusabilidade. O Vue.js possui uma propriedade de renderização declarativa que permite descrever uma saída HTML baseada em seu estado JavaScript. Em comunhão com esta propriedade, o Vue.js possui característica de reatividade, fazendo com que haja um rastreamento de mudanças no estado JavaScript e automaticamente haja uma atualização no documento HTML (VUE.JS, 2022).

No Código-fonte 2.1, o trecho no intervalo de linhas [1 – 3] dentro do marcador `<template>` possui a estrutura HTML da interface de usuário, contendo um botão que ao ser clicado dispara um evento “*click*”, que por sua vez efetua um método que incrementa a variável “*count*”. O trecho encapsulado pelo marcador `<script>` entre as linhas [5 – 13] define a propriedade “*count*” e a inicializa com o valor 0. Por fim, o trecho no intervalo de linhas [15 – 19] dentro do marcador `<style>` possui a estilização do marcador `<button>`, definindo a espessura da fonte como negrito. Quando o valor da propriedade “*count*” é incrementado, uma mudança em seu estado no código JavaScript dispara uma atualização eficiente no documento HTML, alterando o seu valor.

Código-fonte 2.1 – Exemplo de um *Single File Component* Vue.js.

```
1 <template>
2   <button @click="count++">Count is: {{ count }}</button>
3 </template>
4
5 <script>
6 export default {
7   data() {
8     return {
9       count: 0
10    }
11  }
12 }
13 </script>
14
15 <style scoped>
16 button {
17   font-weight: bold;
18 }
19 </style>
```

Fonte: Produção do próprio autor.

2.4.1.1 Quasar

O Quasar é uma *framework* de código livre baseada no Vue.js que possui extensa variedade de componentes, o que permite um desenvolvimento mais otimizado. Os componentes são customizáveis e extensíveis, possibilitando uma adaptação única a cada projeto implementado (QUASAR, 2022).

O propósito da utilização desta *framework* é o de escrever um único código que possa ser implantado de diversas formas, como um site, um aplicativo *mobile* ou um aplicativo *desktop*. Além disso, a escolha do Quasar como opção de desenvolvimento se deve ao fato de o mesmo possuir ampla documentação e comunidade ativa, o que auxilia o trabalho.

2.4.2 Aplicação Multiplataforma

O objetivo deste trabalho é desenvolver uma aplicação que seja multiplataforma. Segundo Nagel (2015), o termo “*multiscreen*”, que pode ser estendido para “multiplataforma”, trata sobre desenvolver uma única aplicação para múltiplas interfaces. O autor ainda divide estas interfaces em quatro específicas: *smartphone*, *tablet*, *desktop* e televisão. Porém, as interfaces que são aplicadas são baseadas nos requisitos dos usuários, seus processos diários e contextos de uso, além do tipo de conteúdo que será disposto na aplicação.

Dessa maneira, utiliza-se de alguns conceitos e premissas para atingir este objetivo. Nesta seção, serão discutidos os conceitos sobre responsividade e PWA, além da metodologia *Mobile First*.

2.4.2.1 Responsividade

Responsividade é um termo utilizado como “adaptabilidade”, no contexto de uma aplicação *web*. Segundo Nagel (2015), o *design* da *web* responsivo utiliza de componentes flexíveis, ajustáveis e escaláveis que respondem a diferentes tamanhos de tela, tipos de dispositivos e ferramentas. Além disso, o conceito de responsividade ainda pode ser ampliado para que o conteúdo da aplicação também se adapte ao contexto de uso do usuário.

A *framework* Quasar possui integrada às suas ferramentas classes CSS que permitem com que haja a identificação da plataforma em que a aplicação está sendo executada, *mobile* ou *desktop*. Além disso, existem classes que permitem a identificação da largura do dispositivo,

altura e orientação. A Tabela 3 expõe as classes de acordo com a largura mínima e largura máxima da tela do dispositivo. Com essas informações, é possível adaptar a disposição dos componentes para uma variedade de telas. Ressalta-se ainda que é possível adicionar outras classes com larguras personalizadas adaptáveis ao projeto.

Tabela 3 – Classes CSS da *framework* Quasar relacionadas à largura da tela que executa a aplicação.

Tamanho da Janela	Nome da Classe	Largura Mínima	Largura Máxima
Muito Pequena	xs	0px	599.99px
Pequena	sm	600px	1023.99px
Média	md	1024px	1439.99px
Larga	lg	1440px	1919.99px
Muito Larga	xl	1920px	Infinito

Fonte: Produção do próprio autor.

2.4.2.2 *Progressive Web App*

Um aplicativo *web* progressivo (PWA) possui recursos modernos que garantem uma experiência ao usuário como a de um aplicativo nativo, sendo hospedados por servidores *web* e acessíveis via navegador. Este é denominado progressivo pois deve trabalhar para qualquer tipo de usuário em qualquer navegador (OSMANI, 2015).

Em termos gerais, a escolha do desenvolvimento de um PWA se deve ao fato de ser uma aplicação responsiva que atende a diversos dispositivos, acessível de maneira prática via URL em navegadores, se comportando como um aplicativo nativo em vários dispositivos e com a possibilidade de instalação sem a necessidade de publicação em lojas de aplicativos (OSMANI, 2015).

Destaca-se que o Quasar possui prática implementação de um PWA por meio de configurações no projeto.

2.4.2.3 *Mobile First*

No cenário da manutenção de computadores dentro do projeto Solidariedade Digital, se faz útil a utilização de um dispositivo móvel para o registro ou atualização de informações, visto que há maior praticidade em realizar esta ação no local da tarefa, ao invés de se

deslocar até um *desktop* próximo. Com isso, é proposta a utilização do princípio *Mobile First* para o desenvolvimento do cliente.

Este princípio parte do desenvolvimento da aplicação inicialmente para dispositivos móveis, que geralmente possuem telas menores do que *desktops* e televisões. O maior ganho do desenvolvimento *Mobile First* é que o desenvolvedor tende a definir melhor o escopo do projeto, impondo maior seletividade e melhor arranjo de conteúdos de acordo com as necessidades dos usuários. Posteriormente, ao implementar o projeto para dispositivos com telas maiores, há maior facilidade em dispor as informações, visto que há maior espaço e já foi realizado um trabalho de tornar a aplicação clara e objetiva (NAGEL, 2015).

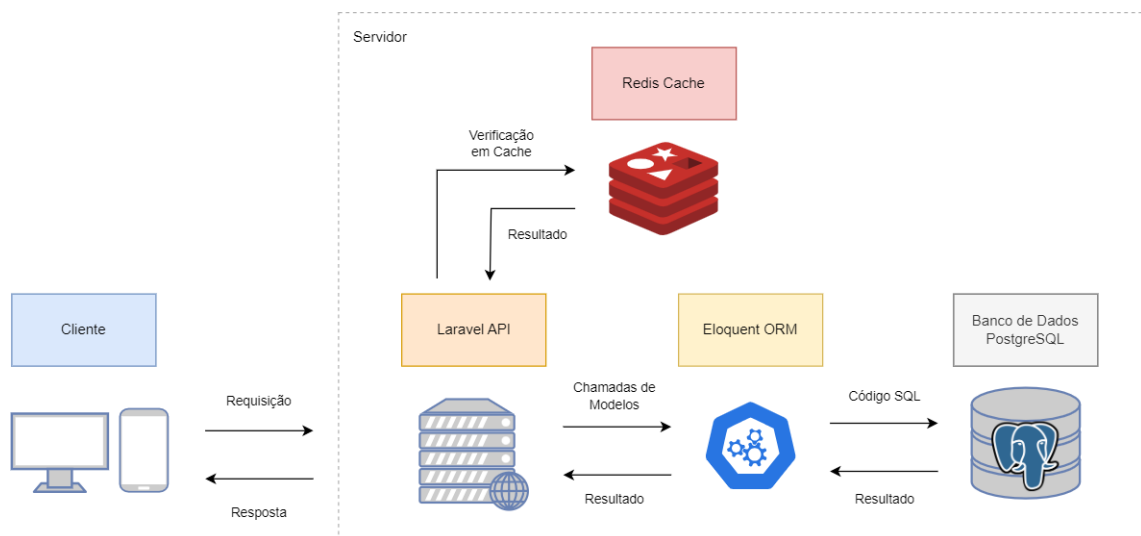
3 METODOLOGIA E ETAPAS DE DESENVOLVIMENTO

No presente capítulo deste projeto, aborda-se o desenvolvimento do sistema proposto, enfocando a arquitetura, modelagem utilizando UML, testes unitários, e as implementações tanto no lado do servidor quanto no lado do cliente. A análise detalhada da arquitetura destaca escolhas fundamentais para a eficiência e escalabilidade do sistema. A modelagem UML é discutida como ferramenta visual essencial, enquanto os testes unitários são abordados como garantia da qualidade do código. O desenvolvimento do lado do servidor com PHP Laravel é destacado pela eficiência e segurança, e o lado do cliente é implementado utilizando Vue.js para criar uma experiência do usuário dinâmica e intuitiva.

3.1 Arquitetura e Análise de Requisitos

A arquitetura do sistema segue os princípios e restrições do estilo de arquitetura REST e pode ser analisada na Figura 4. Um cliente realiza requisições HTTP para a API. A API primeiro busca informações no *cache*: caso esteja presente, a resposta é retornada, caso não esteja, as informações são buscadas no banco de dados. A concepção da consulta no banco de dados PostgreSQL utiliza da ferramenta de mapeamento objeto-relacional Eloquent ORM. Através dessa ferramenta, métodos de criação, consulta, atualização e remoção de objetos PHP são convertidos em declarações SQL.

Figura 4 – Arquitetura do sistema.

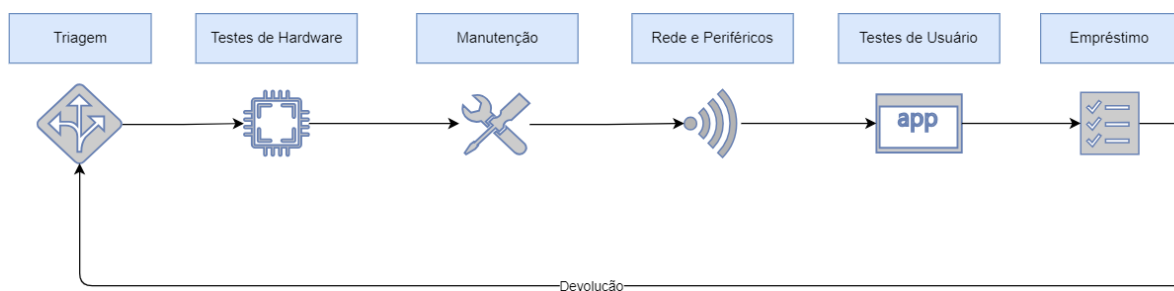


Fonte: Produção do próprio autor.

O desacoplamento do lado do servidor com o cliente reforça a restrição cliente-servidor do estilo de arquitetura REST.

Em um primeiro momento, para a concepção do projeto é necessário reproduzir o processo de manutenção, tanto no lado do servidor quanto do lado do cliente. Neste ponto, é importante ressaltar que o processo de manutenção já existe e funciona no projeto, não sendo escopo deste projeto realizar alterações/melhorias. As etapas de manutenção são implementadas seguindo o exposto na Figura 5.

Figura 5 – Diagrama das fases de manutenção.



Fonte: Produção do próprio autor.

1. **Triagem:** Inicialmente, o computador chega ao laboratório e é higienizado. É realizada uma breve inspeção superficial para verificar se aquele dispositivo é funcional. Nessa etapa, o membro preenche as seguintes informações:
 - a) Patrimônio (caso exista);
 - b) Licença do Sistema Operacional (caso não seja de código aberto);
 - c) Fabricante.
2. **Testes de *Hardware*:** Inicialmente, será solicitado que o usuário adicione as especificações de cada componente do computador. É solicitado que o usuário pontue se os *hardwares* funcionam corretamente ou não. Caso algum destes componentes não funcione corretamente, para seguir para a próxima etapa é necessário que o mesmo seja reparado ou substituído por um funcional. Como aspecto de projeto, são criados objetos referentes a cada componente de *hardware*. Esses componentes, além dos atributos de suas especificações, possuem um atributo “funcional”.
3. **Manutenção:** Nesta etapa estão incluídas possíveis manutenções, que incluem:
 - a) Instalação de programas;
 - b) Instalação de sistema operacional;
 - c) Formatação de discos;

d) Troca de bateria (no caso de *notebooks*).

Estas manutenções não são obrigatórias para seguir para a próxima fase, devendo ser realizadas de acordo com as condições do computador.

4. **Rede e Periféricos:** Assim como no teste de *hardware*, aqui serão cadastrados os componentes de rede e periféricos. Dentre os quais:

- a) Saída HDMI;
- b) Saída VGA;
- c) Saída DVI;
- d) Adaptador de Rede Local;
- e) Adaptador de Rede Sem Fio;
- f) Entrada e Saída de Áudio;
- g) CD-ROM.

Nesta fase não serão criados objetos referentes a cada componente de rede/periférico. Estes itens serão atributos do objeto computador e seguirão o seguinte princípio: caso estes estejam presentes e funcionais, possuirão o valor *booleano true*. Caso estejam presentes e não funcionais, possuirão o valor *booleano false*. Caso não estejam presentes, o valor será nulo. Logo, será possível criar as regras apenas atualizando estes campos. Essa escolha foi dada pois é muito menos comum realizar a transferência destes adaptadores para outros computadores, além de que no processo já implementado não são registradas características específicas destes itens.

5. **Testes de Usuário:** são realizados testes de funcionalidades básicas para comprovar que o computador possui desempenho mínimo para a atividade fim: *(i)* boot automático e inicialização; *(ii)* atalhos *desktop* funcionais; *(iii)* data correta e *(iv)* avaliação de desempenho em aplicações como GSuite, Youtube e Wine.

Passar ou não desta fase estará a cargo do responsável da etapa, que avaliará se o computador está apto para ser liberado para empréstimo conforme as necessidades previstas.

Ainda sobre cada uma destas etapas, são requisitos importantes:

- Cada etapa possuirá um responsável, que deve ser um usuário da aplicação e membro do LAETI;
- Cada computador possuirá um atributo “fase atual” que realizará o controle do processo de manutenção;

- O computador só está apto a ser emprestado depois que passar por todas as fases;
- Após o retorno de um empréstimo, o computador volta para a etapa de triagem novamente;
- Cada usuário poderá adicionar observações ao computador, referenciando a fase em que ele se encontra.

A partir da concepção do processo de manutenção, o sistema então foi separado em módulos, para delimitar melhor os seus requisitos. Estes módulos são: (i) painel; (ii) computadores; (iii) componentes; (iv) empréstimos; (v) tomadores de empréstimo e (vi) usuários.

Com a utilização do diagrama de casos de uso, foi possível definir melhor a interação de cada ator no sistema com cada um desses módulos. Neste caso, os atores do sistema são inicialmente dois: membro da equipe de manutenção e administrador. A interação destes com o sistema é praticamente a mesma, excetuando-se apenas o módulo de usuários, cuja gestão cabe unicamente ao administrador do sistema. A seta nos diagramas da Figura 6 que segue na direção do administrador ao membro indica uma relação de especialização, ou seja, um administrador é uma especialização de um membro (usuário do sistema).

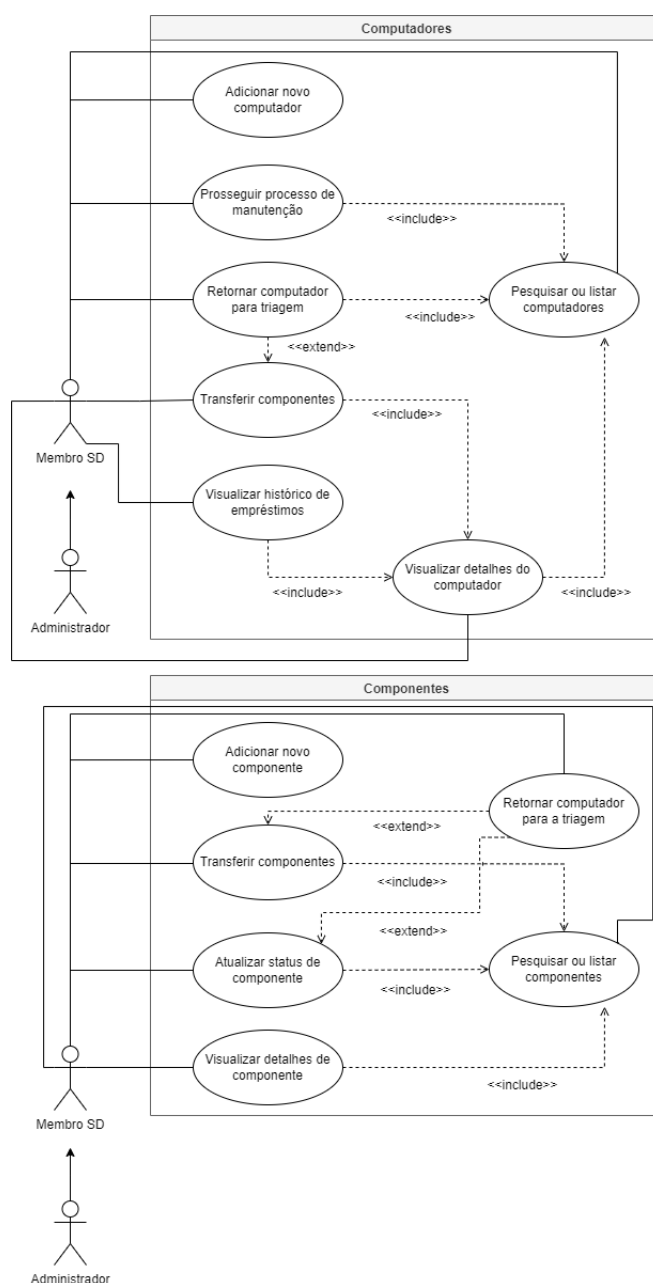
No módulo de computadores exposto, ambos os atores podem:

1. Pesquisar ou listar computadores;
2. Visualizar detalhes de um computador em específico;
3. Registrar novos computadores;
4. Prosseguir processos de manutenção, o que inclui pesquisar ou listar computadores;
5. Retornar um computador para triagem, o que inclui pesquisar ou listar computadores;
6. Transferir componentes do computador, o que inclui visualizar detalhes de um computador em específico, que inclui pesquisar ou listar computadores e que pode resultar em um retorno para triagem;
7. Visualizar histórico de empréstimos do computador, o que inclui visualizar detalhes de um computador em específico, que inclui pesquisar ou listar computadores;

Ao transferir um componente de um computador, pode ser que este seja movido para a etapa de triagem novamente, caso este seja o único componente funcional daquele tipo presente no dispositivo, por isso a utilização da relação “*extend*”.

O módulo de componentes também segue a mesma denotação. Assim como no caso do computador, a transferência de um componente por meio deste módulo também pode interferir no processo de manutenção de um computador em específico. Além disso, alterar o *status* de um componente de funcional para não funcional, por exemplo, pode também levar um computador de volta para a etapa de triagem.

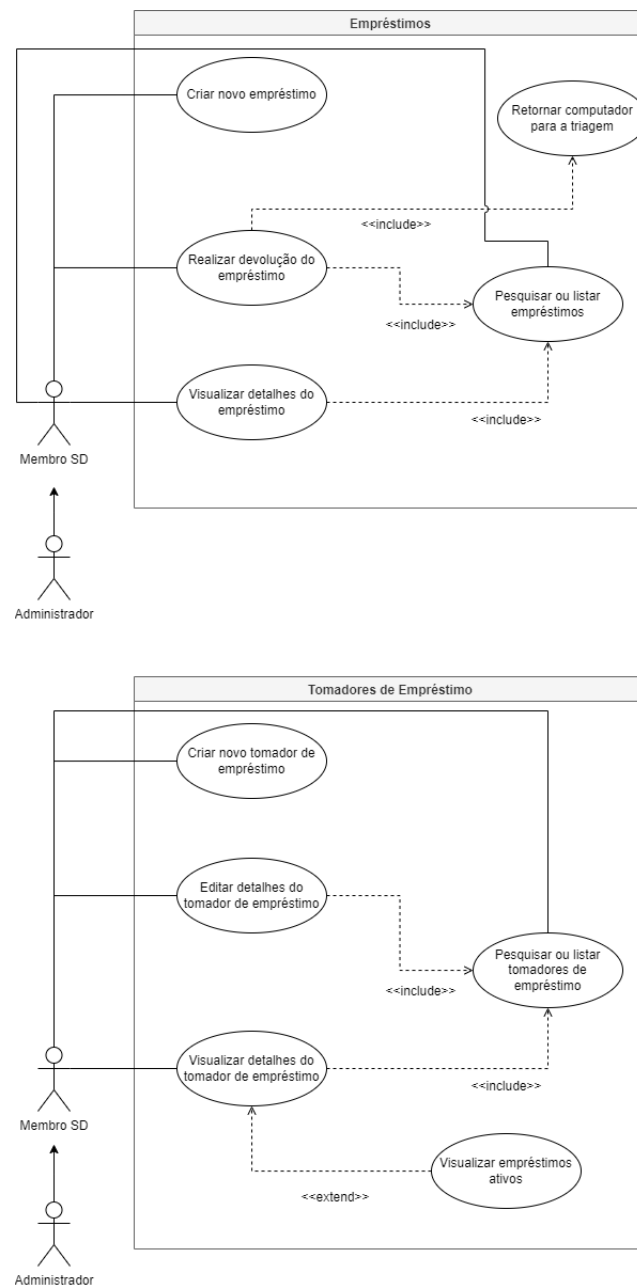
Figura 6 – Diagrama de casos de uso: computadores e componentes.



Fonte: Produção do próprio autor.

A Figura 7 traz os casos de uso para os módulos de empréstimos e tomadores de empréstimos. No módulo de empréstimos, ambos atores podem: registrar novos empréstimos, realizar devoluções e visualizar detalhes do empréstimo. No módulo de tomadores de empréstimo, deve ser possível listar, cadastrar e editar registros.

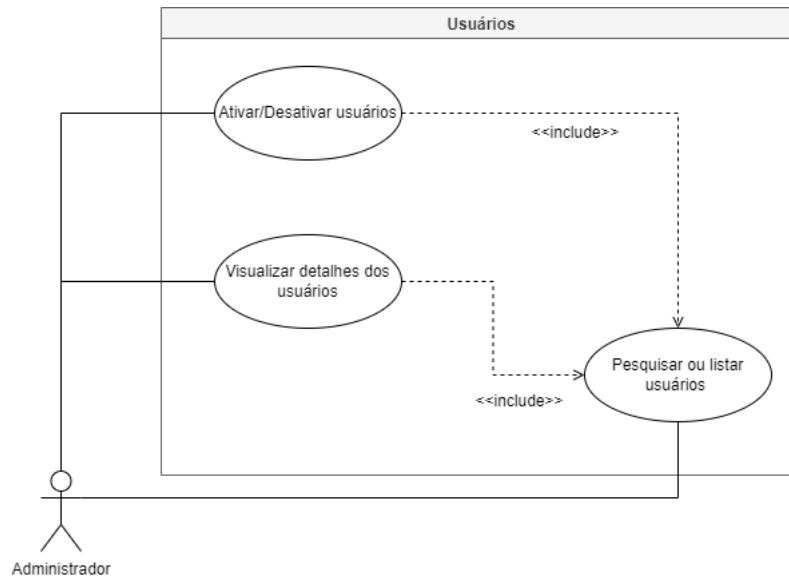
Figura 7 – Diagrama de casos de uso: empréstimos e tomadores de empréstimos.



Fonte: Produção do próprio autor.

Por fim, a Figura 8 exibe os casos de uso para o módulo de usuários. Este é acessível apenas para usuários que possuem o papel de administrador. O administrador será capaz de listar usuários, visualizar detalhes e ativar/desativar suas licenças.

Figura 8 – Diagrama de casos de uso: usuários.

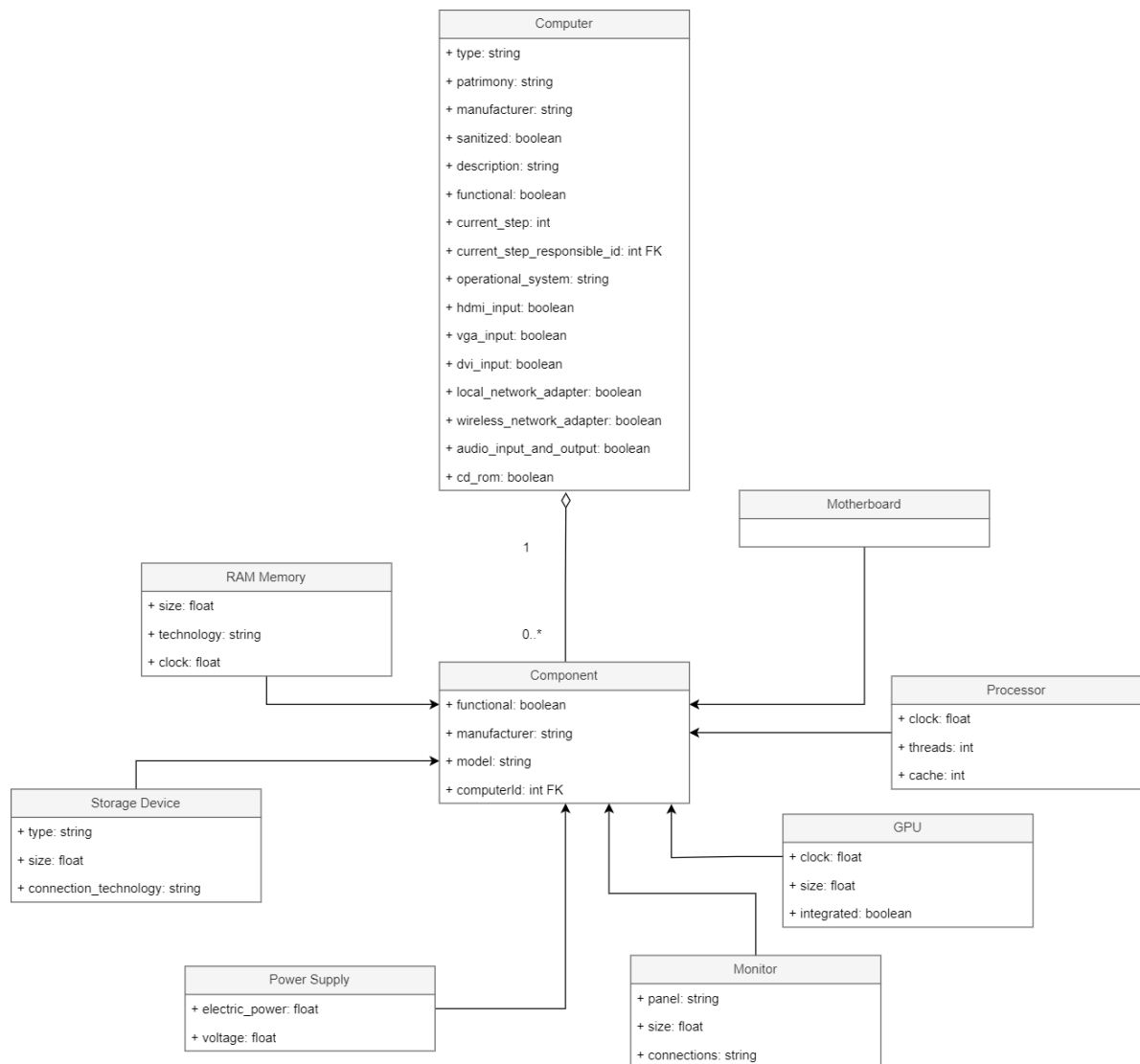


Fonte: Produção do próprio autor.

A utilização de um mapeador objeto-relacional na arquitetura do sistema faz com que haja mais senso em criar um modelo baseado em classes. Dessa forma, nesta altura do texto serão apresentados os diagramas de classes referentes a cada modelo. O diagrama de classes completo foi separado em pequenas partes lógicas para facilitar a compreensão e exibição no texto.

Primeiramente, há o relacionamento entre computador e componentes, apresentados na Figura 9. Os componentes possuem uma relação de agregação com o computador. Cada componente em específico é uma especialização da classe componente e herda suas propriedades fundamentais: funcional, fabricante, modelo e identificador. O campo identificador é uma chave estrangeira utilizada para relacionar computadores e componentes. O computador também possui uma chave estrangeira que é relacionada com um usuário, para identificar o membro do projeto que é o atual responsável por aquele computador. Cada especialização de componente possui suas propriedades específicas de acordo com sua natureza. O relacionamento entre computador e componente é um computador para zero ou muitos componentes. Em outras palavras, um computador pode existir sem nenhum componente e pode possuir mais de um componente.

Figura 9 – Diagrama de classes: computadores e componentes.



Fonte: Produção do próprio autor.

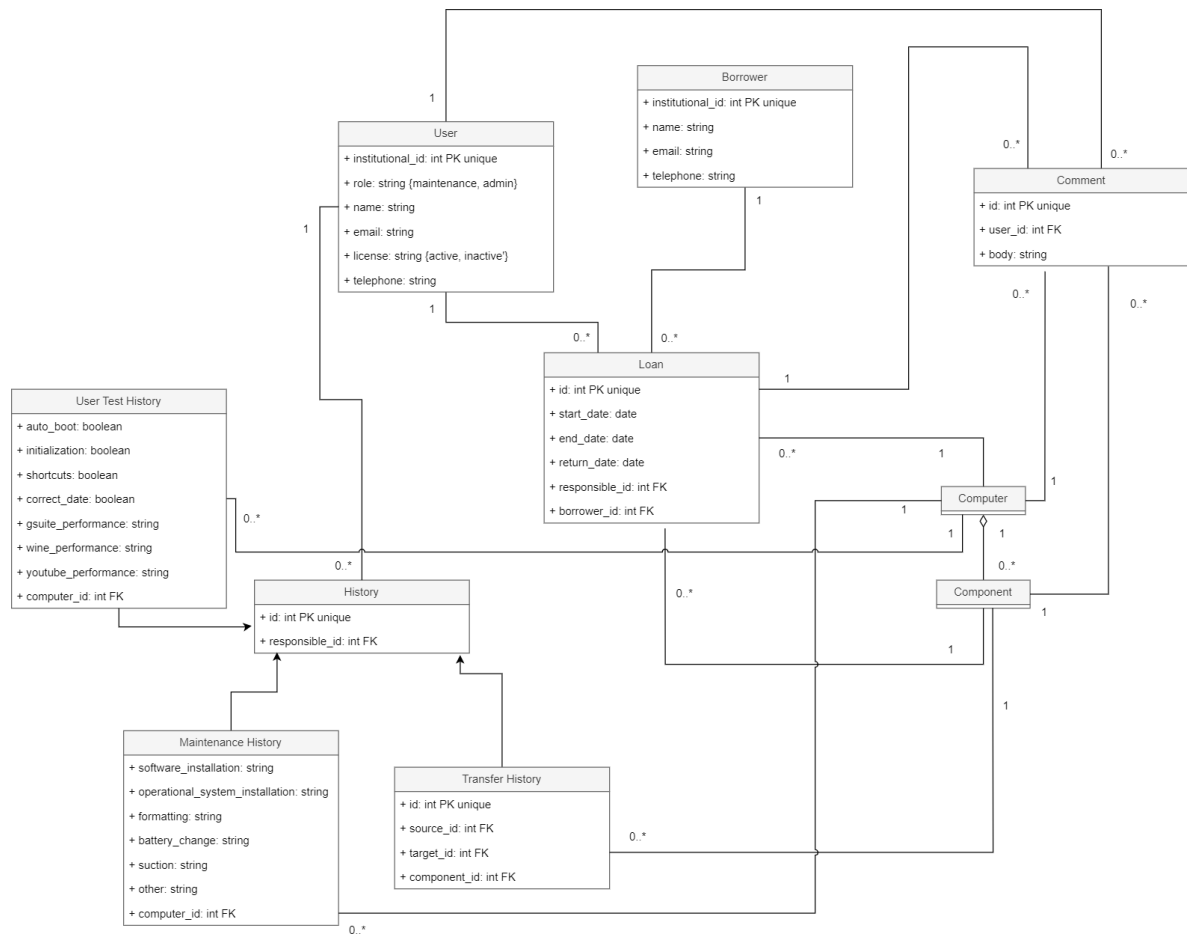
Além das classes expostas, na Figura 10 são exibidos os relacionamentos entre empréstimo e computadores/componentes, comentários e históricos. Um empréstimo pode possuir apenas um computador ou componente por vez. Além disso, o empréstimo possui um relacionamento com um tomador de empréstimo e também com um membro do projeto, responsável por realizar o registro daquele empréstimo.

Um comentário pode ser feito para computadores, componentes e empréstimos. Essa classe foi criada com a finalidade de permitir que o usuário insira observações sobre determinados assuntos.

Por fim, existem as classes de histórico: histórico de transferências, histórico de manutenção e histórico de testes de usuário. O histórico de transferências possibilita ao usuário maior

rastreabilidade de um componente, sabendo exatamente de qual computador ele veio. Já o histórico de manutenção e o histórico de testes de usuário são ferramentas que armazenam dados de etapas de manutenção passadas. Todos esses históricos envolvem um relacionamento com o usuário, que é o responsável por executar uma dessas ações.

Figura 10 – Diagrama de classes: empréstimos, comentários e históricos.



Fonte: Produção do próprio autor.

3.2 Lado do Servidor

Nesta seção será elucidada toda a etapa de desenvolvimento do lado do servidor, utilizando a *framework* PHP Laravel, a partir da explanação do código implementado. Em alguns casos, serão suprimidos trechos de código que são por muitas vezes repetitivos, alterando apenas alguns parâmetros. O código completo está disponível para consulta no *github* (SOUSA, 2022a).

3.2.1 Autenticação

A autenticação do sistema foi desenvolvida utilizando o serviço de autenticação Sanctum, incluso na Laravel. Inicialmente, é concebido o modelo de usuário através da criação da sua migração. No Código-fonte 3.1 é possível verificar os atributos que são migrados pelo mapeamento objeto relacional para a tabela *users*. O primeiro atributo declarado na linha 17 se refere à chave primária da tabela, a matrícula do membro do projeto. O atributo é seguido pelo nome, *e-mail* e telefone, que são do tipo *string*. O *e-mail* deve ser único na tabela, assim como a chave primária.

O próximo atributo a ser analisado é o carimbo de data/hora de verificação de *e-mail*. Esta declaração cria uma coluna do tipo *TIMESTAMP* na tabela, que armazena data e hora de verificação com informações de fuso horário. Esta informação pode ser nula, o que representa que o *e-mail* do usuário ainda não foi verificado. Além desse carimbo de data/hora, na linha 26, são criadas outras duas colunas referentes à data e hora de criação e de atualização de determinado registro.

Na linha 22, é declarada a coluna referente à senha. A senha é criada com o tipo *string* e armazena um *hash* de 24 *bytes* utilizando a codificação na base 64. A Laravel utiliza o algoritmo *bcrypt* para criptografar credenciais utilizadas para autenticação do usuário.

Por fim, são declarados dois atributos do tipo enumeração: papel e licença. O papel pode assumir dois valores: manutenção ou administrador. Enquanto isso, a licença do usuário pode ser: ativa ou inativa.

Código-fonte 3.1 – Trecho do arquivo PHP contendo as migrações da tabela de usuários.

```
14 public function up()
15 {
16     Schema::create('users', function (Blueprint $table) {
17         $table->id('institutional_id');
18         $table->string('name');
19         $table->string('email')->unique();
20         $table->string('telephone');
21         $table->timestamp('email_verified_at')->nullable();
22         $table->string('password');
23         $table->enum('role', ['maintenance', 'admin']);
24         $table->enum('license', ['active', 'inactive']);
25         $table->rememberToken();
26         $table->timestamps();
27     });
28 }
```

Fonte: Produção do próprio autor.

Antes de explanar o controlador da autenticação, é importante debater sobre as regras de validação da requisição. No Código-fonte 3.2 é exposto um trecho do arquivo *UserRequest.php*. Todos os atributos declarados são obrigatórios, o que é representado pela validação *required*. Além disso, é verificado se estes atributos são do tipo *string*. Para o nome e o *e-mail*, os valores presentes na requisição não podem possuir um tamanho maior do que 255 caracteres. No caso do *e-mail*, ainda é utilizada a validação “*email*” integrada à Laravel que utiliza expressões regulares para validar se aquele valor segue os padrões de um *e-mail* válido. A regra *unique:users* verifica se o *e-mail* e a senha são valores únicos em sua respectiva coluna da tabela *users* antes de prosseguir com a requisição. Por fim, a senha deve possuir no mínimo 8 caracteres, enquanto o telefone deve possuir no mínimo 9. A matrícula deve possuir exatamente 10 caracteres.

Em qualquer caso de inconsistência em uma das regras, é retornada uma resposta HTTP com código 400 ao usuário antes mesmo de prosseguir ao controlador.

Código-fonte 3.2 – Regras de validação da requisição para criação de um novo usuário.

```
42 public function rules()
43 {
44     return [
45         'name' => 'required|string|max:255',
46         'email' => 'required|string|email|max:255|unique:users',
47         'password' => 'required|string|min:8',
48         'institutional_id' => 'required|string|size:10|unique:users',
49         'telephone' => 'required|string|min:9'
50     ];
51 }
```

Fonte: Produção do próprio autor.

No arquivo *AuthController.php* há toda a lógica de registro e autenticação de usuários, o que pode ser observado no Código-fonte 3.3.

A primeira função é a de registro, que possui sua assinatura na linha 3. Ela recebe como parâmetro uma requisição do tipo “*UserRequest*” que é validada na linha 4. Caso haja alguma inconsistência nos valores enviados na requisição, um erro é lançado e uma resposta é retornada com código 400. Caso não haja nenhum erro, o código segue e cria um usuário utilizando a sintaxe do Eloquent ORM, passando como parâmetro os dados obtidos da requisição já validados. A senha é criptografada com o método “*Hash::make*”. Após a criação, é disparado um evento que envia um *e-mail* de confirmação ao usuário. Por fim, uma resposta é enviada ao cliente com uma mensagem confirmando o registro e código 200.

A função “*login*” que se inicia na linha 21 é responsável por realizar a autenticação do usuário. Inicialmente, é realizada uma tentativa de autenticação utilizando os dados da requisição. Caso a tentativa falhe, uma mensagem com o código 401 (não autorizado) é retornada para o cliente. Caso haja sucesso, o usuário que está se autenticando é recuperado na linha 28. Com os dados do usuário, são realizadas duas validações: se o usuário possui o *e-mail* verificado e se possui licença ativa. Se o usuário não possuir os dois requisitos, o servidor também retornará uma resposta com erro 401. Se houver sucesso, é criado um *bearer token* e retornado com sucesso ao usuário. Este *token* deve ser utilizado em todas as requisições do usuário.

Por último, na linha 51 há a função utilizada para revogar os *tokens* de acesso, executada quando o usuário autenticado realiza a ação de “sair” do sistema. Em termos gerais, o *token* do usuário é deletado e nas requisições subsequentes não será mais válido, sendo necessário que o usuário realize a autenticação novamente para gerar outro *token*.

Código-fonte 3.3 – Controlador de autenticação.

```
1 class AuthController extends Controller
2 {
3     public function register(UserRequest $request) {
4         $validatedData = $request->validated();
5
6         $user = User::create([
7             'name' => $validatedData['name'],
8             'email' => $validatedData['email'],
9             'institutionalId' => $validatedData['institutional_id'],
10            'telephone' => $validatedData['telephone'],
11            'password' => Hash::make($validatedData['password'])
12        ]);
13
14        event(new Registered($user));
15
16        return response()->json([
17            'message' => "Usuario criado com sucesso! Um e-mail foi
18                encaminhado para o seu endereco para que voce possa
19                confirma-lo."
20        ], 200);
21    }
22
23    public function login(Request $request) {
24        if (!Auth::attempt($request->only('email', 'password'))) {
25            return response()->json([
26                'message' => 'Os dados de acesso nao existem ou estao
27                    incorretos.'
```

```
27
28     $user = User::where('email', $request['email'])->firstOrFail
        ();
29
30     if ($user['email_verified_at'] == null) {
31         return response()->json([
32             'message' => 'E-mail nao verificado.'
33         ], 401);
34     }
35
36     if ($user['license'] != 'active') {
37         return response()->json([
38             'message' => 'Voce nao possui uma licenca ativa.
                Contate um administrador para conceder acesso ao
                sistema.'
39         ], 401);
40     }
41
42     $token = $user->createToken('auth_token')->plainTextToken;
43
44     return response()->json([
45         'user' => $user,
46         'access_token' => $token,
47         'token_type' => 'Bearer',
48     ]);
49 }
50
51 public function revokeAccessTokens () {
52
53     Auth::user()->tokens()->delete();
54
55     return response()->json([
56         'message' => 'Logout realizado com sucesso.'
57     ]);
58 }
59 }
```

Fonte: Produção do próprio autor.

Como parte da autenticação, ainda há um controlador responsável pela verificação do *e-mail*, no arquivo *EmailVerificationController.php*. A verificação se dá através de uma requisição do tipo GET, passando como parâmetro de rota um *hash* criptografado através do algoritmo SHA1. Primeiramente, é verificado se o usuário já possui o *e-mail* verificado, e se não, o parâmetro da rota é comparado com o *e-mail* do usuário criptografado. Com isso, a coluna “*textite-mail verificado em*” é preenchida com a data/hora da verificação e um evento é disparado enviando um *e-mail* de confirmação ao usuário. Ainda neste

controlador, há uma função para reenviar o *e-mail* de verificação, validando se o usuário já não possui *e-mail* verificado antes, e posteriormente disparando o evento de verificação.

3.2.2 Computadores e Componentes

O desenvolvimento dos computadores e seus componentes do lado do servidor, em termos gerais, possui a mesma estrutura de composição:

- Arquivo de modelo, responsável por definir as interações entre o ORM e as tabelas, além de relacionamento entre modelos;
- Migração, que define seus atributos e tipos;
- Requisição, que valida os parâmetros inseridos pelo usuário;
- Controlador, que possui as lógicas e regras de negócio referentes às operações de CRUD (*Create*, *Read*, *Update* e *Delete*) de cada componente;
- *Seeders* e *Factories*, responsáveis por popular as tabelas no banco de dados para eventuais testes.

Por tamanha similaridade no desenvolvimento destes componentes, nesta serão apresentados apenas o modelo de um componente, o processador, e o do computador, visto que para os componentes a lógica é praticamente a mesma, apenas alterando seus atributos e validações.

A migração do modelo do processador segue a mesma linha de construção do usuário, além das regras de validação. O modelo do processador traz a declaração das propriedades que são atribuíveis em massa, ou seja, que podem ser preenchidas utilizando o método “*fill*” do ORM. Além disso, todos os componentes possuem um atributo computado que define se o componente está emprestado ou não. Para isso, são definidas as regras: se o componente está em um computador, é verificado se aquele computador possui um empréstimo cuja data de retorno é nula, ou seja, ainda não foi devolvido; se o componente não está em um computador, é verificado se aquele item possui algum empréstimo em aberto. Por fim, o arquivo de modelo também traz funções que recuperam os relacionamentos do componente. No caso do relacionamento com o computador, um processador possui a relação um para um. No caso de empréstimos, comentários e históricos de transferência, um processador possui a relação um para muitos.

O controlador do processador traz as lógicas de CRUD do componente. Estas lógicas seguem a mesma para todos os componentes e cada função pode ser brevemente analisada abaixo, para o caso do processador:

- *Index*: função responsável por listar todos os componentes. A requisição recebe filtros como parâmetro, que podem ser do tipo exato ou do tipo similar. A consulta é ordenada de forma decrescente pela coluna de última atualização, recuperando também todo o histórico de transferências daquele componente e seus comentários. Os registros são paginados por padrão com o quantitativo de 10 por página.
- *Store*: função responsável por criar um novo registro de processador. Inicialmente a requisição é validada, e então parte para a criação de um objeto da classe “*Processor*”. Ao declarar as propriedades atribuíveis em massa no arquivo de modelo, com apenas a utilização do método “*fill*”, todos os valores são atribuídos automaticamente.
- *Show*: recupera apenas um processador por meio do seu “*id*”. Caso não encontre nenhum registro, retorna uma resposta HTTP de código 404.
- *Update*: atualiza um registro de um processador na tabela. A função pesquisa pelo processador utilizando seu “*id*”, valida a requisição e preenche o registro com os novos dados, ainda sem salvar. Antes de salvar o registro, é verificado se o processador foi transferido de um computador para outro ou se o campo “funcional” foi alterado. Em qualquer um desses casos, pode ser que o computador que este processador pertence ou pertencia tenha que ser retornado para triagem, o que é feito automaticamente.
- *Destroy*: deleta o registro de um processador utilizando seu “*id*”. Caso não encontre nenhum registro, retorna uma resposta HTTP de código 404.

O desenvolvimento do modelo do computador segue os mesmos padrões implementados nos componentes, descritos anteriormente, alterando apenas seus atributos e relacionamentos. A maior divergência está no controlador, em que são implementadas as regras de movimentação de etapa dos computadores.

Em todas as funções que implementam as etapas, é verificado antes se o computador está na etapa adequada para avançar. Não é possível, por exemplo, avançar da etapa de triagem para a etapa de testes de usuário. Na etapa de testes de *hardware*, o usuário só consegue avançar de etapa caso haja ao menos um componente obrigatório de cada tipo funcional. A etapa de manutenção cria um histórico de manutenção para o computador por meio do controlador *MaintenanceHistoryController.php*. Esta implementação não é obrigatória, visto que em determinadas situações não se faz necessária a manutenção, por

isso a função apenas avança a sua etapa. Caso haja manutenção, a requisição também é validada de acordo com as regras implementadas. O mesmo ocorre para a etapa de rede e periféricos e a etapa de testes de usuário. Não foram criadas limitações nessas etapas pois é da responsabilidade do membro julgar se o computador está adequado ou não para seguir de etapa.

3.2.3 Modelos Secundários

Dentre os modelos secundários na aplicação do lado do servidor, existem os modelos de empréstimo, tomador de empréstimo e comentário, além de um controlador que é responsável por trazer as informações pertinentes ao painel inicial.

Os modelos de tomador de empréstimo e de comentário seguem a mesma linha de desenvolvimento de outros modelos aqui apresentados. O comentário possui basicamente um corpo e está atrelado a um modelo “comentável”, que pode ser um computador, componente ou empréstimo. O tomador de empréstimo possui o desenvolvimento similar ao do usuário, excluindo a autenticação e outros atributos. Assim como um usuário possui diversos empréstimos associados como membro que efetivou o empréstimo, o tomador de empréstimo possui este relacionamento, mas no formato de recepção do item de empréstimo.

O controlador de empréstimo implementa uma função responsável por criar um novo empréstimo. Após a validação da requisição, é verificado se o usuário informou uma data de fim posterior à data de início. Além disso, como parâmetro da requisição, é passado o tipo do item a ser emprestado: um componente específico ou um computador. Dessa forma, também é verificado se este tipo existe. Caso exista, o item de empréstimo é recuperado e passa para uma análise posterior:

- Caso seja um computador: é verificado se o computador não está emprestado atualmente, se é funcional e se encontra na etapa de número 6, que indica que o computador está pronto para empréstimo;
- Caso seja um componente: é verificado se o componente não está atualmente emprestado e se é funcional. Além disso, caso o componente esteja associado a um computador, este não pode ser emprestado de maneira avulsa.

Cumpridos todos os requisitos, um novo empréstimo é criado e uma resposta de sucesso é retornada ao usuário.

Em sequência, para a construção do painel, há uma função em um controlador que compila algumas estatísticas da aplicação. Nela são contados os computadores de acordo com a sua respectiva etapa de manutenção, realizados cálculos para obtenção do número de computadores que se encontram emprestados e o número de computadores disponíveis, é criada uma relação de contagem de componentes por tipo e também são recuperados os computadores que estão sob responsabilidade do usuário atual, retornando também via requisição do painel. Todas essas informações são expostas na página inicial da aplicação no lado do cliente.

3.2.4 Rotas e *Middlewares*

Para efetivo acesso às funções dos controladores, no arquivo “api.php” são definidas as rotas relativas a cada funcionalidade. Neste arquivo são declaradas as rotas da API e suas interações com os controladores. Na declaração das rotas é definido: o tipo de requisição, o controlador atrelado, os *middlewares*, os parâmetros da rota e o nome da função.

Inicialmente, no Código-fonte 3.4 entre as linhas [34-37] são declaradas as rotas desprotegidas, ou seja, que podem ser acessadas quando o usuário não está autenticado. São elas: *login*, registro, verificação de *e-mail* e reenvio de verificação de *e-mail*. Os parâmetros de cada rota aparecem entre chaves, no primeiro argumento da declaração da rota, que define a URI.

A seguir, é declarado um agrupamento de rotas que só podem ser acessadas após três *middlewares*: *auth:sanctum*, *verified* e *active*. O *middleware auth:sanctum* é pré-definido pela Laravel e possui a função de verificar se o usuário está autenticado. O *middleware verified* também é pré-definido e verifica se o usuário possui o *e-mail* verificado, enquanto o *middleware active* verifica se o usuário possui licença ativa. Em termos gerais, dentro deste grupo de rotas, apenas usuários autenticados, ativos e verificados podem realizar requisições. As rotas declaradas dentro deste *middleware* foram ocultadas apenas para fins de apresentação no texto, sendo mantidas apenas as rotas do controlador do computador. As rotas detalhadas podem ser visualizadas no *github*.

Por último, entre as linhas [56-61], há um *middleware* aninhado aos outros três definidos anteriormente, que verifica se o usuário é administrador. Com isso, apenas usuários que possuem o papel de administrador podem listar usuários e alterar a sua licença para ativa ou inativa. Os *middlewares* que garantem que o usuário está ativo e que é administrador estão definidos nos arquivos *EnsureLicenseIsActive.php* e *EnsureRoleIsAdmin.php*, respectivamente no Código-fonte 3.5 e Código-fonte 3.6. A função *handle* em ambos os códigos

verifica se a condição de cada *middleware* é falsa, e caso seja, retorna uma resposta HTTP de código 403 (não autorizado). Caso contrário, segue com a requisição para o controlador.

Código-fonte 3.4 – Trecho do arquivo de declaração de rotas da API.

```

34 Route::post('register', [AuthController::class, 'register']);
35 Route::post('login', [AuthController::class, 'login']);
36 Route::post('resend-email-verification', [EmailVerificationController
    ::class, 'resendVerificationEmail']);
37 Route::get('verify-email/{id}/{hash}', [EmailVerificationController::
    class, 'verify'])->name('verification.verify');
38
39 Route::middleware('auth:sanctum', 'verified', 'active')->group(
    function () {
40     Route::controller(ComputerController::class)->group(function () {
41         Route::get('computers', 'index');
42         Route::get('computer/{id}', 'show');
43         Route::post('computer', 'store');
44         Route::delete('computer/{id}', 'destroy');
45
46         Route::put('computer/{id}/sorting-step', 'sortingUpdate');
47         Route::put('computer/{id}/hardware-tests-step', '
            hardwareTestsUpdate');
48         Route::put('computer/{id}/maintenance-step', '
            maintenanceUpdate');
49         Route::put('computer/{id}/network-and-peripherals-step', '
            networkAndPeripheralsUpdate');
50         Route::put('computer/{id}/user-tests-step', 'userTestsUpdate'
            );
51         Route::put('computer/{id}/reset-steps', 'resetSteps');
52
53         Route::put('computer/{id}/responsible', 'changeResponsible');
54     });
55
56     Route::middleware('admin')->group(function () {
57         Route::controller(UserController::class)->group(function () {
58             Route::get('users', 'index');
59             Route::put('user/{id}/switch-user-license', '
                switchUserLicense');
60         });
61     });
62 });

```

Fonte: Produção do próprio autor.

Código-fonte 3.5 – Trecho do arquivo do *middleware* que garante que o usuário está ativo.

```

18 public function handle(Request $request, Closure $next)
19 {
20     if (Auth::user()->license != 'active') {

```

```
21     return response()->json([
22         'message' => 'Usuario inativo.'
23     ], 403);
24 }
25 return $next($request);
26 }
```

Fonte: Produção do próprio autor.

Código-fonte 3.6 – Trecho do arquivo do *middleware* que garante que o usuário possui papel de administrador.

```
18 public function handle(Request $request, Closure $next)
19 {
20     if (Auth::user()->role != 'admin') {
21         return response()->json([
22             'message' => 'Usuario nao possui as permissoes
                necessarias.'
23         ], 403);
24     }
25     return $next($request);
26 }
```

Fonte: Produção do próprio autor.

3.2.5 Cache

O *cache* opera como uma estrutura de chave-valor, armazenando temporariamente informações frequentemente acessadas para otimizar o desempenho. Contudo, devido à natureza dinâmica das informações em vários módulos da aplicação, a eficácia do *cache* pode ser comprometida, uma vez que as atualizações frequentes demandam uma gestão constante. Desse modo, o benefício em não consultar o banco de dados em certos momentos seria perdido ao ter que atualizar o *cache* de maneira extensiva, como é o caso da lista de computadores. Assim, como escolha de projeto, adotou-se uma abordagem mais estratégica, aplicando o *cache* em requisições específicas, por exemplo quando o usuário autenticado faz uma requisição para obter seus dados, já que não são tão frequentemente alterados.

3.2.6 Testes

3.2.6.1 Fábricas e Semeadores

Para realizar os testes das funcionalidades implementadas, se faz importante popular o banco de dados. Para isso, serão apresentados como exemplo o arquivo de fábrica e o

semeador de usuários. Com a finalidade de popular o banco, é utilizada a biblioteca *faker* integrada à Laravel, que provê métodos para gerar dados aleatórios.

A partir do Código-fonte 3.7, é possível observar a utilização do *faker* para criar dados no formato de nome, *e-mail*, número de telefone, data, sequência de caracteres e de inteiros aleatórios, além da criação de um *hash* de senha gerado a partir do texto “*password*”.

Código-fonte 3.7 – Trecho do arquivo de fábrica de usuários.

```
19 public function definition()
20 {
21     return [
22         'name' => $this->faker->name(),
23         'email' => $this->faker->unique()->safeEmail(),
24         'institutional_id' => $this->faker->unique()->numerify('
                #####'),
25         'telephone' => $this->faker->phoneNumber(),
26         'email_verified_at' => now(),
27         'password' => Hash::make('password'),
28         'remember_token' => Str::random(10)
29     ];
30 }
```

Fonte: Produção do próprio autor.

O alimentador de usuários no Código fonte 3.8 é o responsável por trazer as definições da fábrica e inserir os dados diretamente na tabela do banco. Neste exemplo, são criados 50 usuários com as definições padrões, 10 usuários não verificados, 2 administradores e 30 usuários ativos.

Código-fonte 3.8 – Trecho de código do arquivo alimentador de usuários.

```
9 class UserSeeder extends Seeder
10 {
11     public function run()
12     {
13         User::factory()
14             ->count(50)
15             ->unverified(10)
16             ->admin(2)
17             ->active(30)
18             ->create();
19     }
20 }
```

Fonte: Produção do próprio autor.

Há um arquivo responsável por centralizar todos os alimentadores e popular o banco denominado “*DatabaseSeeder.php*”. Nele, além da declaração de todos os alimentadores de interesse, também é possível inserir registros a parte. Por padrão, são criados dois usuários manualmente: um com o papel de administrador e outro com o papel de membro do projeto.

Com o *container* do lado do servidor em execução, é executado via CLI o comando “*sail artisan migrate:fresh --seed*”. Através deste comando, todas as tabelas do banco de dados são removidas e geradas novamente. O parâmetro “*seed*” indica que as novas tabelas geradas devem ser populadas utilizando como padrão o arquivo “*DatabaseSeeder.php*”. O banco de dados PostgreSQL é executado por padrão na porta 5632 e pode ser acessado para comprovar que o banco de dados foi populado. Na Figura 11 é realizada uma consulta simples que recupera apenas a coluna “*name*” da tabela “*users*”. Nesta imagem se comprova que o banco de dados foi populado efetivamente, iniciando pelos dois usuários criados manualmente no arquivo “*DatabaseSeeder.php*”, seguidos pelos usuários criados aleatoriamente utilizando o *faker*.

Figura 11 – Consulta ao banco de dados que recupera os nomes dos usuários.

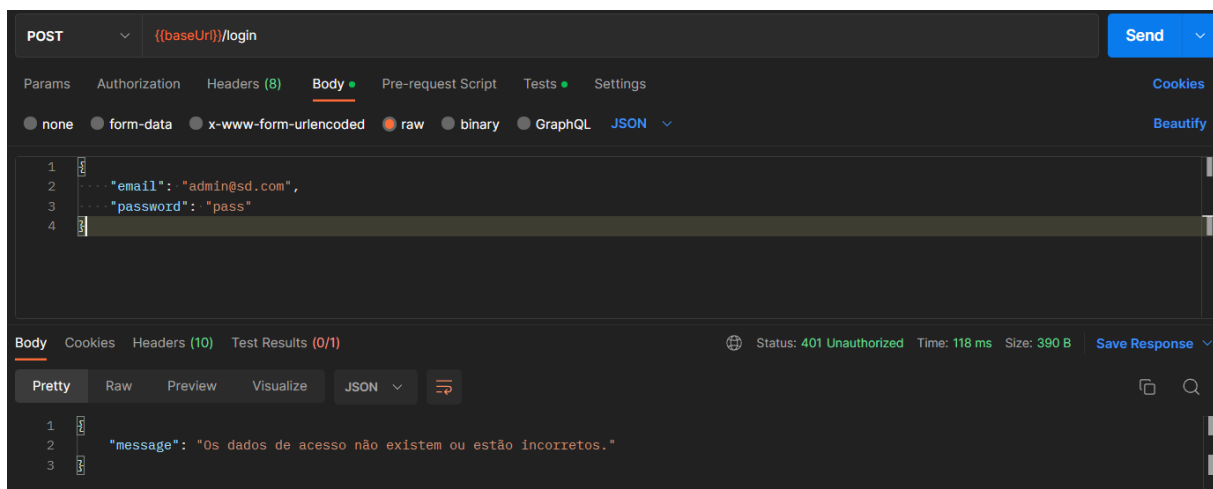
```
sd_backend=# select name from users;
           name
-----
Administrador
Responsável
Maiara Thalia Gusmão Sobrinho
Dr. Lara Leon Camacho
Clara Nayara Cordeiro
Constância Santana Bittencourt
Sra. Eva Cruz
Luan Fabrício Bittencourt Filho
Sra. Hosana Marina Salazar
```

Fonte: Produção do próprio autor.

3.2.6.2 Testes Unitários

Nesta seção é proposta a realização de uma bateria de testes unitários para validar a implementação do lado do servidor. Para isso, é utilizado o *Postman*, uma plataforma cliente capaz de realizar requisições HTTP para testes. Em um primeiro momento, são testadas as rotas de autenticação. Na Figura 12, no corpo da requisição é enviado um JSON com as credenciais incorretas do usuário. A resposta traz o *status* 401 e uma mensagem que indica que os dados de acesso são inválidos, como esperado.

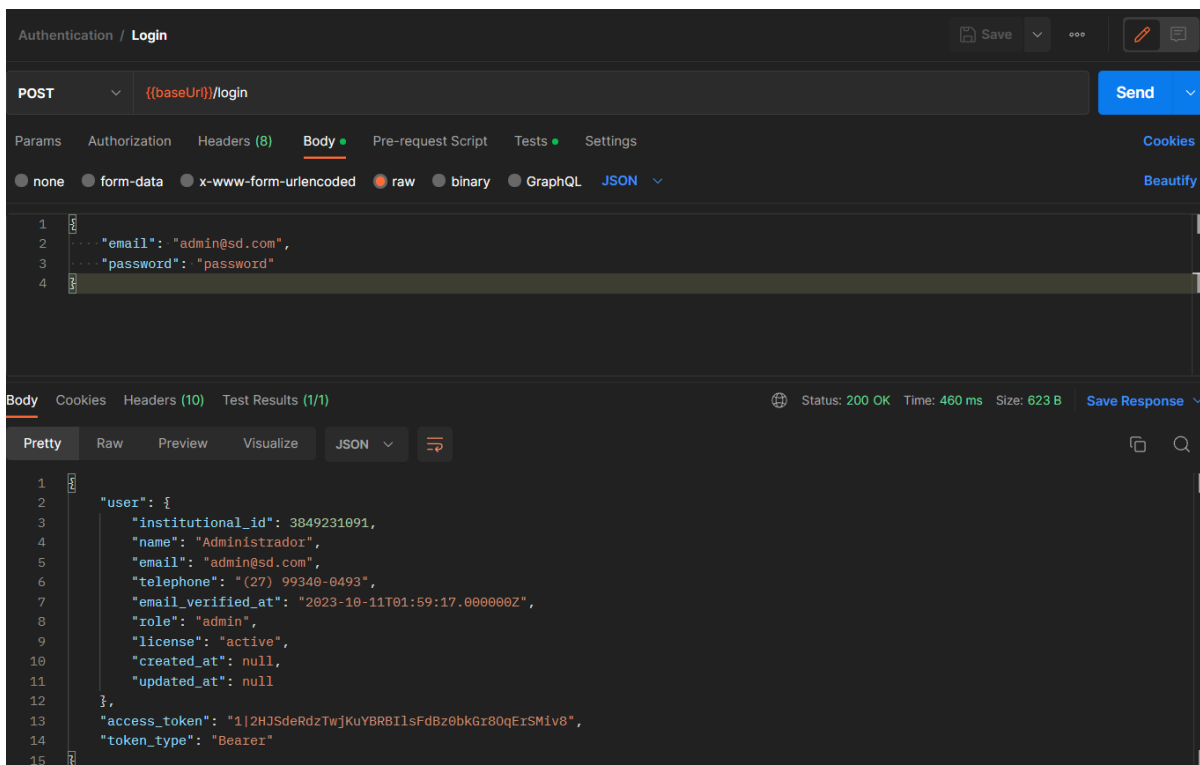
Figura 12 – Teste de autenticação utilizando credenciais inválidas.



Fonte: Produção do próprio autor.

A Figura 13, por sua vez, traz a requisição de autenticação utilizando credenciais válidas. Na corpo da resposta, o JSON traz consigo informações do usuário que podem ser utilizadas na aplicação cliente e um *token* de acesso. Este token, do tipo *Bearer*, deve ser enviado em todas as requisições subsequentes por meio do cabeçalho HTTP *Authorization*.

Figura 13 – Teste de autenticação utilizando credenciais válidas.

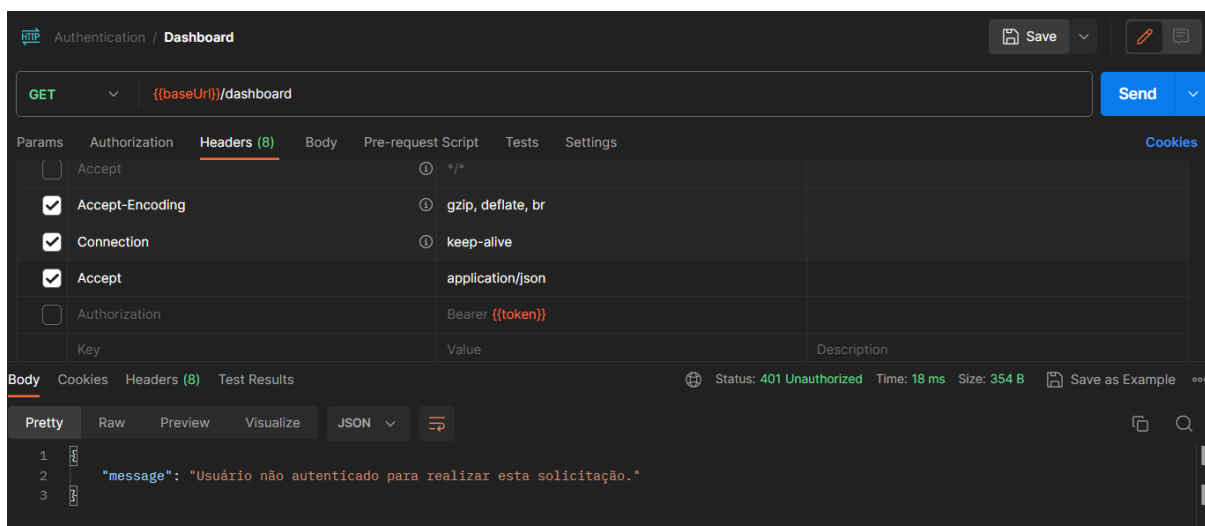


Fonte: Produção do próprio autor.

Em sequência, na Figura 14 é testada a aplicação do *middleware* de autenticação. Como

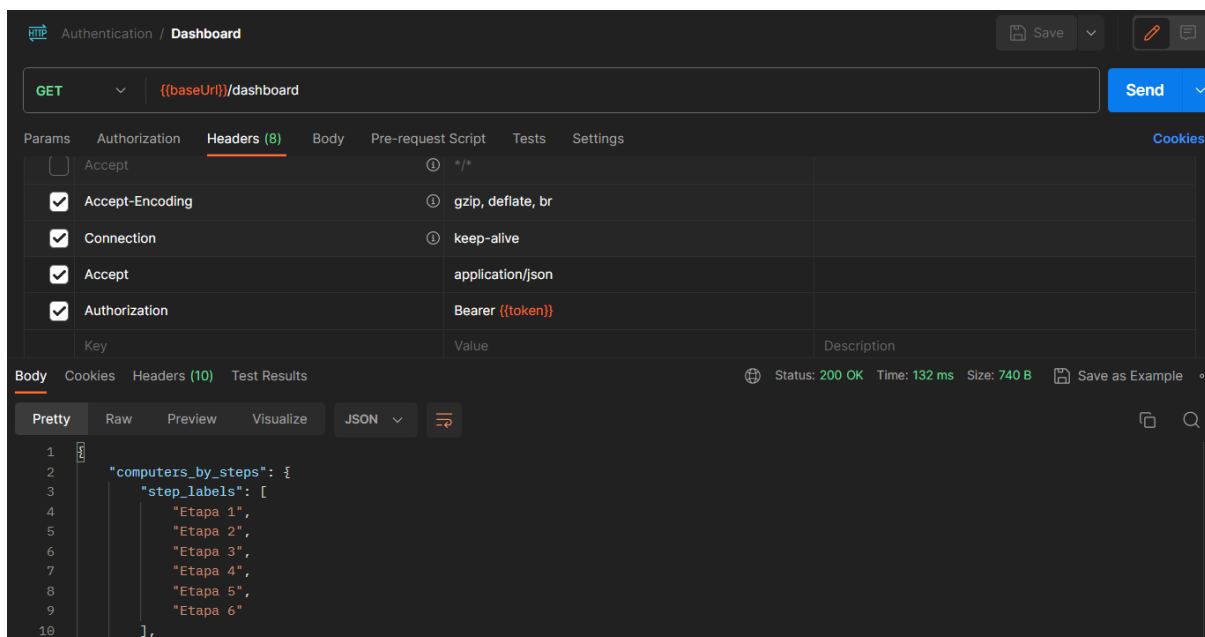
observado, é feita uma requisição GET para a rota do painel, que está inclusa no *middleware* de autenticação. Neste exemplo, nenhum cabeçalho de autenticação é adicionado à requisição, o que pode ser observado com a caixa de seleção do cabeçalho *Authorization* desmarcada na aba superior. Dessa forma, o *middleware* atua antes mesmo de chegar ao controlador, retornando uma resposta com *status* 401. Ao inserir o cabeçalho na Figura 15, a rota retorna os dados tratados pelo controlador com um *status* 200, conforme esperado.

Figura 14 – Teste de requisição para uma rota protegida sem a presença do cabeçalho de autorização.



Fonte: Produção do próprio autor.

Figura 15 – Teste de requisição para uma rota protegida com o cabeçalho de autorização.

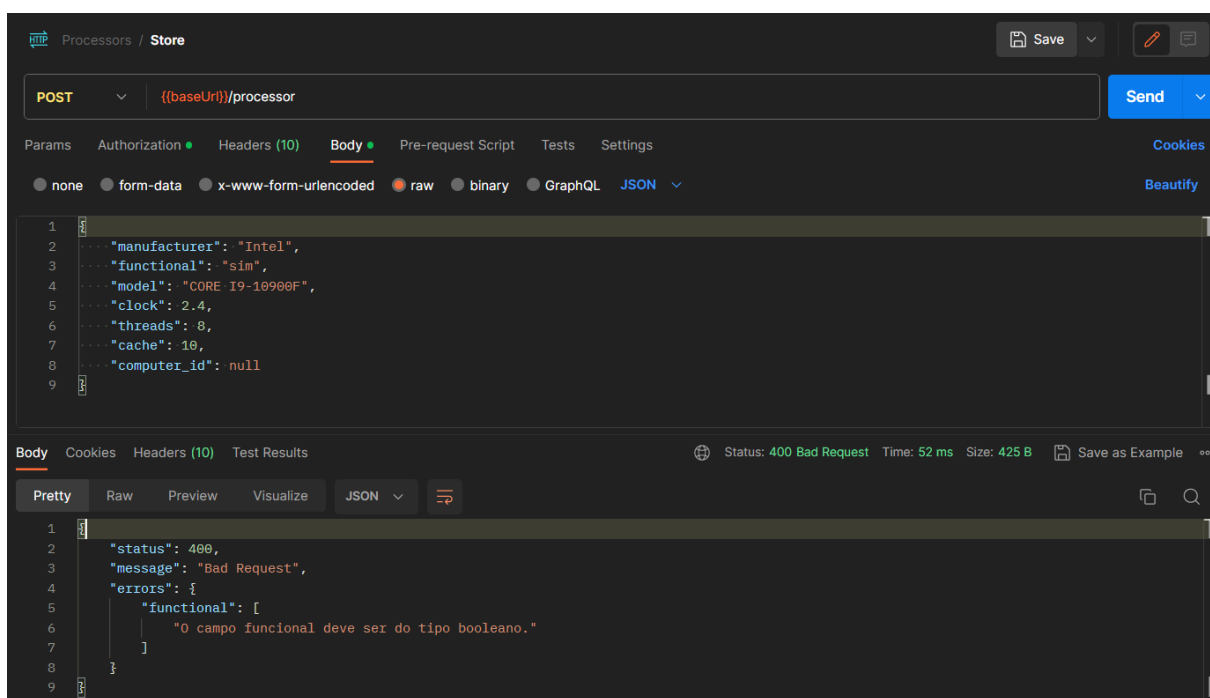


Fonte: Produção do próprio autor.

Já autenticado, ainda é possível testar uma rota de criação de recurso, utilizando como base o modelo do processador, tratado na Seção 3.2.2. Na Figura 16, é testada a validação

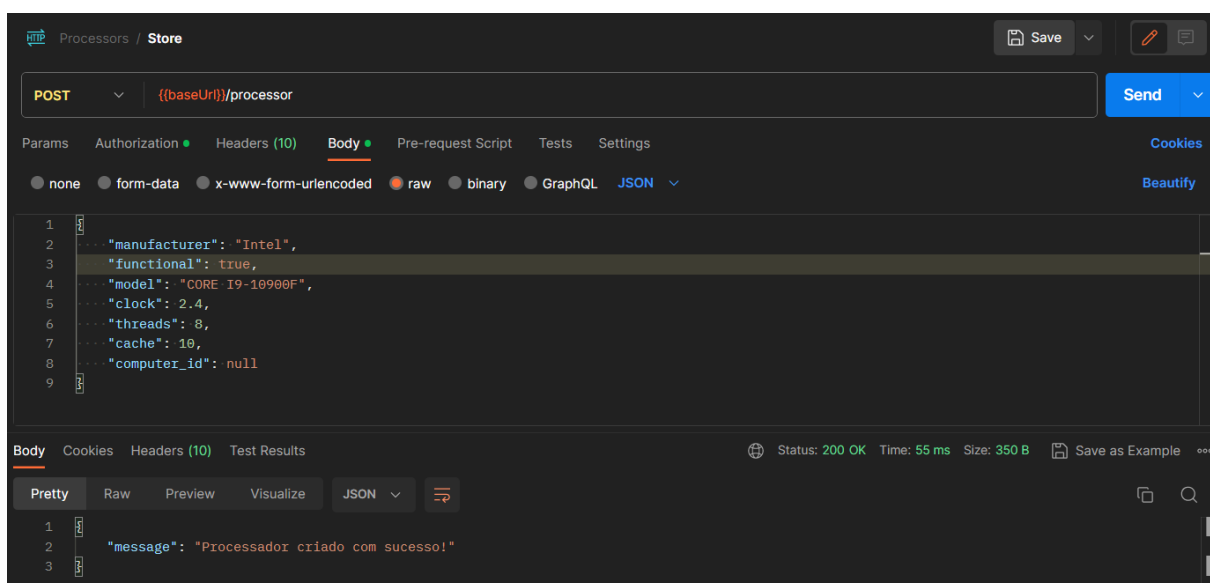
de atributos para criação de um processador. O valor do campo “*functional*” deve ser do tipo *booleano*: dessa forma, ao tentar realizar uma requisição passando como valor do mesmo campo a *string* “sim”, o servidor retorna um *status* 400 de má requisição. Na Figura 17, é utilizado o valor do tipo correto, recebendo uma resposta válida confirmando a criação do processador.

Figura 16 – Teste com falha de validação da requisição de criação de um processador.



Fonte: Produção do próprio autor.

Figura 17 – Teste de sucesso de validação da requisição de criação de um processador.



Fonte: Produção do próprio autor.

Todos os testes de cada funcionalidade implementada foram avaliados unitariamente a fim de garantir maior eficiência e qualidade de código durante o desenvolvimento. Após os testes unitários, essas funcionalidades foram integradas e testadas como conjunto, de maneira progressiva, até o resultado esperado. Nesta seção foram demonstrados testes de autenticação, *middleware*, validação de atributos, recuperação e criação de recursos.

3.3 Lado do Cliente

O desenvolvimento do lado do cliente envolve a criação da interface de usuário responsiva, a lógica de negócio da aplicação e a integração com a API desenvolvida do lado do servidor. A escolha do Vue.js possibilitou que todo o desenvolvimento fosse realizado através de componentes, o que torna o código mais legível e otimizado. O lado do cliente seguiu a mesma ideia de modularização proposta na modelagem do sistema, o que corresponde às páginas implementadas. O código completo está disponível para consulta no *github* (SOUSA, 2022b).

A responsividade da interface de usuário foi garantida com a utilização das classes de apoio do Quasar. Desde a fase inicial do desenvolvimento do *software*, optou-se por seguir a metodologia *Mobile First*. Essa escolha estratégica possibilitou uma disposição mais eficiente e fluida das informações em dispositivos móveis, simplificando a transição para ambientes *desktop*.

No *layout* da aplicação, existem alguns componentes específicos para *desktop* e outros para dispositivos móveis. No trecho de código apresentado no Código-fonte 3.9, existem componentes de cabeçalhos e rodapé para ambas as versões. A diretiva “*v-if*” na linha 5 somente monta o componente de cabeçalho *desktop* caso o tamanho da tela do dispositivo seja maior que “600px”. Caso contrário, o componente montado é o feito para dispositivos móveis. O mesmo acontece para o rodapé. As páginas são renderizadas dentro do componente “*q-page-container*”, representadas pela *tag* “*router-view*”.

Código-fonte 3.9 – *Layout* principal da aplicação.

```
1 <template>
2   <q-layout view="lHh LpR lFf">
3
4     <desktop-header
5       v-if="$q.screen.gt.sm"
6       :username="username"
7       :profile-links="profileLinks"
8       :profile-pic="profilePic"
9     />
```

```

10
11     <mobile-header
12         v-else
13         :profile-links="profileLinks "
14     />
15
16     <mobile-footer
17         v-if="$q.screen.lt.md "
18         :links="dashboardLinks "
19         :breakpoint="550 "
20     />
21
22     <desktop-drawer v-if="$q.screen.gt.sm " :links="dashboardLinks " /
23         >
24     <q-page-container>
25         <router-view />
26     </q-page-container>
27 </q-layout>
28 </template>

```

Fonte: Produção do próprio autor.

Abordando um novo ponto, a página que implementa a manutenção de computadores é interessante pois traz consigo uma renderização dinâmica de componentes de acordo com uma variável de estado. Nesse caso, cada etapa de manutenção é um componente dentro da aplicação. Além disso, nela é possível ver tanto a lógica de negócio da aplicação quanto a integração da API, via *axios*.

No trecho de código do Código-fonte 3.10, é possível observar que há um componente genérico declarado dentro de um formulário, que ao ser respondido chama o método “*handleSubmit*”. Esse componente genérico é definido pela propriedade computada “*currentComponent*”, que traz consigo sempre o número da etapa atual do computador. Em adição, o componente é atualizado sempre que o método “*showComputer*” é executado.

Código-fonte 3.10 – Trecho do código HTML da página de manutenção.

```

53 <q-step
54     v-for="step in stepOptions "
55     :key="step.value "
56     :name="step.value "
57     :title="step.label "
58     :icon="maintenanceIcons [step.value] "
59     :done-icon="maintenanceIcons [step.value] "
60     :active-icon="maintenanceIcons [step.value] "
61     :done="computer.current_step > step.value ||
        computer.current_step == 6"

```

```
62     :header-nav="computer.current_step >= step.value"
63     done-color="secondary"
64 >
65     <q-form ref="stepForm" @submit="handleSubmit">
66         <component @refresh="showComputer" v-model="computer" :is="
           currentComponent" />
67     </q-form>
68 </q-step>
```

Fonte: Produção do próprio autor.

Dessa forma, pode-se analisar ambos os métodos, no trecho de Código-fonte 3.11. Cada etapa dentro do formulário ao ser enviada, é validada e então chama pelo método “*handleSubmit*”. O método primeiro verifica se a etapa atual é a 2: caso seja, ele verifica se há pelo menos um componente funcional dentro do computador sob análise antes de prosseguir.

De acordo com a etapa em que o computador está, uma requisição do tipo POST assíncrona é realizada para a API através do *axios*, encapsulada no bloco *try-catch* para tratamento de exceções. Por fim, é chamado o método “*showComputer*”, que realiza uma requisição GET para a API que recupera os dados daquele computador, agora com o número da etapa atual incrementado. Dessa forma, a variável computada “*currentComponent*” é atualizada, o que atualiza também o componente a ser renderizado para a etapa seguinte.

Código-fonte 3.11 – Trecho do código *JavaScript* da página de manutenção.

```
189 async handleSubmit () {
190     if (this.computer.current_step == 2) {
191         const functionalMotherboard = this.computer.motherboard?.
           functional
192         const functionalProcessor = this.computer.processor?.functional
193         const functionalPowerSupply = this.computer.power_supply?.
           functional
194         const functionalRamMemory = this.computer.ram_memories?.filter(
           itm => itm.functional == true).length > 0
195         const functionalStorageDevice = this.computer.storage_devices?.
           filter(itm => itm.functional == true).length > 0
196         const functionalGpu = this.computer.gpus?.filter(itm =>
           itm.functional == true).length > 0
197         const functionalMonitor = this.computer.monitors?.filter(itm =>
           itm.functional == true).length > 0
198
199         if (!(functionalMotherboard && functionalProcessor &&
           functionalPowerSupply &&
200             functionalRamMemory && functionalStorageDevice &&
           functionalGpu && functionalMonitor)
201     ) {
```

```
202
203     return this.$q.notify({
204         message: 'O computador nao possui os componentes funcionais
                minimos para a proxima etapa.',
205         type: 'warning'
206     })
207 }
208 }
209
210 try {
211     this.$q.loading.show()
212
213     const { data } = await this.$axios.put('computer/' +
        this.computer.id + '/' + this.currentApiPath, {
214         ...this.computer
215     })
216
217     this.$q.notify({
218         message: data.message,
219         type: 'positive'
220     })
221
222     this.showComputer()
223 } finally {
224     this.$q.loading.hide()
225 }
226 }
```

Fonte: Produção do próprio autor.

Como parte do projeto, foi implementado um leitor e gerador de *QR Code*. O desenvolvimento teve como base a biblioteca *vue-qrcode-reader*¹ para realizar a leitura. Utilizando a câmera do dispositivo através do navegador, um evento é disparado sempre que é detectado um *QR Code*, decodificando o resultado no cliente. A geração do código, por sua vez, foi dada utilizando a biblioteca *qrious*².

As demais particularidades do sistema, abrangendo aspectos como roteamento, proteção e validação de dados, espelham de forma similar as implementadas no lado do servidor.

¹ <<https://www.npmjs.com/package/vue-qrcode-reader>>

² <<https://github.com/neocotic/qrious>>

4 RESULTADOS

Neste capítulo serão apresentados os resultados obtidos através da integração do cliente com o lado do servidor. Para isso, é feita uma abordagem de apresentação por meio das interfaces de usuário do lado do cliente que exibem dados provenientes da API.

4.1 Recursos Computacionais

Com relação aos recursos computacionais para o ambiente de desenvolvimento, foi utilizado o WSL¹, um subsistema Linux compatível com o ambiente da Microsoft, no Windows. Além disso, o Docker² foi empregado para a virtualização dos ambientes em contêineres, permitindo maior compatibilidade e facilidade de instalação em qualquer máquina. Adicionalmente, a plataforma do Github³ foi adotada como método de hospedagem e controle de versionamento de código-fonte. Na etapa de desenvolvimento da API e do cliente, foram utilizadas as *frameworks* Quasar⁴ e Laravel⁵, com o uso da aplicação Postman⁶ para realização de testes. Por fim, para o banco de dados foi utilizado o SGBD PostgreSQL⁷.

Todo o trabalho apresentado foi realizado em *localhost*, com a aplicação cliente escutando a porta 8081 e a aplicação do servidor escutando a porta 8000.

4.2 Módulos da Aplicação

Para fim de apresentação dos resultados da aplicação, os resultados são separados em módulos lógicos, que correspondem aos anteriormente discutidos na modelagem do sistema.

¹ <<https://docs.microsoft.com/pt-br/windows/wsl/>>

² <<https://www.docker.com/>>

³ <<https://github.com/>>

⁴ <<https://quasar.dev/>>

⁵ <<https://laravel.com/>>

⁶ <<https://www.postman.com/>>

⁷ <<https://www.postgresql.org/>>

4.2.1 Autenticação

A autenticação é a base do sistema, tratada tanto no lado do servidor quanto no lado do cliente. É a primeira página que o usuário se depara ao inicializar a aplicação. Na Figura 18 é apresentada a tela de *login* sob a ótica de um dispositivo *desktop*. Nesta tela, são solicitadas as credenciais do usuário para autenticação: *e-mail* e senha. Ambos os campos são obrigatórios e possuem validação simples para verificar se o campo de e-mail possui uma formatação válida. Abaixo do botão de *login* é possível também abrir a tela de cadastro, que também conta com validações de campo.

Figura 18 – Tela principal de autenticação.



Fonte: Produção do próprio autor.

Ao realizar o cadastro, o usuário recebe um *e-mail* de verificação, como observado na Figura 19. Este *e-mail*, no âmbito deste projeto, foi interceptado pelo *Mailtrap*⁸, um serviço online projetado para facilitar o teste de envio de *e-mails* em ambientes de desenvolvimento e teste de software. Ele fornece um ambiente seguro e isolado para testar o envio de *e-mails* sem que mensagens reais sejam entregues aos destinatários finais. Ao efetuar o clique no botão ou no *link* exibido no rodapé do *e-mail*, o usuário é redirecionado para uma página da aplicação que efetua uma requisição HTTP para a rota de verificação de *e-mail* no lado do servidor.

⁸ <<https://mailtrap.io/>>

Figura 19 – E-mail de verificação recebido pelo usuário.

Olá, José Henrique

Estamos muito felizes de tê-lo conosco em nosso projeto.

O Solidariedade Digital está cada vez mais somando para que os alunos da UFES tenham acesso à condições de qualidade de inclusão e conhecimento.

Por favor, clique no link abaixo para confirmar o seu e-mail e começar a utilizar o nosso software.

Confirmar e-mail

Atenciosamente,
Solidariedade Digital

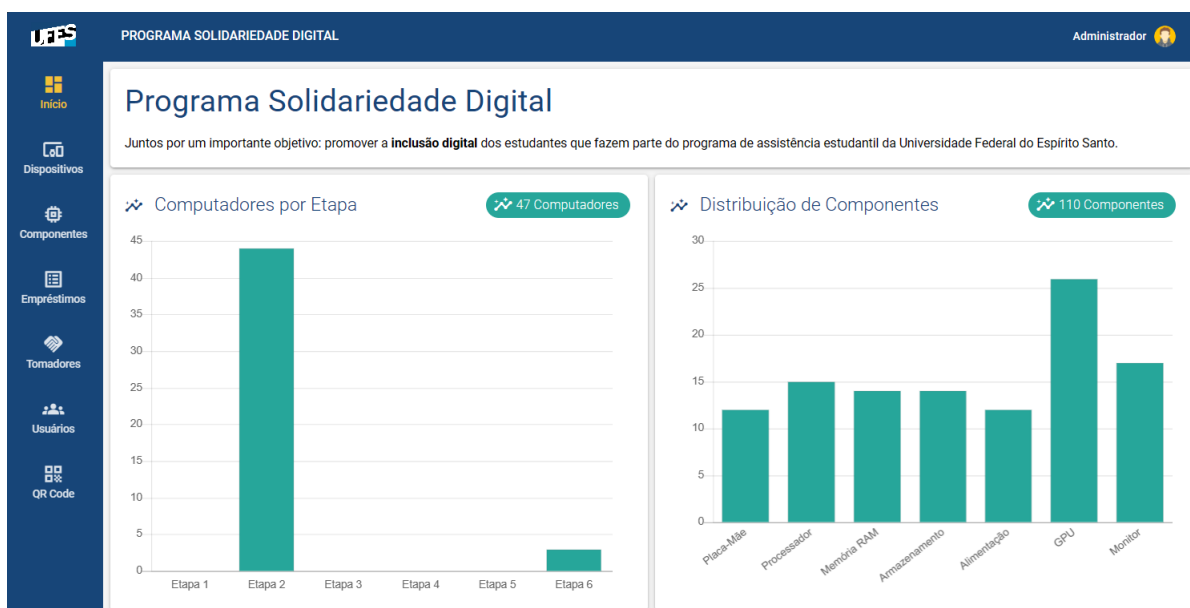
Se está tendo algum problema clicando no botão acima, copie e cole a URL abaixo no seu navegador: <http://localhost:8081/verificar-email/68/a89aafc9349accaf0a9918d79c0c75ce97179027>

Fonte: Produção do próprio autor.

4.2.2 Painel Inicial

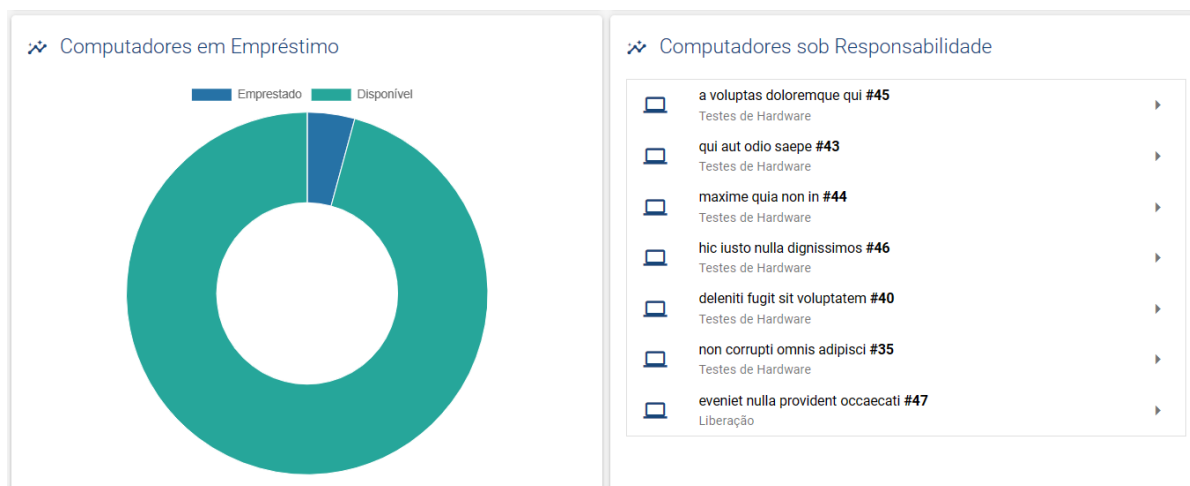
O painel inicial possui uma apresentação mais visual com o usuário, utilizando de artefatos como gráficos e cartões para exibir os indicadores do projeto. Na Figura 20 são exibidos dois gráficos de barras: computadores por etapa e distribuição de componentes. Nota-se ainda que em cada um dos gráficos, no canto superior direito, possui um indicador que totaliza o número de computadores e o número de componentes, em cada um dos casos. Ainda rolando a página para baixo, como na Figura 21, também são exibidos os gráficos de computadores em empréstimo, utilizando um gráfico de setores: este gráfico indica uma proporção entre computadores que estão em outras etapas e o quantitativo de computadores em empréstimo. O cartão do lado direito exibe os computadores sob a responsabilidade do usuário que está autenticado atualmente: além de facilitar a sua visualização, também proporciona um rápido acesso à página de manutenção daquele computador, ao clicar em uma das linhas exibidas.

Figura 20 – Tela de visualização de indicadores.



Fonte: Produção do próprio autor.

Figura 21 – Gráfico de empréstimos e cartão de computadores sob responsabilidade.



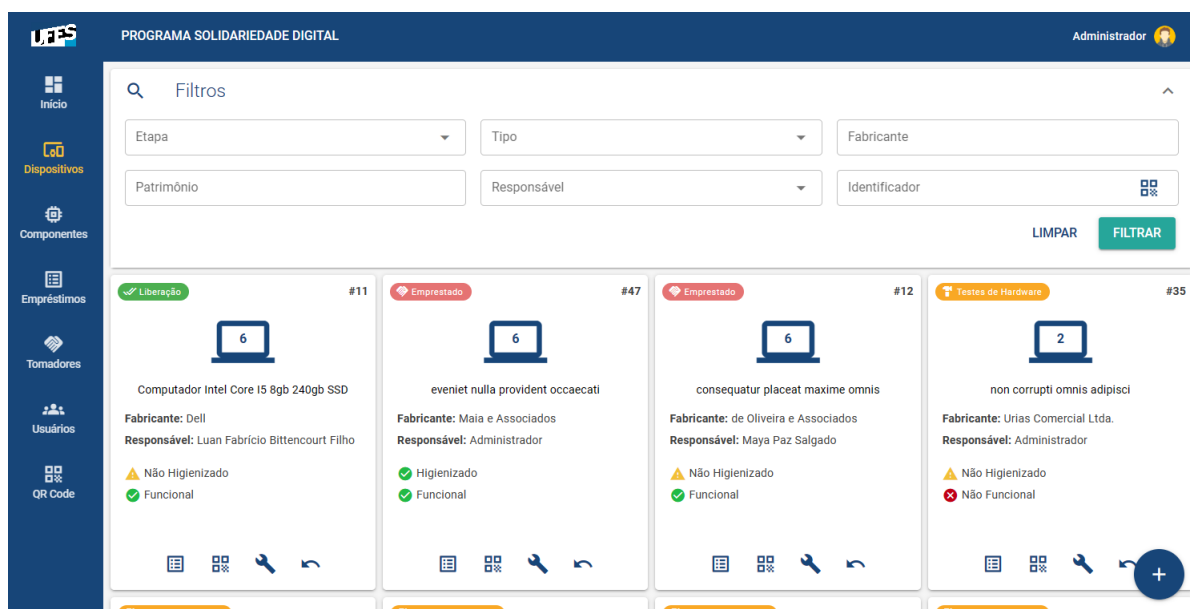
Fonte: Produção do próprio autor.

4.2.3 Computadores

O módulo dos computadores é o centro do modelo de negócio do sistema. Na Figura 22 é apresentada a tela de listagem de computadores. Acima da tela há uma série de filtros que podem ser realizados, inclusive um filtro por identificador. O filtro por identificador permite que o usuário abra a câmera (caso em dispositivo móvel) e escaneie um *QR Code*. Ao ser escaneado, o campo é preenchido automaticamente. Logo abaixo, a lista de computadores traz informações breves como etapa atual e descrição. Os botões abaixo

de cada cartão possuem as funcionalidades, respectivamente da esquerda para a direita: visualizar empréstimos, gerar *QR Code*, visualizar processo de manutenção e retornar computador para triagem. O botão flutuante no canto inferior direito permite abrir um diálogo para registrar um novo computador.

Figura 22 – Tela de listagem de computadores.

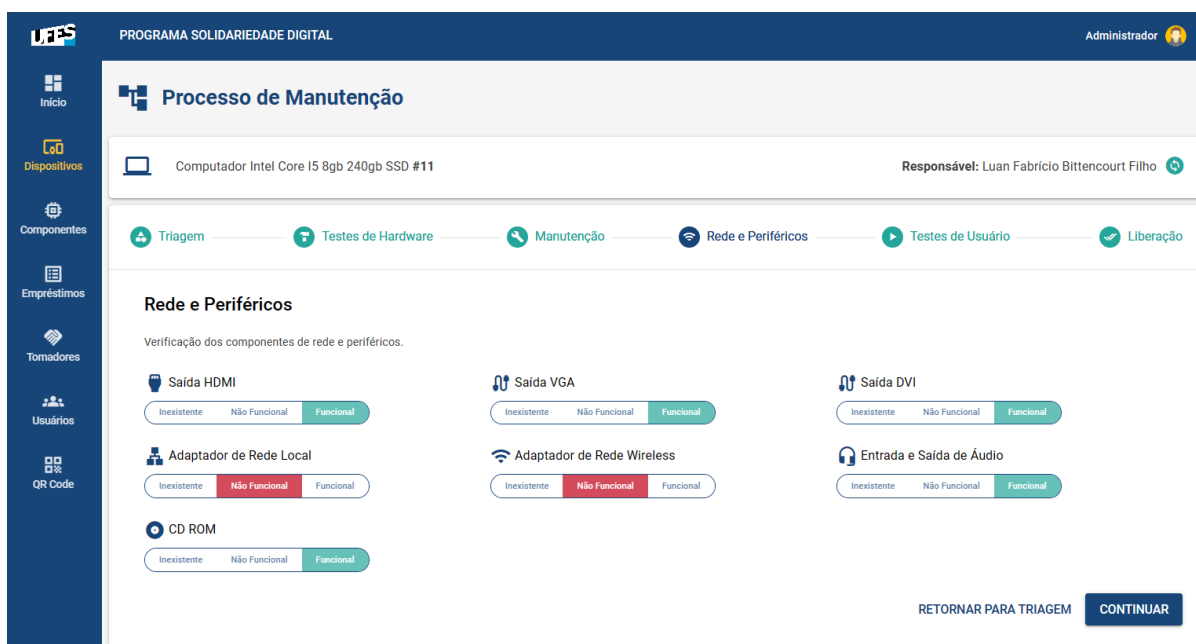


Fonte: Produção do próprio autor.

Ao acessar a manutenção de determinado computador, é possível verificar seu processo de manutenção, como segue na Figura 23. Este computador específico se encontra na etapa de Rede e Periféricos.

No primeiro cartão abaixo do título “Processo de Manutenção” é possível observar a descrição do computador, bem como o atual responsável por aquela etapa de manutenção, no canto direito, podendo ser alterado ao clicar no botão verde ao lado do nome. Abaixo do cartão há um passo a passo com a indicação das fases de manutenção, exibindo a fase atual. No caso da etapa de rede e periféricos, o usuário seleciona para cada um dos periféricos o estado em que se encontra naquele computador. Por fim, o último cartão é colapsável e traz os comentários feitos pelos membros sobre aquele computador.

Figura 23 – Tela do processo de manutenção na etapa de Rede e Periféricos.



Fonte: Produção do próprio autor.

4.2.4 Componentes

A página de componentes, na Figura 24, funciona como um redirecionador para a página de cada componente específico. Por padrão, esta página é inicializada nos processadores. O menu superior permite navegar entre os componentes, que traz uma listagem paginada com filtros e informações acerca de cada componente.

Na página de cada componente é possível realizar as operações de editar, visualizar o histórico de transferências e os comentários. Quando o componente está emprestado, como no caso do processador de *id* 1, a edição é bloqueada. O botão flutuante, assim como na página de computadores, cria um novo registro de componente.

Figura 24 – Tela de listagem de componentes.

Identificador	Fabricante	Modelo	Funcional	Emprestado	Computador	Atualizado em	Ações
#5	Leal-Zamana	aperiam quo at qui	Funcional	Liberado	#11	25/10/2023 - 09:14	Editar, Histórico, Comentar
#4	Perez-Gonçalves	ad fugit odio fugit	Não Funcional	Liberado	-	24/10/2023 - 23:19	Editar, Histórico, Comentar
#1	Vega-Medina	sed excepturi laboriosam et	Funcional	Emprestado	#47	24/10/2023 - 23:16	Editar, Histórico, Comentar
#7	Amaral Comercial Ltda.	provident molestiae assum...	Não Funcional	Liberado	-	10/10/2023 - 22:59	Editar, Histórico, Comentar
#8	Galhardo e Espinoza	sint corporis illo ipsum	Funcional	Liberado	-	10/10/2023 - 22:59	Editar, Histórico, Comentar
#9	Zamana e Cervantes	explicabo quis ut aperiam	Não Funcional	Liberado	-	10/10/2023 - 22:59	Editar, Histórico, Comentar
#2	Sanches-Carrara	vel ut libero nam	Funcional	Emprestado	#12	10/10/2023 - 22:59	Editar, Histórico, Comentar
#11	Carmona-Ávila	nostrum saepe ab qui	Funcional	Liberado	-	10/10/2023 - 22:59	Editar, Histórico, Comentar

Fonte: Produção do próprio autor.

4.2.5 Empréstimos e Tomadores de Empréstimo

Acerca dos empréstimos e tomadores de empréstimo, a exibição é semelhante. Os empréstimos são listados na Figura 25 de maneira paginada com filtros. Na página é possível visualizar a situação do empréstimo (atrasada/liberada/em andamento), comentários e acesso rápido às informações do computador/componente do empréstimo. O botão de devolução realiza a ação de baixa na aplicação, em que o responsável pelo empréstimo insere a data de devolução.

A tela de tomadores de empréstimos, na Figura 26, traz uma relação dos dados pessoais dos tomadores de empréstimos e uma propriedade computada que diz se o tomador possui ou não um empréstimo ativo no momento. O botão de empréstimos na última coluna da esquerda para a direita traz a listagem de todos os empréstimos, ativos ou não, da pessoa selecionada.

Em ambos os casos, também como nos outros módulos, o botão flutuante cria um novo registro.

Figura 25 – Tela de listagem de empréstimos.

Identificador	Tipo	ID do Item	Tomador de Empréstimo	Responsável	Data de Início	Data de Término	Data de Devolução	Emprestado	Ações
#8	Computador	#47	Sra. Flor Emily Galvão Jr.	Administrador	25/10/2023	25/10/2023	-	Atasado	Visualizar, Devolução, Comentários
#7	Computador	#12	Dr. Viviane Fontes	Administrador	25/10/2023	31/10/2023	-	Atasado	Visualizar, Devolução, Comentários
#3	Monitor	#17	Sra. Miriam Ester Saito	Clarice Valência Toledo	10/10/2023	10/10/2023	10/10/2023	Librado	Visualizar, Devolução, Comentários
#4	Computador	#13	Emiliano Pereira Jr.	Eloah Branco Serna Filho	10/10/2023	10/10/2023	10/10/2023	Librado	Visualizar, Devolução, Comentários
#1	Monitor	#15	Regiane Neves	Mathus Urias	10/10/2023	10/10/2023	10/10/2023	Librado	Visualizar, Devolução, Comentários
#6	Computador	#15	Wagner Corona Jr.	Dr. Guilherme Alves Marin F...	10/10/2023	10/10/2023	10/10/2023	Librado	Visualizar, Devolução, Comentários
#5	Computador	#14	Alan Vila Campos Neto	Natan Joaquim Pacheco Ne...	10/10/2023	10/10/2023	10/10/2023	Librado	Visualizar, Devolução, Comentários
#2	Monitor	#16	Sra. Rebeca Campos Corde...	Dr. Isabel Madeira Romero ...	10/10/2023	10/10/2023	10/10/2023	Librado	Visualizar, Devolução, Comentários

Fonte: Produção do próprio autor.

Figura 26 – Tela de listagem de tomadores de empréstimos.

Matrícula	Nome	Email	Telefone	Empréstimo Ativo	Ações
3396753126	Sra. Rebeca Campos Cordeiro	aguiar.paloma@example.org	(15) 3862-3676	Ativo	Empréstimos
6027627280	Sra. Miriam Ester Saito	flavio.amos@example.org	(18) 4067-1740	Ativo	Empréstimos
4183528290	Emiliano Pereira Jr.	malu.assuncao@example.com	(34) 94027-5262	Ativo	Empréstimos
2120070390	Alan Vila Campos Neto	mari.martines@example.net	(64) 90432-0356	Ativo	Empréstimos
7484587652	Wagner Corona Jr.	hmaldonado@example.com	(91) 4721-4360	Ativo	Empréstimos
3767140383	Ícaro Aaron Amaral Sobrinho	afonso01@example.net	(98) 94611-9508	Inativo	Empréstimos
2888615389	Jácomo Dante Santos Neto	ndelatorre@example.com	(94) 90067-0859	Inativo	Empréstimos
4684104128	Dr. Théo Antônio Grego Neto	rosana.uchoa@example.org	(33) 4919-5046	Inativo	Empréstimos
201819614	Dr. Edson Bonilha Corona	vcorreia@example.org	(65) 98219-4646	Inativo	Empréstimos

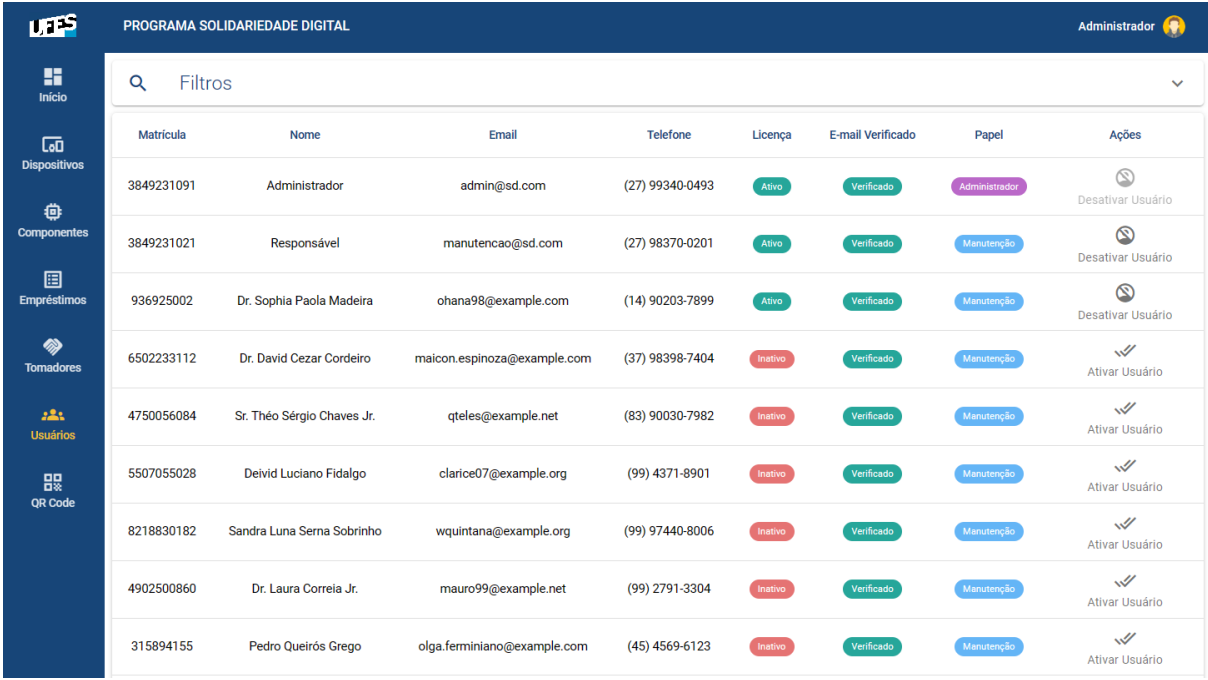
Fonte: Produção do próprio autor.

4.2.6 Usuários

O módulo de usuários é de único e exclusivo acesso aos usuários que possuem o papel de administrador no sistema. Através deste, é possível listar os dados dos usuários e realizar a gestão de acesso dos membros à aplicação, como exibido na Figura 27. Um usuário só pode

utilizar o sistema caso possua uma licença ativa e o *e-mail* verificado. O administrador, por meio dos botões de ativar/desativar usuário, pode liberar ou revogar o acesso de determinado membro a qualquer instante.

Figura 27 – Tela de listagem de usuários.

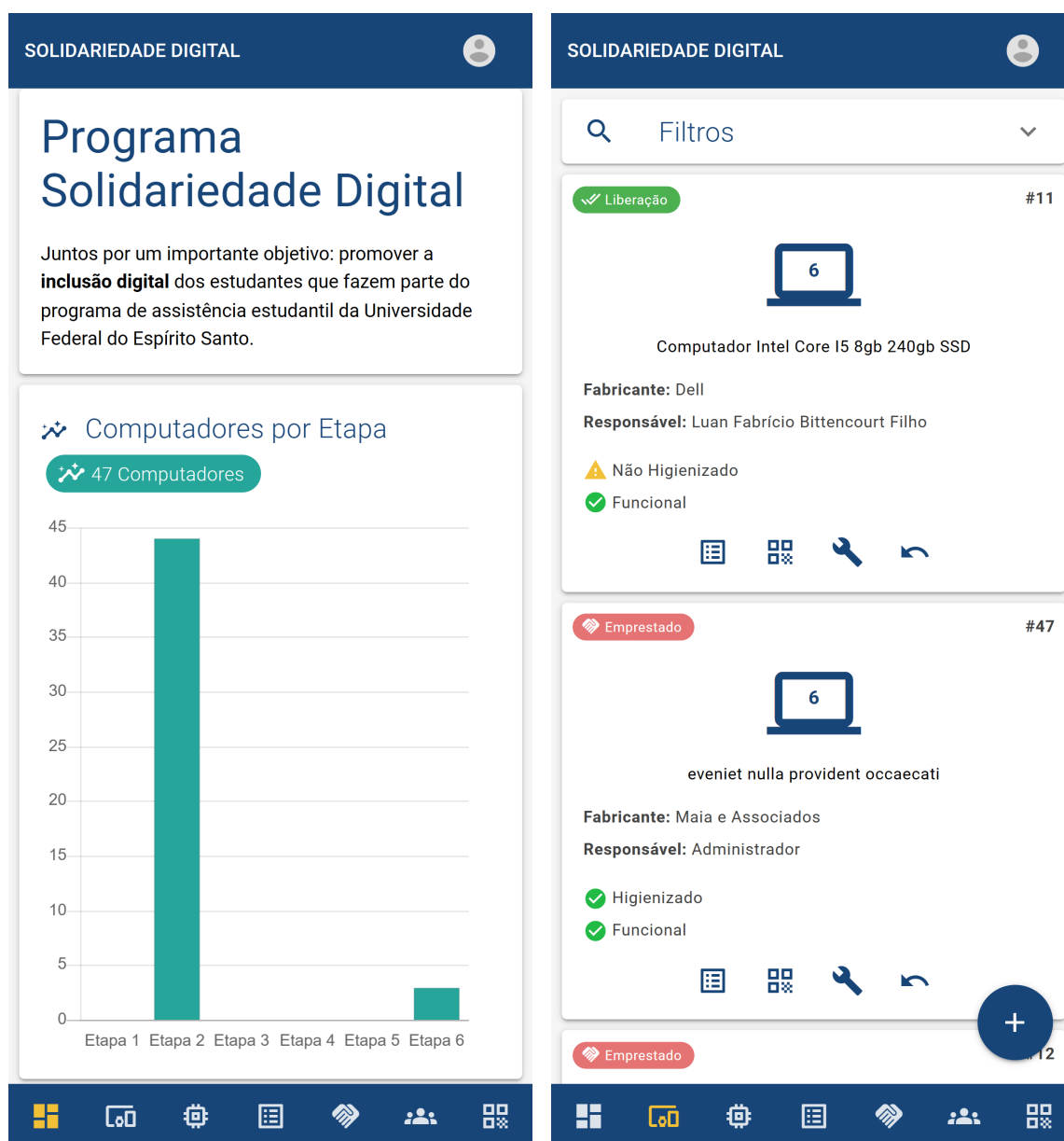


Matricula	Nome	Email	Telefone	Licença	E-mail Verificado	Papel	Ações
3849231091	Administrador	admin@sd.com	(27) 99340-0493	Ativo	Verificado	Administrador	Desativar Usuário
3849231021	Responsável	manutencao@sd.com	(27) 98370-0201	Ativo	Verificado	Manutenção	Desativar Usuário
936925002	Dr. Sophia Paola Madeira	ohana98@example.com	(14) 90203-7899	Ativo	Verificado	Manutenção	Desativar Usuário
6502233112	Dr. David Cezar Cordeiro	maicon.espinosa@example.com	(37) 98398-7404	Inativo	Verificado	Manutenção	Ativar Usuário
4750056084	Sr. Théo Sérgio Chaves Jr.	qteles@example.net	(83) 90030-7982	Inativo	Verificado	Manutenção	Ativar Usuário
5507055028	Deivid Luciano Fidalgo	clarice07@example.org	(99) 4371-8901	Inativo	Verificado	Manutenção	Ativar Usuário
8218830182	Sandra Luna Serna Sobrinho	wquintana@example.org	(99) 97440-8006	Inativo	Verificado	Manutenção	Ativar Usuário
4902500860	Dr. Laura Correia Jr.	mauro99@example.net	(99) 2791-3304	Inativo	Verificado	Manutenção	Ativar Usuário
315894155	Pedro Queirós Grego	olga.ferminiano@example.com	(45) 4569-6123	Inativo	Verificado	Manutenção	Ativar Usuário

Fonte: Produção do próprio autor.

4.3 Responsividade

Ao implementar classes CSS e componentes especializados, a aplicação alcançou um novo patamar de responsividade, estendendo-se de forma fluida e eficaz para dispositivos móveis. Através desses elementos estrategicamente aplicados, a experiência do usuário foi otimizada, garantindo uma visualização harmoniosa e funcionalidade consistente, independentemente do dispositivo utilizado. A Figura 28 apresentada é um testemunho visual dessa abordagem eficaz para a implementação da responsividade, solidificando a aplicação como uma escolha moderna e adaptável para os usuários. Ao construir a aplicação como uma PWA, esta pode ser instalada na tela inicial de um dispositivo móvel como um aplicativo nativo. As capturas de tela foram realizadas utilizando o recurso de emulação do navegador Edge, com as dimensões de tela de um Iphone 12 Pro (390px por 844px).



(a) Painel Inicial.

(b) Computadores.

Figura 28 – Tela do painel inicial e listagem de computadores em dispositivo móvel.

Fonte: Produção do próprio autor.

5 CONCLUSÃO E TRABALHOS FUTUROS

5.1 Conclusão

O propósito principal deste estudo consistiu no desenvolvimento de uma aplicação destinada à gestão dos processos de manutenção do projeto Solidariedade Digital. A arquitetura proposta tem como base os princípios do REST, incorporando conceitos modernos da *web*. Essa abordagem foi escolhida principalmente com ênfase no desacoplamento entre cliente e servidor, visando a escalabilidade do sistema.

Modelar o *software* utilizando UML proporcionou uma representação precisa do fluxo de manutenção, alinhando-se de maneira efetiva aos requisitos funcionais e não funcionais. Por sua vez, a aplicação de conceitos de experiência do usuário permitiu a criação de uma interface multiplataforma, fluida, responsiva e adaptável a uma variedade de dispositivos e tamanhos de tela. A redundância na validação de dados foi implementada para assegurar maior integridade ao sistema, garantindo que nenhuma restrição do banco de dados seja violada, reforçada por uma camada adicional de proteção. A incorporação de testes unitários ao longo do processo de desenvolvimento contribuiu para a criação de um sistema mais limpo e coeso.

As restrições fundamentais do estilo de arquitetura REST foram aplicadas neste projeto, evidenciando a conformidade com os princípios essenciais desse modelo:

- A desvinculação entre cliente e servidor foi efetivamente estabelecida, proporcionando uma separação clara e independência entre as duas entidades;
- As requisições foram implementadas como *stateless*, garantindo que cada solicitação seja autônoma, sem a necessidade de armazenamento de estado no servidor;
- A utilização de *cache* foi incorporada, demonstrando uma abordagem eficiente para otimização do desempenho em pontos específicos da aplicação;
- A interface uniforme foi rigorosamente seguida, proporcionando consistência nas interações por meio do protocolo HTTP;
- O sistema em camadas foi implementado de maneira coerente, fortalecendo a ideia de uma arquitetura flexível, com clara distinção e organização das diferentes camadas, como a API, o *cache* e o banco de dados;

- A restrição de código sob demanda, opcional, não foi aplicada por não ser considerada essencial dadas as características específicas da aplicação. Os principais benefícios dessa restrição não seriam justificados dada a introdução de complexidade adicional no projeto e riscos de segurança ao se executar códigos dinâmicos no cliente.

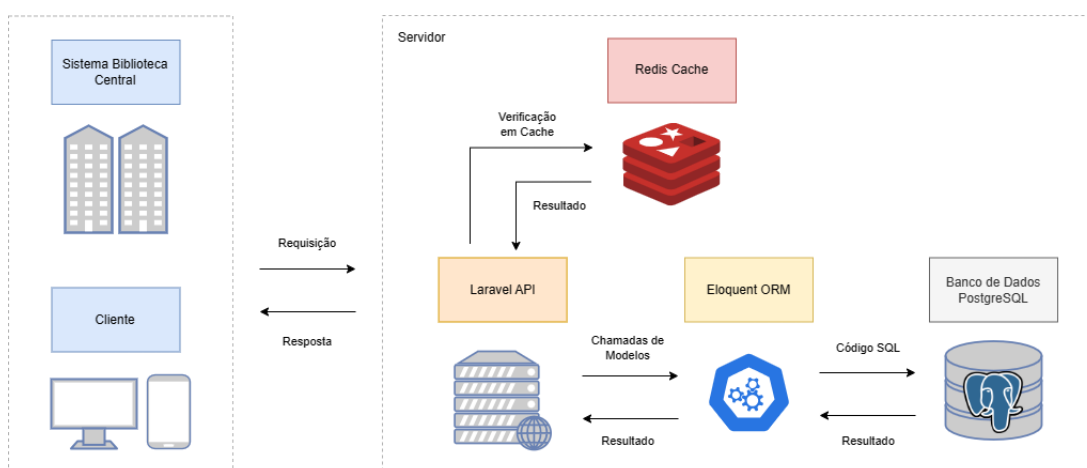
Este trabalho contribuiu significativamente para a otimização do processo de manutenção do projeto, promovendo uma estrutura mais organizada e produtiva para seus fluxos operacionais, se comparada com a sua organização via planilhas eletrônicas.

5.2 Trabalhos Futuros

Atualmente, está em discussão a implementação de um sistema de empréstimo de computadores aos alunos da universidade, semelhante ao procedimento de empréstimo de livros na biblioteca. Devido à natureza escalável e desacoplada do sistema, há a possibilidade de integrar eficientemente esses dois sistemas sem a necessidade de alterações significativas, como exposto na arquitetura da Figura 29.

Nesse cenário, realizando uma requisição simples à API desenvolvida neste trabalho, o sistema da biblioteca disponibilizaria para empréstimo apenas os computadores que houvessem completado a última etapa do fluxo de manutenção.

Figura 29 – Proposta de integração do sistema com a biblioteca.



Fonte: Produção do próprio autor.

Além disso, propõe-se a integração da autenticação do sistema em questão com o sistema de autenticação da universidade, que atualmente opera por meio do protocolo LDAP (*Lightweight Directory Access Protocol*). Essa iniciativa visa fortalecer a segurança e a

eficiência do sistema, alinhando-o aos padrões de autenticação estabelecidos pela instituição acadêmica.

REFERÊNCIAS

- AL-ATRAQCHI, O. M. A Proposed Model for Build a Secure Restful API to Connect between Server Side and Mobile Application Using Laravel Framework with Flutter Toolkits. Cihan University of Erbil, 2022. 28–35 p. Disponível em: <<https://journals.cihanuniversity.edu.iq/index.php/cuesj/article/view/575/274>>. Acesso em: 19 de dez. de 2023. Citado na página 16.
- BERNERS-LEE, T.; FIELDING, R. T.; MASINTER, L. M. Uniform Resource Identifier (URI): Generic Syntax. RFC Editor, 2005. RFC 3986. (Request for Comments, 3986). Disponível em: <<https://www.rfc-editor.org/info/rfc3986>>. Acesso em: 20 de jun. de 2022. Citado na página 21.
- CROCKER, D. H. STANDARD FOR THE FORMAT OF ARPA INTERNET TEXT MESSAGES. RFC Editor, 1982. RFC 822. (Request for Comments, 822). Disponível em: <<https://www.rfc-editor.org/info/rfc822>>. Acesso em: 25 de jun. de 2022. Citado na página 23.
- ELMASRI, R.; NAVATHE, S.; PINHEIRO, M. Sistemas de banco de dados. [S.l.]: Pearson Addison Wesley, 2009. Citado 2 vezes nas páginas 27 e 29.
- FIELDING, R. T. REST: Architectural Styles and the Design of Network-based Software Architectures. Tese (Doctoral dissertation) — University of California, Irvine, 2000. Disponível em: <<http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>>. Acesso em: 20 de jun. de 2022. Citado 6 vezes nas páginas 15, 18, 19, 20, 21 e 22.
- GANCHEV, M. Database vs spreadsheet: Full comparison. 2022. Disponível em: <<https://365datascience.com/tutorials/sql-tutorials/database-vs-spreadsheet/>>. Acesso em: 16 de jun. de 2022. Citado 2 vezes nas páginas 14 e 15.
- GOURLEY, D.; TOTTY, B.; SAYER, M.; AGGARWAL, A.; REDDY, S. HTTP: The Definitive Guide: The Definitive Guide. [S.l.]: O'Reilly Media, 2002. (Definitive Guides). Citado 2 vezes nas páginas 23 e 24.
- GUEDES, G. UML 2 - Uma Abordagem Prática - 3ª Edição. [S.l.]: Novatec Editora, 2018. ISBN 9788575226469. Citado 2 vezes nas páginas 28 e 29.
- LARAVEL. The PHP framework for web artisans. 2022. Disponível em: <<https://laravel.com/docs/9.x>>. Acesso em: 28 de jul. de 2022. Citado 2 vezes nas páginas 29 e 30.
- MUNIZ, A.; BARROS, A.; CONCEIÇÃO, B. da; CASTRO, C. de; QUERINO, D.; MAGNANI, M. Jornada API na prática: unindo conceitos e experiências do Brasil para acelerar negócios com a tecnologia. [S.l.]: Brasport, 2023. ISBN 9786588431979. Citado na página 18.
- NAGEL, W. Multiscreen UX Design: Developing for a Multitude of Devices. [S.l.]: Elsevier Science, 2015. Citado 2 vezes nas páginas 32 e 34.

- NIELSEN, H.; MOGUL, J.; MASINTER, L. M.; FIELDING, R. T.; GETTYS, J.; LEACH, P. J.; BERNERS-LEE, T. Hypertext Transfer Protocol – HTTP/1.1. RFC Editor, 1999. RFC 2616. (Request for Comments, 2616). Disponível em: <<https://www.rfc-editor.org/info/rfc2616>>. Acesso em: 25 de jun. de 2022. Citado 3 vezes nas páginas 24, 25 e 26.
- OSMANI, A. Getting started with Progressive Web Apps. 2015. Disponível em: <<https://addyosmani.com/blog/getting-started-with-progressive-web-apps/>>. Acesso em: 05 de ago. de 2022. Citado na página 33.
- QUASAR. CSS Visibility. 2022. Disponível em: <<https://quasar.dev/style/visibility>>. Acesso em: 05 de ago. de 2022. Citado na página 32.
- ROEBUCK, K. Object-Relational Mapping: High-impact Strategies - What You Need to Know. [S.l.]: Emereo Pty Limited, 2011. ISBN 9781743044759. Citado na página 30.
- SOUSA, J. Solidariedade Digital - Backend. 2022. Disponível em: <<https://github.com/jhmerlo/sd-backend>>. Acesso em: 16 de nov. de 2023. Citado na página 43.
- SOUSA, J. Solidariedade Digital - Frontend. 2022. Disponível em: <<https://github.com/jhmerlo/sd-frontend>>. Acesso em: 16 de nov. de 2023. Citado na página 59.
- TARRIÑO, F. B.; ALBALADEJO GEMMA, d. S. Aplicación multiclente para la creación de portales cautivos y gestión de los datos capturados. Dissertação (Mestrado), 2017. Disponível em: <<https://ddd.uab.cat/record/181601>>. Acesso em: 19 de dez. de 2023. Citado na página 16.
- UFES. Solidariedade Digital. 2021. Disponível em: <<https://proex.ufes.br/solidariedade-digital>>. Acesso em: 03 de jun. de 2022. Citado na página 13.
- VUE.JS. Introduction: Vue.js. 2022. Disponível em: <<https://vuejs.org/guide/introduction.html>>. Acesso em: 01 de ago. de 2022. Citado na página 31.