

ASSOCIAÇÃO EDUCACIONAL LUTERANA BOM JESUS / IELUSC

Meraki: Uma aplicação Full Stack desenvolvida em JavaScript

ACADÊMICO:

Roger Fernando de Oliveira Carvalho

ORIENTADOR:

Prof. M.e. Gabriel Caixeta Silva





ROGER FERNANDO DE OLIVEIRA CARVALHO

**MERAKI: UMA APLICAÇÃO FULL STACK DESENVOLVIDA EM
JAVASCRIPT**

JOINVILLE

2021

ROGER FERNANDO DE OLIVEIRA CARVALHO

**MERAKI: UMA APLICAÇÃO FULL STACK DESENVOLVIDA EM
JAVASCRIPT**

Trabalho de conclusão de curso para obtenção do título de graduação de tecnólogo em Sistemas para internet apresentado à Associação Educacional Luterana BOM JESUS/IELUSC .

Orientador: M.e. Gabriel Caixeta Silva

JOINVILLE

2021

Roger Fernando de Oliveira Carvalho

MERAKI: UMA APLICAÇÃO FULL STACK DESENVOLVIDA EM JAVASCRIPT

Trabalho de conclusão de curso para obtenção do título de graduação em Sistemas para Internet - Tecnólogo apresentado à Associação Educacional Luterana BOM JESUS/IELUSC

Banca Examinadora:

M.e. Gabriel Caixeta Silva
Associação Educacional Luterana BOM
JESUS/IELUSC

Esp. Cristofer de Sousa Pereira
Associação Educacional Luterana BOM
JESUS/IELUSC

M.a. Danielle Antunes Oliveira
Associação Educacional Luterana BOM
JESUS/IELUSC

Joinville, 22 de junho de 2021

RESUMO

Um dos principais obstáculos para quem está iniciando ou procurando se aprofundar na área de desenvolvimento de *software*, é a escolha das diversas e complexas tecnologias que será foco de estudo. Isso porque a escolha da tecnologia a ser estudada, direcionará o aprendiz a técnicas, processos e demais elementos específicos da tecnologia escolhida. Propõe-se através de consulta bibliográfica nas diversas esferas e fases do desenvolvimento de software, absorver as melhores técnicas, práticas, e tecnologias, e sintetizar através de um projeto de construção de um aplicativo de *streaming* de vídeo aulas. Este projeto tem complexidade suficiente para abranger as sofisticadas e complexas tecnologias utilizadas atualmente, como autenticação, bancos não relacionais, roteamento, serviços, componentização, disparo de *e-mail*, *upload* de arquivos, criptografia, entre outros. Com o advento do NodeJS e *frameworks* Javascript, criou-se uma *stack* que utiliza Javascript desde o banco de dados, passando pelo *back-end* até o *front-end*. Devido à flexibilidade dessa *stack* e somando com a facilidade de desenvolver todas as partes da aplicação utilizando uma linguagem única, e considerando os requisitos definidos pela aplicação, foi definido então as tecnologias que seriam empregadas no desenvolvimento do protótipo. Todo o processo, pontos positivos e negativos, foram descritos, concluindo-se que o JavaScript é uma excelente opção para construção de *software*, salvo casos específicos os quais os requisitos impedem sua utilização.

Palavras-chaves: JavaScript, desenvolvimento de software, tecnologias, aplicativo, streaming.

ABSTRACT

One of the main obstacles for those starting or looking to go deeper in software development is the choice of various and complex technologies that will be the focus of study. This doubt is because the choice of technology to be studied will direct the apprentice to techniques, processes, and other specific elements of the chosen technology. Through bibliographic consultation in the various spheres and stages of software development, we intend incorporate the best techniques, practices, and technologies through a project to build video lessons streaming application. This project is sufficiently complex to cover the most sophisticated and different technologies currently used, such as authentication, non-relational banks, routing, services, componentization, email triggering, file uploading, encryption, among others things. With the advent of NodeJS and frameworks Javascript, a full-stack was created, using Javascript from the database, passing through the back-end to the front-end. The flexibility of this stack and adding with the ease of developing all parts of the application using a single language and considering the requirements defined by the application, the technologies used in the development of the prototype were defined. The whole process, positive and negative points, was described, and it was concluded that JavaScript is an excellent option for building software, except in specific cases where the requirements prevent its use.

Key-words: JavaScript, software development, tecnologies, aplication, streaming.

LISTA DE ILUSTRAÇÕES

Figura 1 – Diagrama de blocos de um sistema robotizado de empacotamento	20
Figura 2 – Fluxo da especificação completa de um <i>design de software</i>	21
Figura 3 – Elementos do modelo de análise.	22
Figura 4 – Relacionamento entre elementos e os níveis do projeto.	23
Figura 5 – Exemplo de diagrama de atividades.	24
Figura 6 – Exemplo de diagrama de caso de uso.	25
Figura 7 – Exemplo de diagrama de sequência.	26
Figura 8 – Exemplo de diagrama de classe.	27
Figura 9 – Exemplo de diagrama de estado.	28
Figura 10 – Exemplo de <i>Extensible Markup Language</i> (XML) na arquitetura <i>Simple Object Access Protocol</i> (SOAP)	31
Figura 11 – Exemplo de requisição inspecionada via browser.	34
Figura 12 – Ranking de linguagens.	39
Figura 13 – Funcionamento do <i>event loop</i>	44
Figura 14 – Funções aninhadas formando um " <i>Callback Hell</i> ".	45
Figura 15 – Uso de <i>Async/Await</i> para tornar funções síncronas.	46
Figura 16 – Exemplos de rotas criadas com ExpressJS.	47
Figura 17 – Comparativos entre comando Yarn e NPM	51
Figura 18 – Comparativos Google Trends entre <i>Integrated Development Environment</i> (IDE)	52
Figura 19 – Código sem uso de <i>Template Engine</i>	54
Figura 20 – Código com uso do <i>Template Engine</i> padrão do Laravel (<i>Blade templates</i>).	55
Figura 21 – Comando utilizado para criar uma <i>migration</i> em Laravel	55
Figura 22 – Exemplo de uma <i>migration</i> em Laravel	56
Figura 23 – Exemplo de formulário <i>HyperText Markup Language</i> (HTML) para upload de arquivos.	57
Figura 24 – Uso básico do objeto XMLHttpRequest	60
Figura 25 – Uso básico do Fetch API	60
Figura 26 – Uso básico da biblioteca Axios	61
Figura 27 – Fluxo do processo Scrum	64
Figura 28 – Exemplo de quadro tradicional Kanban	65
Figura 29 – Representação gráfica de relacionamento entre entidades de um banco <i>Structured Query Language</i> (SQL).	66
Figura 30 – Representações indicativas de cardinalidade de relacionamentos entre entidades.	66

Figura 31 – Popularidade dos protocolos de <i>streaming</i>	71
Figura 32 – Exemplo do modelo de negócio CANVAS	73
Figura 33 – Macro áreas e os nove blocos do modelo de negócio CANVAS.	74
Figura 34 – Roteamento com middlewares	79
Figura 35 – Arquitetura de stream peer-to-peer e troca de fragmentos	80
Figura 36 – Logo Meraki	82
Figura 37 – Telas de autenticação: <i>Splash screen</i> , login e cadastro	82
Figura 38 – Telas de autenticação: Esqueci a senha, verificar código e trocar senha	83
Figura 39 – Tela inicial e pesquisar	83
Figura 40 – Tela inscrições e meus cursos	84
Figura 41 – Tela meus cursos e minha conta	84
Figura 42 – Mode de negócio do projeto MERAKI	85
Figura 43 – Diagrama de Raia descrevendo o fluxo de autenticação	87
Figura 44 – Diagrama de Raia descrevendo o fluxo de cadastro	88
Figura 45 – Diagrama de Raia descrevendo o fluxo de recuperação de senha . .	90
Figura 46 – Diagrama de caso de uso descrevendo os procedimentos do subsistema Cursos	91
Figura 47 – Diagrama de caso de uso descrevendo os procedimentos de atualização de dados cadastrais	92
Figura 48 – Diagrama de classe da UML representando as entidades da aplicação	93
Figura 49 – Diagrama de sequência representando o fluxo de <i>streaming</i> de vídeo	94
Figura 50 – Solicitação do vídeo pelo <i>player</i> do Aplicativo (APP) Meraki.	95
Figura 51 – Encaminhamento da requisição do vídeo para a rota <i>stream</i>	96
Figura 52 – <i>Token middleware</i>	97
Figura 53 – <i>Token authentication</i>	97
Figura 54 – <i>Controller streams</i>	98
Figura 55 – <i>Ranges stream</i>	99
Figura 56 – Estrutura de pastas do projeto	101
Figura 57 – Estrutura da pasta <i>assets</i>	101
Figura 58 – Estrutura da pasta <i>config</i>	102
Figura 59 – Estrutura da pasta <i>routes</i>	102
Figura 60 – Estrutura da pasta <i>src</i>	103
Figura 61 – Estrutura da pasta <i>templates</i>	104
Figura 62 – Estrutura das pastas <i>test</i> , <i>upload</i> e <i>utils</i>	105
Figura 63 – Arquivos da raiz do projeto	106
Figura 64 – Scripts do package.json	106
Figura 65 – Arquivo index.js	107
Figura 66 – Estrutura de pastas do aplicativo Meraki	108
Figura 67 – Estrutura da pasta SRC	109

Figura 68 – Funcionamento básico da *Aplication Programming Interface* (API)
Meraki 112

LISTA DE TABELAS

Tabela 1 – Principais códigos de status de sucesso	35
Tabela 2 – Principais códigos de status de erro por parte do cliente	36
Tabela 3 – Principais códigos de status de erro genérico	36
Tabela 4 – Principais códigos de status de erro por parte do servidor	37
Tabela 5 – Mapeamento dos métodos <i>HyperText Transfer Protocol</i> (HTTP) para CRUD no protocolo <i>Representational State Transfer</i> (REST).	38
Tabela 6 – Números dos repositórios do Git Hub dos projetos de IDE	53
Tabela 7 – Princípios compartilhados entre os métodos ágeis.	62
Tabela 8 – Analise comparativa Relacional X Não Relacional	68
Tabela 9 – Atributos da qualidade de software	72
Tabela 10 – MEAN <i>stack</i> VS MERN <i>stack</i>	78

LISTA DE SIGLAS E ABREVIATURAS

AJAX *Asynchronous JavaScript and XML*

API *Application Programming Interface*

APP Aplicativo

ASF *Apache Software Foundation*

BMP *Bitmap*

CORBA *Common Object Architecture*

CRUD *Create, Retrieve, Update, Delete*

CSS *Cascading Style Sheets*

DCOM *Distributed Component Object Model*

DOM *Document Object Model*

ES5 *ECMAScript 5*

ES6 *ECMAScript 6*

EJS *Embedded JavaScript templates*

GIF *Graphics Interchange Format*

GPS *Global Position System*

HLS *HTTP Live Streaming*

HTML *HyperText Markup Language*

HTTP *HyperText Transfer Protocol*

I/O *Input/Output*

IBM *International Business Machines Corporation*

IDE *Integrated Development Environment*

IP *Internet Protocol*

Java EE *Java Enterprise Edition*

JPEG *Joint Photographic Experts Group*

JRE *Java Runtime Environment*

JS *JavaScript*

JSON *JavaScript Object Notation*

JSP *Java Server Pages*

LAMP *Linux, Apache, MySQL, PHP*

MEAN *MongoDB, Express, AngularJS, NodeJS*

MERN *MongoDB, Express, React, NodeJS*

MEVN *MongoDB, Express, VueJS, NodeJS*

MVC *Model, View, Controller*

MySQL *Structured Query Language Manager*

NoSQL *Not Only Standard Query Language*

NPM *Node Package Manager*

PHP *Hypertext Preprocessor*

PNG *Portable Network Graphics*

RDBMS *Relacional Database Management System*

REST *Representational State Transfer*

RMI *Remote Method Invocation*

RPC *Remote Procedure Call*

RTP *Real Time Protocol*

RTSP *Real Time Streaming Protocol*

SEAN *Sequelize, Express, AngularJS, NodeJS*

SERN *Sequelize, Express, React, NodeJS*

SEVN *Sequelize, Express, VueJS, NodeJS*

SGBD Sistema de Gerenciamento de Banco de Dados

SMTP *Simple Mail Transfer Protocol*

SOA *Service Oriented Architecture*

SOAP *Simple Object Access Protocol*

SPA *Single Page Application*

SQL *Structured Query Language*

SRC *Source*

SSL *Secure Sockets Layer*

SVN *Apache Subversion*

TCP *Transmission Control Protocol*

UDP *User Datagram Protocol*

UML *Unified Modeling Language*

URI *Uniform Resource Identifiers*

URL *Uniform Resource Identifiers*

VND *Vendor*

W3C *World Wide Web Consortium*

WAMP *Windows, Apache, MySQL, PHP*

WSDL *Web Services Description Language*

XHR *XML HTTP REQUEST*

XML *Extensible Markup Language*

SUMÁRIO

1	INTRODUÇÃO	16
1.1	OBJETIVOS	17
1.2	ESCOPO	17
1.3	METODOLOGIA	18
1.4	ESTRUTURA	18
2	REFERENCIAL TEÓRICO	19
2.1	MODELAGEM E ARQUITETURA DE SOFTWARE	19
2.1.1	Modelo do projeto de software	19
2.1.2	Arquiteturas e modelos de arquiteturas de software	28
2.1.3	Arquitetura de comunicação de serviços web	30
2.1.3.1	<i>Simple Object Access Protocol (SOAP)</i>	31
2.1.3.2	<i>Transferência representacional de estado (REST)</i>	33
2.2	STACKS DE DESENVOLVIMENTO	38
2.2.1	Stack PHP	39
2.2.2	Stack Java	41
2.2.3	Stack JavaScript	42
2.3	SOLUÇÕES DE DESENVOLVIMENTO	48
2.3.1	Versionamento de código	48
2.3.2	Gerenciamento de pacotes	50
2.3.3	IDE e editores	52
2.3.4	Ferramentas de análise estática de código	53
2.3.5	Templates Engines	54
2.3.6	Migrations	55
2.3.7	Upload de arquivos	57
2.3.8	Envio de e-mail	58
2.4	REQUISIÇÕES Asynchronous JavaScript and XML (AJAX)	59
2.4.1	XHR	59
2.4.2	Fetch API	60
2.4.3	Axios	61
2.5	METODOLOGIAS DE DESENVOLVIMENTO DE SOFTWARE	61
2.5.1	Metodologia Ágil	62
2.5.2	Scrum	62
2.5.3	Kanban	64
2.6	BANCO DE DADOS	65

2.6.1	Relacionais	65
2.6.2	Não relacionais	67
2.6.3	Relacional X Não relacional	67
2.7	STREAMING	68
2.7.1	Tipos e protocolos de streaming	69
2.7.2	Protocolos	70
2.7.3	Qualidade de software	71
2.8	MODELO DE NEGÓCIO	72
2.9	CONSIDERAÇÕES FINAIS	74
3	TRABALHOS RELACIONADOS	77
4	DESENVOLVIMENTO DA APLICAÇÃO MERAKI	81
4.1	PLANEJAMENTO	81
4.2	DESENVOLVIMENTO BACK-END	99
4.3	DESENVOLVIMENTO FRONT-END	107
4.3.1	Considerações do capítulo	111
5	RESULTADOS E DISCUSSÕES FINAIS	113
5.0.1	Trabalhos futuros	115
	REFERÊNCIAS	117

1 INTRODUÇÃO

Atualmente existe uma grande popularização de ferramentas, *softwares* e plataformas que operam na internet ou em redes privadas. O mercado de desenvolvimento de aplicações em geral vem crescendo ano após ano, juntamente ao uso crescente de dispositivos, sejam eles *desktops*, *smartphones*, *laptops*, *smart TVs*, entre outros (CRUZ; PETRUCELLI; SOTTO, 2018).

Muitos desses sistemas não foram desenvolvidos originalmente para operar na *web*, entretanto, as constantes evoluções nas tecnologias desse segmento de desenvolvimento de *software*, faz com que cada vez mais empresas e desenvolvedores migrem suas soluções e produtos para essa plataforma. Segundo Subramanian (2017), o desenvolvimento de aplicações *web* não era muito comum há poucos anos atrás. Como essa área se desenvolveu muito, alguns desenvolvedores se sentem frequentemente confusos sobre quais das opções é a melhor a seguir. Isto é aplicável não só para a escolha entre desenvolvimento *desktop*, *web*, embarcado ou *mobile*, mas também para as linguagens que são muitas, pois as tecnologias de desenvolvimento *web* não ficam restritas apenas a soluções da plataforma *web*, mas começaram a ser uma das alternativas para empresas de programação e startups de desenvolvimento *desktop*, *web*, aplicativos móveis, microcomputadores e assim por diante, conforme Cruz, Petrucelli e Sotto (2018).

Uma das opções de tecnologia é o JavaScript e parte da evolução mencionada só foi possível graças a sua evolução. Alguns desenvolvedores ainda tentam achar um caminho a seguir em meio a tantas linguagens e tecnologias. Mas essa linguagem em questão é vista como promissora pela comunidade de desenvolvedores das plataformas de versionamento de código. Tanto que se tornou atualmente a principal linguagem da *web*. Assim, GARBADE (2016) afirma que se vê claramente a evidencia dessa evolução ao observarmos essas plataformas como o GitHub, onde o JavaScript está sempre no topo das linguagens de programação quando observado o número de repositórios.

Neste trabalho, utilizando o JavaScript como única linguagem de programação, busca-se desenvolver uma aplicação de *stream* de vídeo voltado para educação, em uma arquitetura em camadas utilizando o conceito de API e banco de dados não relacional, de modo a demonstrar através de uma *stack* flexível as tecnologias e conceitos retirando a complexidade de uma *stack* multi-linguagem.

1.1 OBJETIVOS

O objetivo geral desse trabalho é criar uma aplicação no formato de API utilizando Javascript como linguagem de programação única capaz de abranger desde o banco de dados, o *back-end* até o *front-end*, de modo a demonstrar na prática que é possível criar aplicações *full-stack* com esta linguagem. Para atingir o objetivo geral, foram definidos os seguintes objetivos específicos:

- Escolher um tema para a construção da aplicação que abranja certa complexidade e variedade de funções a serem implementadas, a fim de obter dados suficientes para o estudo.
- Desenvolver uma API no padrão REST que contenha a lógica da aplicação e forneça e processe os dados a serem consumidos pelo *front-end*.
- Criar uma ou mais interfaces *front-end web* ou *mobile* para consumir a API e permitir a manipulação dos dados pelos usuários.

1.2 ESCOPO

A fim de delimitar o escopo desta pesquisa e do projeto prático, serão analisadas bibliografias relacionadas ou correlacionadas a alguma das fases do desenvolvimento do protótipo.

Referente ao protótipo, será desenvolvido um aplicativo que poderá ser executado em Android e IOS, porém os testes serão executados apenas na plataforma Android devido limitação de acesso a plataforma IOS.

Não será desenvolvido a aplicação para a plataforma *web* ou *desktop*, a fim de respeitar os prazos estabelecidos. Porém será considerado o desenvolvimento para plataforma web como uma sugestão futura.

Para obter uma complexidade suficiente a fim de demonstrar as principais tecnologias escolhidas após a análise bibliográfica, o projeto contemplará:

- Roteamento;
- Autenticação;
- Disparo de e-mail;
- Criptografia;
- *Create, Retrieve, Update, Delete* (CRUD);

- Persistência em banco;
- *Upload* e transmissão de arquivos;
- Artefatos de modelagem e arquitetura;
- Versionamento;

Todos os requisitos aqui não mencionados, serão considerados adicionais para o estudo, e se aplicados ao protótipo, será em caráter extra ou suplementar.

1.3 METODOLOGIA

Para o desenvolvimento deste trabalho, foi utilizado a metodologia ágil para o desenvolvimento da pesquisa e projeto utilizando o Kanban para organização das tarefas com *sprints* semanais. Para a realização da pesquisa, primeiramente foi feito um estudo bibliográfico, bem como estudo de documentos técnicos sobre o padrão REST, construção de APIs, a linguagem JavaScript, convenções e as melhores tecnologias e técnicas a serem aplicadas no desenvolvimento com essa linguagem. Como proposta de implementação de protótipo de software para avaliação dos resultados, foi desenvolvida a aplicação responsável por integrar as informações do banco de dados e arquivos, e entregá-los ao usuário através das implementações de técnicas e padrões estudados como roteamento, CRUD, autenticação, testes e validações. Também foi realizado um estudo técnico sobre streaming de vídeo e segmentação de arquivos na linguagem JavaScript devido ao tema escolhido para a construção da aplicação.

Foi realizado a análise qualitativa do protótipo implementado, observando se os requisitos foram atendidos, bem como os pontos positivos e negativos de se utilizar o JavaScript como linguagem única, com base no levantamento bibliográfico.

1.4 ESTRUTURA

A fim de documentar a pesquisa, este trabalho foi organizado da seguinte forma: O capítulo 2 contém o referencial teórico que detalha os tópicos do desenvolvimento *full stack*. No capítulo 3 contém alguns trabalhos relacionados a esta pesquisa. No capítulo 4 é descrito as etapas durante o desenvolvimento da solução proposta. Por fim, no capítulo 5 são abordados resultados, discussões finais e trabalhos futuros.

2 REFERENCIAL TEÓRICO

Este capítulo destina-se a apresentar os conceitos relacionados as etapas do desenvolvimento *full stack*, as opções de tecnologias, arquiteturas, padrões e técnicas para a construção de aplicações.

2.1 MODELAGEM E ARQUITETURA DE SOFTWARE

Pressman (2010) ao descrever o que é arquitetura de software faz uma analogia a arquitetura de edifícios, afirmando que a arquitetura é a maneira com a qual os vários componentes de uma construção são integrados para formar um todo de forma coesiva. É a maneira com a qual a construção se encaixa no ambiente e se mescla com outras construções. É o grau o qual a construção cumpre seu propósito e satisfaz as necessidades do proprietário. É o sentimento estético da estrutura, o impacto visual da construção, a forma das texturas, cores e materiais que foram combinados.

Entretanto, para descrever de forma sucinta, pode-se dizer que um projeto arquitetônico de software representa uma estrutura de dados e componentes necessários para construir um programa segundo Pressman (2010). Já Sommerville (2011) explica de forma resumida que o projeto de arquitetura trata da compressão de como organizar o sistema, bem como sua estrutura geral, sendo o primeiro estágio no processo de desenvolvimento de software.

A arquitetura não é o software em si no sentido operacional. É uma representação que permite analisar a eficácia do projeto sobre os requisitos, considerar modificações arquitetônicas quando ainda é fácil de realiza-las e reduzir riscos na construção do software segundo Pressman (2010).

2.1.1 Modelo do projeto de software

As arquiteturas de software são geralmente modeladas utilizando diagramas de blocos conforme explica Sommerville (2011). Um exemplo de um diagrama de bloco pode ser observado na Figura 1. Os componentes básicos de um diagrama de bloco são caixas que são interligadas por flechas. As caixas representam um componente ou subcomponente. Já as flechas indicam o fluxo dos dados e sinais de controle entre os componentes.

Figura 1 – Diagrama de blocos de um sistema robotizado de empacotamento.



Fonte: Adaptado de Sommerville (2011).

Conforme Sommerville (2011) os diagramas de bloco facilitam a comunicação e entendimento entre as equipes das diversas áreas envolvidas na construção do software, incluindo os *stakeholders* (partes interessadas) da aplicação. O autor também fala que esse tipo de diagrama representa uma imagem de alto nível do sistema. Sendo assim não possui muitos detalhes técnicos, o que facilita o entendimento para os stakeholders, mas é um diagrama pobre no sentido de documentar a arquitetura, pois existem outros modelos mais completos e detalhados e com semântica bem definida. Contudo algumas pessoas acreditam que detalhar a documentação não vale o custo do seu desenvolvimento, preferindo uma documentação mais rápida.

Pressman (2010) fala ainda que existem 3 pontos principais do porque a arquitetura de software é importante:

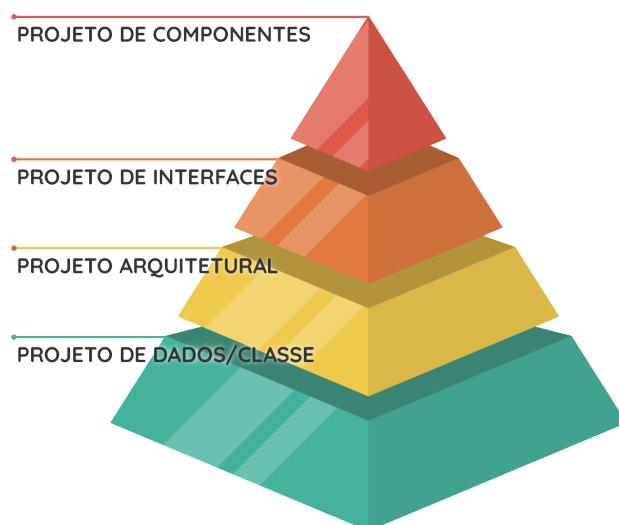
- A representação da arquitetura de um software facilita a comunicação com *stakeholders*, no desenvolvimento do sistema.
- A arquitetura destaca decisões de design iniciais que terão grande impacto em

todo o trabalho que se segue.

- A arquitetura constitui um modelo pequeno e compreensível de como o sistema está estruturado e como os seus componentes funcionam juntos.

Apesar dos desafios deparados durante a criação do projeto de software, Pressman (2010) descreve um modelo de projeto que ajuda a organizar o projeto como um todo, garantindo sua qualidade. Esse modelo de projeto é dividido em 4 partes conforme explica Pressman (2010). Essas partes seguem um fluxo, uma hierarquia, uma ordem, onde cada um dos níveis do modelo projeto do *software* limita as escolhas do nível seguinte. E cada escolha feita durante o projeto do *software*, deve ser considerado os requisitos para a tomada de decisões. Na Figura 2 podemos observar a representação do modelo de projeto proposto por Pressman (2010).

Figura 2 – Fluxo da especificação completa de um *design de software*



Fonte: Adaptado de Pressman (2010).

Segundo Pressman (2010), para se ter uma definição completa do projeto, é necessário criar o modelo de projeto com os 4 níveis. São eles:

- O projeto de dados/classes transforma os modelos de classe em realizações de classe de projetos e nas estruturas de dados dos requisitos necessários para implementar o software... Parte do projeto de classes pode ocorrer com o projeto da arquitetura.
- O projeto arquitetural define relacionamentos entre os principais elementos estruturais, estilos arquitetônicos, padrões de projeto e as restrições que afetam a

arquitetura que possa ser implementada. A representação do projeto arquitetural é derivada do modelo de requisitos.

- O projeto de interfaces descreve as interfaces do software. Não só as interfaces de comunicação com os usuários, mas também as interfaces de comunicação com outras partes do sistema, subsistemas e até outros sistemas.
- O projeto de componentes é o responsável por transformar elementos estruturais da arquitetura de software em uma descrição procedural dos componentes de software. As informações obtidas servem como base para o projeto de componentes.

Conforme explica Pressman (2010), o *design do software* é composto pelos requisitos da aplicação na forma de artefatos. Esses artefatos são reunidos em grupos denominados de elementos do modelo de análise. E cada elemento possui um conjunto de artefatos. A representação desse agrupamento é demonstrada na Figura 3.

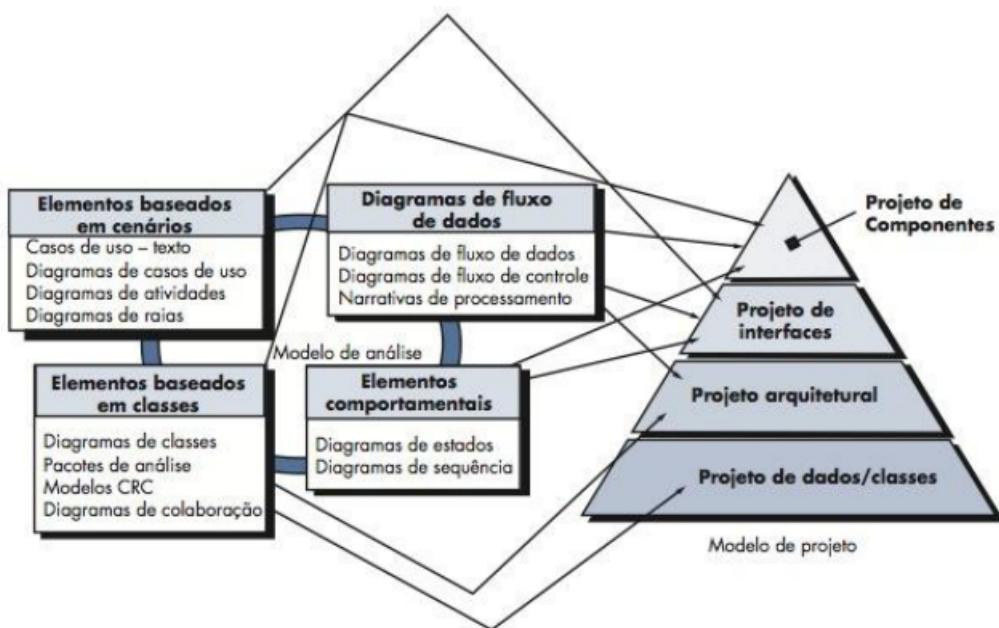
Figura 3 – Elementos do modelo de análise.



Fonte: Adaptado de Pressman (2010).

Dado os elementos mencionados, Pressman (2010) faz a associação de cada um deles aos níveis do modelo de projeto. Essa associação é representada na Figura 4.

Figura 4 – Relacionamento entre elementos e os níveis do projeto.



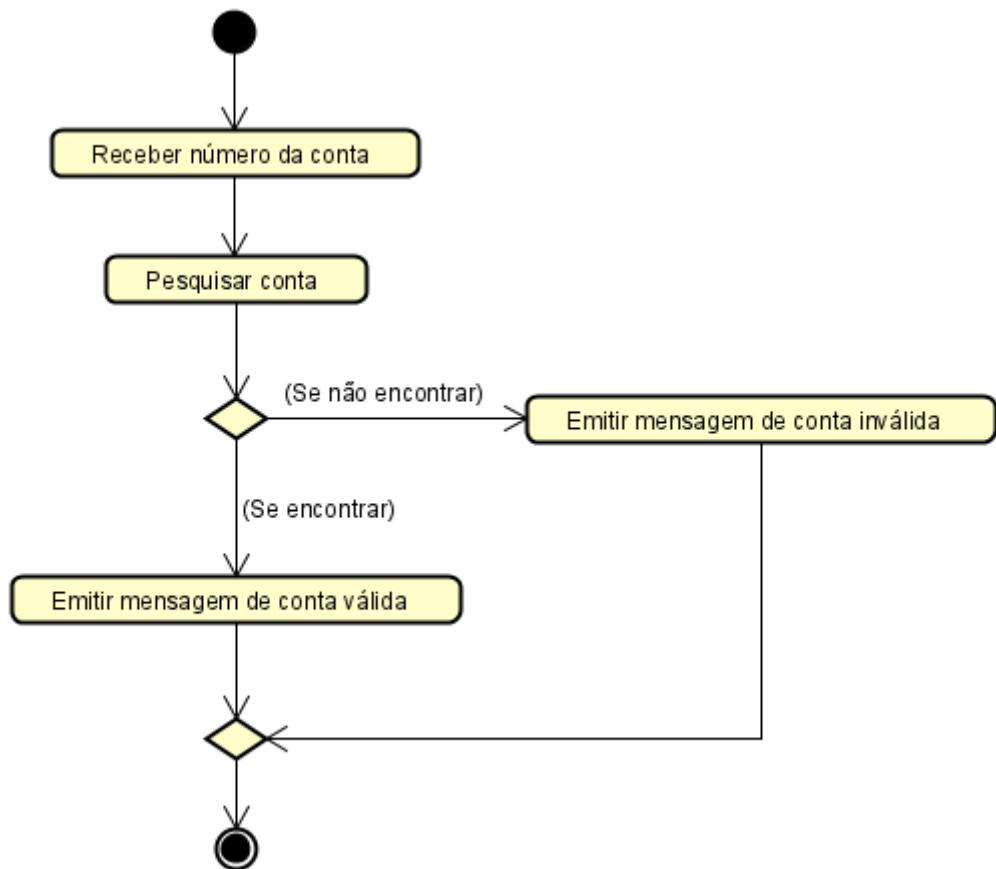
Fonte: Pressman (2010).

Apesar de Pressman (2010) afirmar que um modelo de projeto completo é representado por todos os elementos e artefatos já mencionados, Sommerville (2011) diz que a maioria dos usuários do *Unified Modeling Language* (UML) (linguagem padrão para estrutura do projeto de software), acreditam que 5 artefatos podem representar a essência do sistema. Conforme descreve Sommerville (2011), os 5 artefatos são:

- Diagramas de atividades, que mostram as atividades envolvidas em um processo ou no processamento de dados. O diagrama de atividades UML, representa de forma gráfica um fluxo de interação em um determinado cenário conforme Pressman (2010). Dessa forma, o diagrama de atividades da UML, complementa o caso de uso. Sua notação é similar a um fluxograma. Funções do sistema são representadas por retângulos com cantos arredondados, setas representando um fluxo pelo sistema, losangos representando uma decisão ramificada com sua descrição e linhas horizontais para indicar atividades paralelas. Também existe

o diagrama de raias, que é uma variação do diagrama de atividades conforme (PRESSMAN, 2010). A diferença é que no diagrama de raias existem separações onde é possível visualizar as possíveis ações de vários atores ao mesmo tempo. Outras notações não mencionadas pelo autor são observadas na Figura 5, que mostra um exemplo básico de um diagrama de atividades.

Figura 5 – Exemplo de diagrama de atividades.



Fonte: O próprio autor.

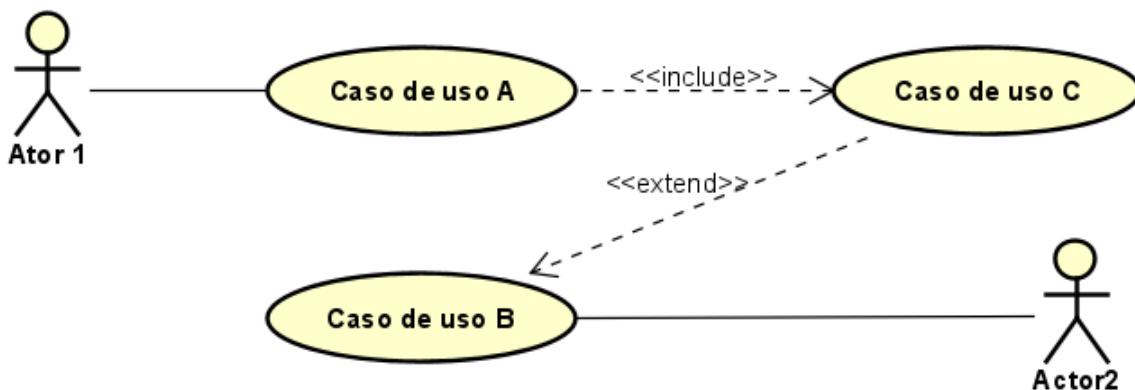
- Diagramas de casos de uso, que mostram as interações entre um sistema e seu ambiente. A modelagem do caso de uso foi incorporada na primeira *release* da UML, e é amplamente usada para apoiar o levantamento de requisitos. De forma sucinta, *um caso de uso pode ser tomado como um cenário simples que descreve o que o usuário espera de um sistema (SOMMERVILLE, 2011)*.

Existem muitas formas de medir o sucesso de um sistema, mas a satisfação do usuário de um sistema está no topo da lista. Então para entender como os

usuários (e outros atores) querem interagir com o sistema deve-se começar com a criação de cenários de uso, segundo Pressman (2010).

Conforme ambos autores Pressman (2010) e Sommerville (2011) exemplificam, um caso de uso de um sistema ou subsistema é representado por uma elipse, os atores por bonecos palito (Notação desenvolvida originalmente para representar a figura humana, mas também utilizada para representar outros sistemas e hardwares segundo Sommerville (2011)), linhas que indicam fluxo de mensagens, quadros delimitadores do software e associações entre casos de uso dos tipos *include* (que indica que o primeiro caso de uso sempre incluirá o segundo caso de uso) e *extend* (que indica que o primeiro caso de uso pode incluir o segundo). O diagrama de caso de uso da UML é uma visão simplificada de uma interação, podendo ser acompanhada de outros artefatos mais detalhados. Um exemplo de um diagrama de caso de uso é observado na Figura 6.

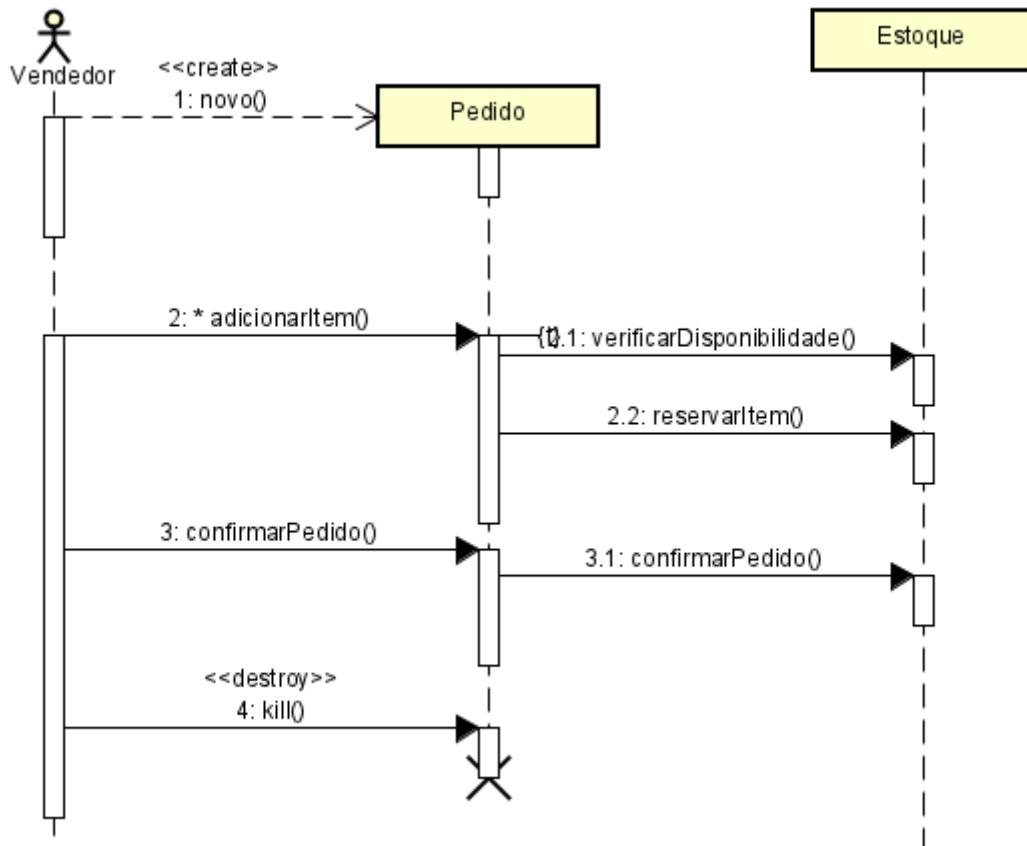
Figura 6 – Exemplo de diagrama de caso de uso.



Fonte: O próprio autor.

- Diagrama de sequência, que mostram as interações entre os atores e o sistema que ocorrem durante um caso de uso em específico. Dessa forma, esse diagrama também complementa o caso de uso. Sua notação básica é representada por objetos e atores alocados na parte superior do diagrama com uma linha tracejada alinhada verticalmente a partir deles. Interações entre eles são representados por uma seta com descrição textual. Os retângulos nas linhas tracejadas representam o tempo em que ocorre o processamento do objeto em questão. A leitura desse diagrama deve ocorrer verticalmente de cima para baixo. Um exemplo do diagrama de sequência da UML é observado na Figura 7.

Figura 7 – Exemplo de diagrama de sequência.



Fonte: O próprio autor.

- Diagramas de classe, que mostram as classes de objeto no sistema e as associações entre elas. Resumidamente, uma classe de objeto é a representação de um objeto do sistema, e a associação é um link entre classes que indicam relacionamento segundo Sommerville (2011). Os diagramas de classe são utilizados para mostrar classes e associações em um sistema orientado a objeto conforme diz Sommerville (2011).

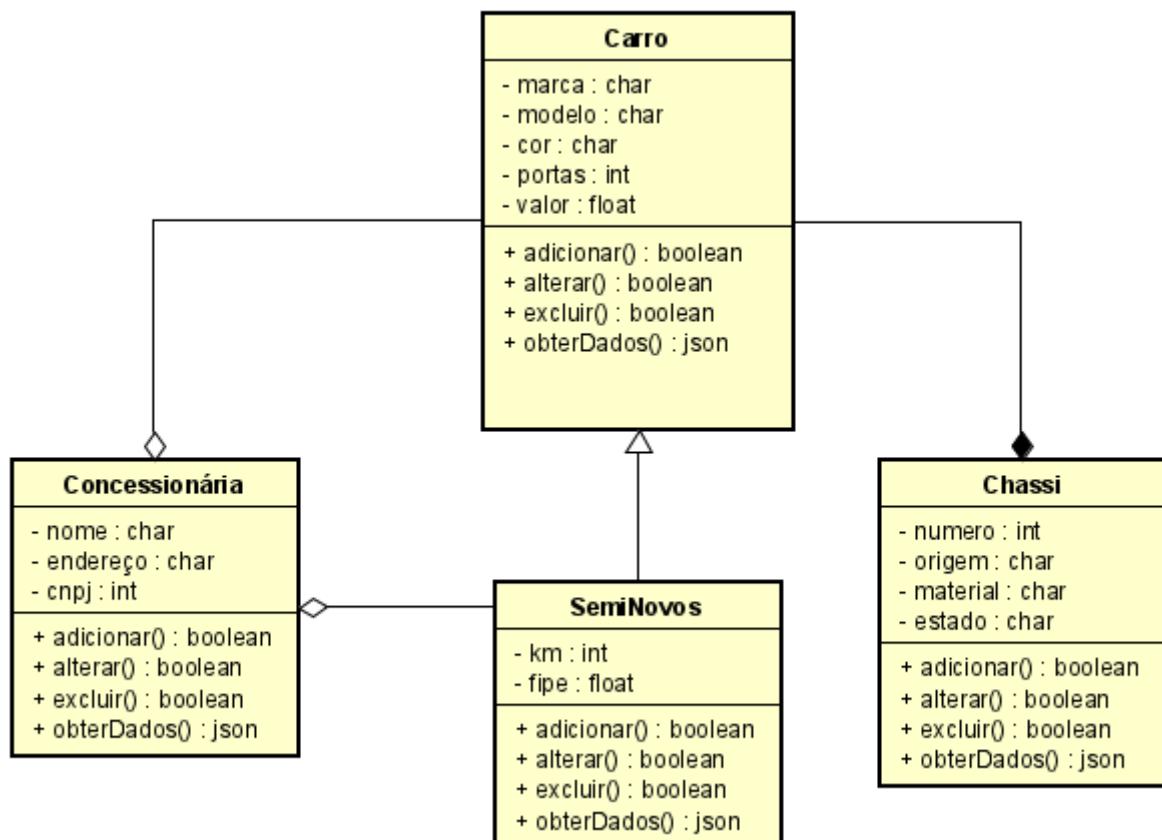
Na UML uma classe é representada por um retângulo que contém seu nome, atributos e as operações (chamadas de métodos em linguagens de programação). As associações entre classes são nomeadas no diagrama de classe e também é exibido a cardinalidade do relacionamento segundo Sommerville (2011).

Sommerville (2011) explica que no diagrama de classes da UML aparecem também conceitos de relacionamento como generalização e agregação. O autor explica que a generalização ocorre quando classes de nível baixo são subclasses e herdam atributos e operações de classes superiores denominadas superclasses.

Já a agregação ocorre quando uma classe é formada por objetos de outras classes, sendo as classes agregadoras existentes por si mesmas e independentes da classe agregada sendo representada por um losango vazado junto a classe agregadora. Pressman (2010) exemplifica uma terceira notação de relacionamento chamada de composição que ocorre quando uma classe é formada por outras sendo essas outras classes estritamente próprias da classe composta, e não compartilhada, sendo representada por um losango preto junto a classe composta.

Um exemplo de diagrama de classes da UML é observado na Figura 8.

Figura 8 – Exemplo de diagrama de classe.

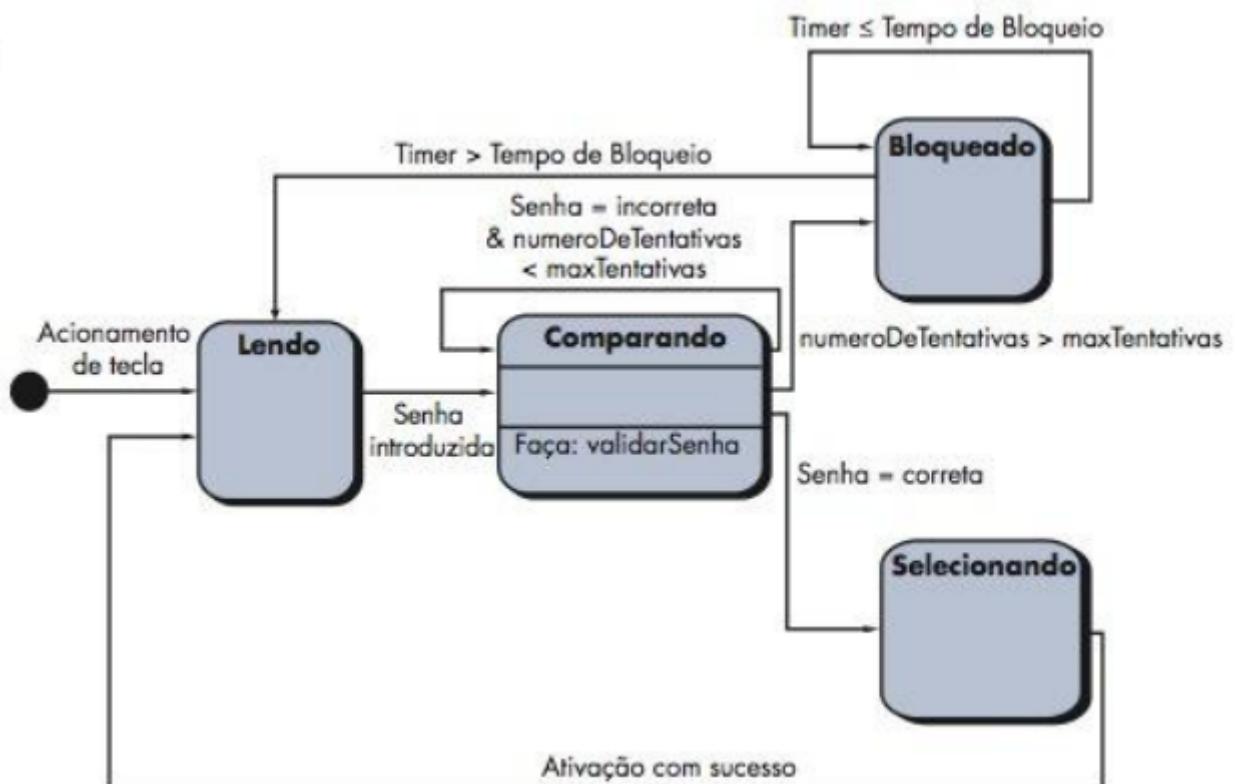


Fonte: O próprio autor.

- Diagramas de estado, que mostram como o sistema reage aos eventos internos e externos. É o componente de um modelo comportamental. Ele representa estados ativos para as classes e eventos (gatilhos). Os gatilhos por sua vez, provocam mudanças entre os estados ativos. Em sua notação, as setas representam

a transição de um estado ativo de um objeto para outro. Os objetos são representados por caixas, contendo o nome do objeto, podendo ter também alguma ação o qual o objeto é responsável. Descrições dos eventos fazem parte da notação. E o início do fluxo é representado pelo círculo preto. É possível observar um exemplo de um diagrama de estados da UML na Figura 9.

Figura 9 – Exemplo de diagrama de estado.



Fonte: (PRESSMAN, 2010).

2.1.2 Arquiteturas e modelos de arquiteturas de software

A modelagem do projeto do software estudada na seção anterior é muito importante para definirmos qual será a arquitetura utilizada. Segundo Sommerville (2011), "Você pode pensar em um padrão de arquitetura como uma descrição abstrata, estilizada, de boas práticas, experimentadas e testadas em diferentes sistemas e ambientes.". Pressman (2010) diz que nos últimos 60 anos milhões de sistemas foram criados, mas que a grande maioria pode ser classificada em um pequeno número de padrões de arquitetura. São eles:

- Arquitetura centralizada em dados ou quadro-negro segundo Pressman (2010) ou arquitetura de repositórios segundo Sommerville (2011), basicamente é centralização dos dados em um único local que é acessado por todos os clientes. Os clientes operam de forma independente e os dados do repositório podem ser alterados sem comprometer os demais clientes, promovendo assim a integrabilidade, conforme explica Pressman (2010). No entanto Sommerville (2011) explica que a sua distribuição em redes em computadores diferentes pode causar problemas quanto a redundância e inconsistência de dados.
- Arquitetura de fluxo de dados segundo Pressman (2010) ou arquitetura de dutos e filtros segundo Sommerville (2011), é uma arquitetura que se aplica quando se deve manipular os dados de saída. Essa manipulação ocorre por meio de componentes que são interligados. Os componentes são denominados filtros, e as interligações são os tubos ou dutos. os componentes operam de forma independente dos demais componentes acima ou abaixo deles, e são projetados para receber e enviar os dados em um determinado padrão para que os componentes entendam o conteúdo, no entanto os componentes não precisam saber o funcionamento dos demais, conforme explica Pressman (2010). Segundo Sommerville (2011) um exemplo de uso dessa arquitetura, são os aplicativos de processamento em lote.
- Arquitetura de chamadas e retornos segundo Pressman (2010), permite-nos ter uma estrutura relativamente fácil de modificar e escalar. Existem uma série de subcategorias dessa arquitetura como a arquitetura de programa principal/sub-programa, que separa uma função em uma hierarquia, no qual o programa principal é acionado, e este invocará uma série de componentes que invocarão outros componentes. Existe também a subcategoria chamada arquitetura de chamadas a procedimentos remotos, onde os componentes dessa arquitetura são distribuídos ao longo de vários computadores.
- Arquitetura orientada a objetos, segundo Pressman (2010), funciona com o encapsulamento dos dados pelos componentes do sistema e troca de mensagem para comunicação entre os componentes.
- Arquiteturas em camadas segundo Pressman (2010), separa camadas o sistema, onde cada camada realiza operações de modo que na camada mais externa, é tratado operações de interface do usuário, e na camada mais interna, é trabalho operações mais relacionadas ao sistema operacional. As camadas intermediárias, são responsáveis por funções utilitárias, e funções da aplicação. Sommerville (2011) complementa dizendo que cada camada só depende da camada imediatamente abaixo dela. Além de ser uma arquitetura mutável e por-

tável, ou seja, enquanto a interface permanecer inalterada, pode se trocar uma camada por outra equivalente sem afetar. E quando uma camada é alterada, somente a camada seguinte é afetada.

Pressman (2010) ainda fala que estes padrões arquiteturais são algumas das várias arquiteturas existentes. E que quando os requisitos forem levantados, características do sistema que se pretende desenvolver pode se escolher a arquitetura ou combinação de padrões arquiteturais que melhor sirva ao propósito da aplicação.

2.1.3 Arquitetura de comunicação de serviços web

O termo *service* (serviço) tem sido usado para se referir a qualquer software que realiza alguma tarefa de negócio, fornece acesso a arquivos ou executa funções genéricas como autenticação e registro (DAIGNEAU, 2012). Existem muitas maneiras de implementar serviços utilizando uma arquitetura de comunicação para que seja possível a transmissão dos dados. Durante a construção de um serviço, uma das escolhas que deve ser feita, é qual arquitetura será utilizada para a transmissão de dados desse serviço. Essa escolha dependerá da plataforma no qual o serviço está sendo executado.

Em seu livro, Daigneau (2012) descreve uma plataforma como sendo qualquer combinação de *hardware*, sistema operacional e linguagem de programação. Cada combinação de plataformas, possuem opções específicas de arquitetura para a comunicação de um serviço. Na plataforma Windows por exemplo, utiliza-se a arquitetura *Distributed Component Object Model* (DCOM). Já na plataforma Java utiliza-se *Remote Method Invocation* (RMI), com Unix/Linux é usado Sun *Remote Procedure Call* (RPC), para plataforma *web* uma opção é o SOAP, entre outras opções.

REST e SOAP são duas das arquiteturas de comunicação muito utilizadas atualmente. Ambas são arquiteturas específicas para a plataforma *web*. Os serviços desenvolvidos para a plataforma *web* são chamados de *web service* (Serviço de internet) conforme descreve Daigneau (2012), o qual explica também, que a principal característica desse tipo de serviço, é o protocolo de comunicação utilizado pelos Web Services, o HTTP. Ou seja, se comunicam pela *web* (internet e/ou intranet), através da publicação, localização, invocação destes serviços em redes que utilizam o *Transmission Control Protocol* (TCP) e *Internet Protocol* (IP), conforme também descreve Machado et al. (2006).

Outra característica importante dos *web services*, conforme descreve Machado et al. (2006), é o seu fraco acoplamento. Isso permite que as camadas do serviço rodem em computadores diferentes, com linguagens e plataformas diferentes, desde que utilizem o mesmo padrão de comunicação. Isso possibilita o compar-

tilhamento de funções do serviço em aplicativos e plataformas distintas (DAIGNEAU, 2012). Por exemplo, posso executar um aplicativo para na plataforma Android que consome o mesmo serviço que um aplicativo na plataforma IOS ou Windows, ou Linux.

2.1.3.1 Simple Object Access Protocol (SOAP)

Conforme explica Saudate (2014b) o SOAP foi criado e é mantido pela *World Wide Web Consortium* (W3C), comunidade que define as normas para *web* (*Web standards*).

Segundo Machado et al. (2006), SOAP é um protocolo leve para a troca de dados estruturados com XML pela *Web*. Serve como um envelopamento de um documento XML para que este possa ser transmitido pela *Web*. Um dos objetivos do design do SOAP é encapsular e fazer chamadas RPC (*Remote Procedure Call*) usando a extensibilidade e flexibilidade do XML (SILVEIRA et al., 2011)

Conforme Saudate (2014b), O SOAP também é conhecido como Envelope SOAP, devido a estrutura em XML que tem como elemento raiz o "Envelope". Obedecendo o seguinte formato conforme mostra a Figura 10.

Figura 10 – Exemplo de XML na arquitetura SOAP

```

1 <soapenv:Envelope
2   xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
3   xmlns:ser="http://servicos.estoque.knight.com/">
4   <soapenv:Body>
5     <ser:listarLivros />
6     </soapenv:Body>
7   </soapenv:Envelope>
8

```

Fonte: Saudate (2014b).

Segundo Saudate (2014b), ao observar essa estrutura básica, nota-se que todos as informações da requisição estão dentro de uma *tag* principal chamada "soapenv:Envelope", que é justamente um contêiner para as *tags* de *Header* e *Body* da requisição. O *Body* é o corpo da nossa requisição, onde estão o nome da operação, parâmetros etc. Já no *Header* ficam os metadados da requisição, como chave de autenticação, endereço de retorno da mensagem etc.

Esse tipo de mensagem XML envelopadas pelo protocolo SOAP são enviadas através de um protocolo de internet, geralmente o protocolo HTTP. Também é possível

utilizar *Web Services Description Language* (WSDL), que serve para descrever como o *web service* funciona. conforme explica Machado et al. (2006).

Geralmente pode-se acessar o WSDL de um *web service* acrescentando "?wsdl" ao final do endereço onde o *web service* está hospedado conforme menciona Oliveira (2014). Oliveira (2014) diz também que além de dados, é possível saber como o *web service* funciona através da tag "*documentation*", que possibilita a inclusão da documentação do *web service*, dispensando a criação de um documento auxiliar para explicar os objetivos do serviço, pois no WSDL já constará as informações necessárias para invocar os *End-points*(Métodos) e seus parâmetros.

Até o ano de 2009 o SOAP era considerada a única solução para arquitetura de serviços web, mas nesse ano, foi criado o *Service Oriented Architecture* (SOA) Manifesto, que trazia um conjunto de características importantes para essa arquitetura, e segundo o SOA *Manifesto*, é possível implementar uma arquitetura orientada a serviços em cima do *Common Object Architecture* (CORBA), SOAP, DCOM (Distributed Component Object Model), REST, etc, conforme explica Saudate (2014b) em seu livro.

Apesar do surgimento do REST (outro tipo de *web service*), ainda existem muitos *web services* do tipo SOAP, tais como Nota fiscal eletrônica e TISS (Troca de informações na Saúde Suplementar da ANS), conforme descreve Oliveira (2014).

Para SILVEIRA et al. (2011), o SOAP deixou de lado pontos que o protocolo HTTP já suporta atualmente e cada inclusão de extensão para suprir esses pontos torna o sistema mais complexo. Portanto deve se pensar sobre o benefício e desvantagens da aplicação do protocolo. O SOAP, após mais de 10 anos de existência, apresenta hoje altos custos e tem se mostrado de difícil manutenção devido à alta abstração do protocolo utilizado (SILVEIRA et al., 2011)

Segundo SILVEIRA et al. (2011), o mercado como um todo está cada vez mais abandonando a adoção do SOAP. O Google por exemplo deixou de realizar buscas através de SOAP em 2009 (Mesmo ano em que foi publicado o SOA Manifesto).

SILVEIRA et al. (2011) quando diz que atualmente as vantagens técnicas do SOAP não valem mais a pena, e que ele deve ser usado apenas em sistemas os quais já estejam disponibilizados com essa arquitetura, corrobora com a ideia de desuso da tecnologia SOAP.

Conforme SILVEIRA et al. (2011), muitas ferramentas envolvidas no processo de desenvolvimento de software não suportam ou não encorajam o uso das funcionalidades das versões mais recentes do SOAP, fazendo este perder vantagem se comparado ao HTTP. Além disso, os XMLs envelopados nas requisições e respostas costumam ser muito grandes, deixando a performance mais lenta dependendo da infraestrutura.

2.1.3.2 Transferência representacional de estado (REST)

Segundo Saudate (2014a), REST é um estilo de desenvolvimento de *web services*. Saudate (2014b) também define REST como sendo uma arquitetura criada por um dos criadores do protocolo HTML, o doutor Roy Fielding, em sua tese de doutorado.

Para Saudate (2014a), pelo fato do REST ter sido criado pela mesma pessoa que criou o protocolo HTTP, faz com que o protocolo REST seja guiado pelas boas práticas do HTTP, como o uso adequado dos métodos (*GET, POST, PUT, DELETE, HEAD, OPTIONS*), uso adequado de endereços, códigos de status padronizados, uso adequado dos cabeçalhos HTTP (*Headers*), dentre outros recursos.

Para Saudate (2014a), além dos princípios e boas práticas mencionadas, a principal marca do REST, foi a utilização do conceito de recurso. No REST tudo é recurso, e esses recursos são representados por *Uniform Resource Identifiers* (URLs) ou *Uniform Resource Identifiers* (URIs) as quais são os identificadores dos recursos. URL e URI na *web* são a mesma coisa segundo Saudate (2014a). Os recursos possuem operadores (Métodos HTTP), com os quais ocorrem as mudanças de estado da aplicação.

Saudate (2014b) explica que a própria *web* funciona baseada nesses princípios, e pode se observar isso ao abrir uma página na internet que automaticamente estará utilizando esses conceitos. Quando uma URL é inserida no *browser* (navegador), uma requisição é disparada utilizando o método *GET* ao recurso que se encontra no endereço informado. Ao informar uma URL no browser, você está requisitando um recurso via método *GET*. Esse recurso pode ser por exemplo um arquivo HTML (HyperText Markup Language), "index.html" por exemplo, o qual é servido no endereço que você informou no *browser*. O servidor que hospeda esse arquivo, irá retornar uma resposta ao *browser* contendo o arquivo e algumas informações como o *media type* e o *Headers*. O *media type* é o tipo do arquivo o qual o *browser* está recebendo. No exemplo mencionado, o *media type* seria "text/html". Isso permite que o *browser* saiba que tipo de arquivo está recebendo, para que possa renderizá-lo de forma apropriada. Já o *Headers* conforme Saudate (2014b) são os cabeçalhos de uma requisição ou de uma resposta, o qual ficam contidos metadados como o *status code*, *token*, endereço de resposta etc.

No exemplo mencionado, pelo fato de o recurso requisitado ser um HTML, o *browser* irá procurar no arquivo referências a outros recursos como folhas de estilos, scripts e imagens, e para cada um dos recursos, irá fazer uma nova requisição para busca-los e renderizá-los. Uma *tag* de imagem (*img*) dentro do arquivo HTML por exemplo, o *browser* realiza uma requisição para o endereço do *Source* (SRC) da

imagem, e a resposta então retorna os dados da imagens que o *browser* renderiza na tela. O retorno dessa imagem, virá com o *media type* conforme o tipo da imagem como por exemplo *Joint Photographic Experts Group* (JPEG), *Graphics Interchange Format* (GIF), *Portable Network Graphics* (PNG), *Bitmap* (BMP) etc, para que o *browser* saiba como renderizá-la apropriadamente conforme menciona Saudate (2014b).

Pode ser observado o conteúdo dos *requests* e *responses*, ao inspecionar uma página web. Os *browsers* atuais possuem ferramentas que possibilitam a visualização dessas informações. No Chrome, por exemplo, basta pressionar o botão F12 para abrir o inspetor. Nele é possível obter muitas informações. Uma delas são as requisições. A Figura 11 exibe uma requisição à URL <https://faculdade.ielusc.br/>, bem como o Método HTTP utilizado, O *status code* da resposta, o *content type*, dentre outras informações. Também após ser feita essa requisição, por conta do *content type* ser do tipo HTML, o browser fez outras requisições a recursos de folha de estilo em cascata (*Cascading Style Sheets* (CSS)), scripts *JavaScript* (JS), e imagens que compõem a página.

Figura 11 – Exemplo de requisição inspecionada via browser.

Name	Headers	Preview	Response	Initiator	Timing	Cookies
faculdade.ielusc.br	Request URL: https://faculdade.ielusc.br/					
js?key=AlzaSyDlcsNJ6Krs_baci	Request Method: GET					
style.min.css?ver=5.3.2	Status Code: 200 OK					
style.min.css	Remote Address: 177.91.48.10:443					
bg-menu-white.png	Referrer Policy: strict-origin-when-cross-origin					
bg-menu.png						
bg-wave-block-1.png						
img-woman.png						
img-1-block-2.png						
img-2-block-2.png						
img-3-block-2.png						
analytics.js						

Fonte: O próprio autor.

Para Saudate (2014b), com esse conceito em mente, pode-se visualizar como funciona um serviço baseado em REST. Tendo uma URL pré-definida, o conteúdo que é retornado na resposta pode ser pré-estabelecido entre o cliente e servidor, utilizando as marcações padronizadas no resultado. Pode se ter diversos *media types* de recursos sendo requisitados e enviados. Os principais são *application*, *audio*, *image*, *text*, *video*, *Vendor (VND)*. Quando se trata de dados estruturados em REST, existem duas maneiras de representá-los, utilizando XML ou JSON, os quais devem conter

um *Header* com o *media types* da resposta, como *application/xml* e *application/json* respectivamente.

Segundo Saudate (2014a), o REST também prevê o uso correto dos *status code* HTTP, onde deve-se conhecê-los para aplicá-los corretamente. Os códigos iniciados com o número 1, são códigos informacionais. Já os iniciados com o número 2, são códigos que indicam sucesso.

A Tabela 1 mapeia os principais *status code* de sucesso (*status code* iniciados com 2).

Tabela 1 – Principais códigos de status de sucesso

Código	Descrição	Função
200	OK	Indica que a operação indicada teve sucesso.
201	Created	Indica que o recurso desejado foi criado com sucesso. Retorna um Header Location contendo a URL do recurso criado.
202	Accepted	Indica que a solicitação foi recebida e será processada em outro momento.
204	No Content	Indica que o servidor recusou a enviar conteúdo.
206	Partial Content	Indica que o conteúdo recebido é apenas parcial.

Fonte: Adaptado de Saudate (2014a)

A Tabela 2 mapeia os principais *status code* de erros por parte do cliente (*status code* iniciados com 3).

Tabela 2 – Principais códigos de status de erro por parte do cliente

Código	Descrição	Função
301	Moved Permanently	Significa que o recurso solicitado foi realocado permanentemente. Retorna um Header Location contendo a URL completa.
303	See Other	Indica que a requisição foi processada, mas o servidor não quer enviar a resposta.
304	Not Modified	Indica a resposta se houve alteração em relação a uma requisição anterior.
307	Temporary Redirect	Similar ao 301, mas indica que o redirecionamento é temporário, não permanente.

Fonte: Adaptado de Saudate (2014a)

A Tabela 3 abaixo mostra os principais *status code* de erros genéricos (*status code* iniciados com 5).

Tabela 3 – Principais códigos de status de erro genérico

Código	Descrição	Função
500	Internal Server Error	Indica erro Genérico, utilizada quando nenhuma outra se aplica.
503	Service Unavailable	Indica que o serviço não está funcionando corretamente.

Fonte: Adaptado de Saudate (2014a)

A Tabela 4 mostra os principais *status code* de erro por parte do servidor (*status code* iniciados com 4), e sua função.

Tabela 4 – Principais códigos de status de erro por parte do servidor

Código	Descrição	Função
400	Bad Request	Indica uma resposta genérica para qualquer tipo de erro de processamento cuja responsabilidade é do cliente.
401	Unauthorized	Indica que o cliente está tentando realizar uma operação sem ter autenticação válida
403	Forbidden	Indica que o cliente está tentando realizar uma operação sem ter a devida autorização.
404	Not Found	Indica que o recurso solicitado não existe.
405	Method Not Allowed	Indica que o método HTTP utilizado não é suportado pela URL. Retornará um Header Allow contendo os métodos suportados.
409	Conflict	Indica conflitos entre dois recursos. Comumente quando tentado criar um recurso que já existe.
410	Gone	Semelhante ao 404, mas indica que um recurso já existiu neste local.
412	Precondition failed	Indica resposta a requisições GET condicionais.
415	Unsupported Media Type	Indica que foi solicitado um tipo de dado não suportado.

Fonte: Adaptado de Saudate (2014a)

Conforme Saudate (2014a), outro conceito importante do REST, são as especificidades e as peculiaridades que cada método HTTP tem, como idempotência, segurança e mecanismo de passagem de parâmetros. Dessa forma a utilização de um determinado método HTTP, dependerá do tipo de operação desejada. A Tabela 5, exemplifica o uso dos métodos HTTP na arquitetura REST.

Tabela 5 – Mapeamento dos métodos HTTP para CRUD no protocolo REST.

Operação	Método HTTP	Recurso	Exemplo	Observação
Ler - Listar (Read - List)	GET	Coleção	GET /clientes	Lista objetos (parâmetros de URL adicionais podem ser incluídos para filtrar os campos).
Ler (Read)	GET	Object	GET /clientes/1234	Retorna um objeto individual (parâmetros de URL adicionais podem ser incluídos para filtrar os campos).
Criar (Create)	POST	Coleção	POST /clientes/1234	Cria um objeto, e o objeto é fornecido no corpo.
Atualizar (Update)	PUT	Object	PUT /clientes/1234	Substitui o objeto pelo objeto fornecido no corpo.
Atualizar (Update)	PATCH	Object	PATCH /clientes/1234	Modifica alguns atributos de um objeto especificados no corpo.
Excluir (Delete)	DELETE	Object	DELETE /clientes/1234	Exclui o objeto.

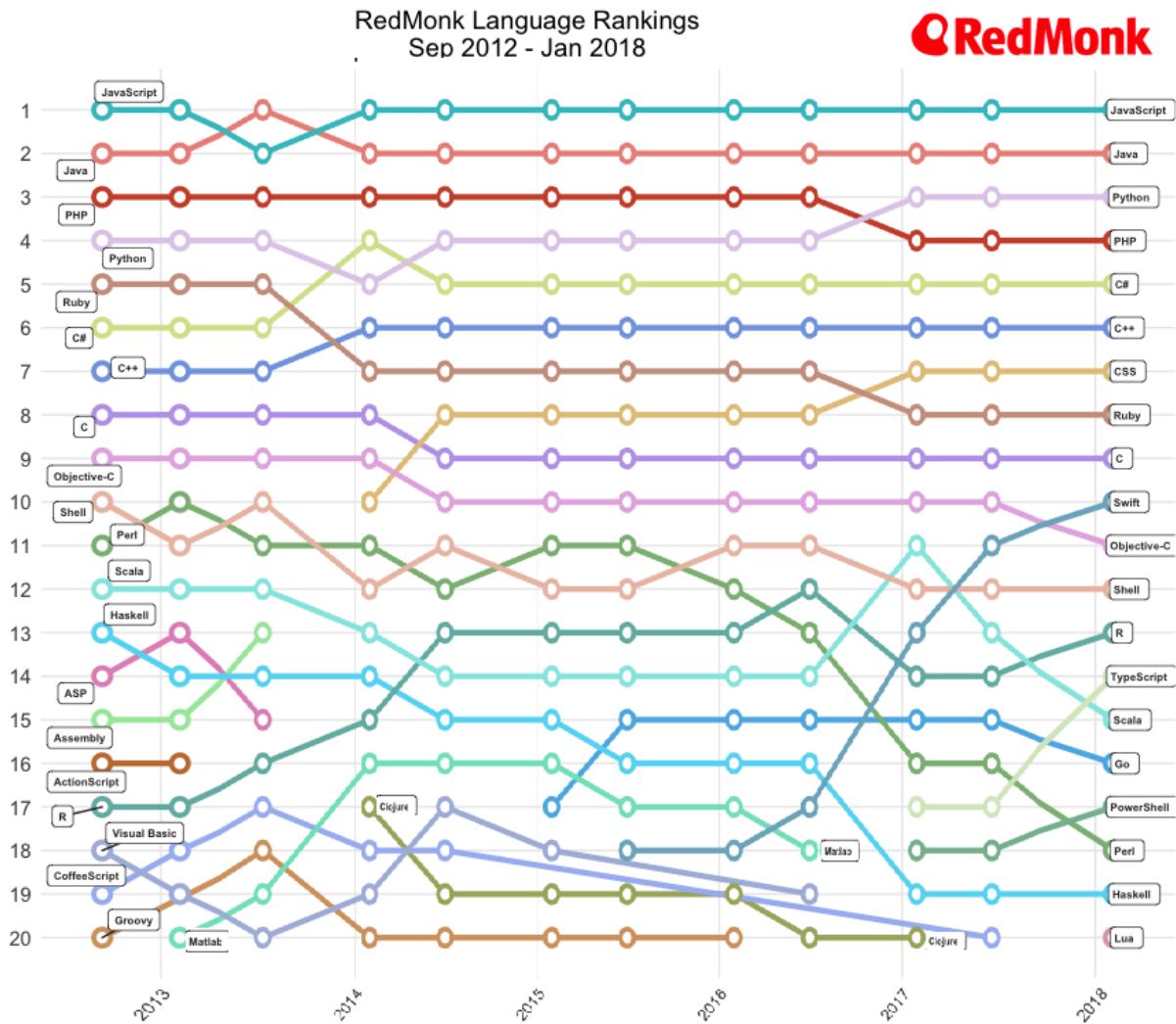
Fonte: Adaptado de Subramanian (2017)

2.2 STACKS DE DESENVOLVIMENTO

Para se construir uma aplicação, necessitamos de várias tecnologias. O conjunto dessas tecnologias utilizadas para criar aplicações são denominadas *stack* conforme explica Subramanian (2017), esse termo ficou popularizado pela *stack Linux, Apache, MySQL, PHP* (LAMP), porém existem muitas outras combinações que formam *stacks* diferentes, cada uma com suas linguagens e tecnologias.

Ao observar a Figura 12 é visto um ranking de linguagens de programação desenvolvido pela equipe de análise e consultoria da RedMonk. Nota-se que ocorre muitas variações na popularidade e utilização das linguagens, entretanto na última década, as 3 linguagens que mais permaneceram no topo do ranking foram JavaScript, Java e PHP, e será abordado sobre algumas *stacks* dessas linguagens nas próximas seções.

Figura 12 – Ranking de linguagens.



Fonte: <https://redmonk.com/fryan/2018/03/15/redmonk-language-rankings-over-time/>

2.2.1 Stack PHP

Segundo Karanjit (2016) a *stack* LAMP é formada por uma série de tecnologias de código aberto, como o sistema operacional Linux, servidor web Apache, Banco de dados MySQL, e linguagem de *script* PHP. Esta é uma *stack* muito antiga, mas continua sendo uma das preferidas por muitas organizações, primeiramente por ser uma *stack* gratuita e de código aberto, e também por conta disso, facilita a integrações e fornece acesso total ao servidor, incluindo acesso remoto, facilitando a realização de tarefas administrativas de qualquer lugar.

Conforme explica Karanjit (2016), Linux é um sistema operacional que faz

parte dessa *stack* PHP. Assim como outros sistemas operacionais que fazem parte de variações dessa *stack* como o Windows (*software* proprietário e não gratuito) que está incluso na *stack* *Windows, Apache, MySQL, PHP* (WAMP), executam os aplicativos e operações para acessar o *hardware*. São classificados por sua velocidade, requisitos de hardware, segurança e controle remoto.

Outra tecnologia integrante dessa stack, é o servidor *web* Apache. Desenvolvido pela *Apache Software Foundation* (ASF), roda em Linux e Windows, sendo que no Windows ele sofre por perda de performance devido ao gerenciamento de memória da Microsoft e também devido a algumas diferenças arquiteturais, por esse motivo é preferível utilizá-lo com Linux. Algumas das características do Apache é que ele permite criar *hosts* virtuais, possibilitando rodar múltiplos *web sites* em um único servidor, permitindo acesso via *web*, protegendo os acessos aos diretórios e suporte a *Secure Sockets Layer* (SSL) e *gateways comuns*, conforme explica Karanjit (2016).

Segundo Karanjit (2016) O MySQL é mais uma tecnologia que compõe a *stack* LAMP. Responsável pela persistência das informações, o MySQL é um *Relacional Database Management System* (RDBMS) criado pelo pesquisador da *International Business Machines Corporation* (IBM) Edgar Frank Codd. Algumas das características desse sistema de banco de dados são:

- Os avançados relacionamentos de dados organizados em tabelas;
- Sua velocidade para computar esses relacionamentos;
- Facilidade para projetar e desenvolver utilizando essa tecnologia;
- Aceita diferentes tipos de dados como booleanos, texto, inteiros, imagens, binários, objetos, data etc;
- Permite a criação, recuperação, ou alteração de dados através de linguagens de *script* como o PHP.

Vale ressaltar também que muitos que utilizam a *stack* LAMP, utilizam o MariaDB como alternativa ao MySQL. Porém ambas tecnologias funcionam por meio da SQL.

Por último temos o PHP como integrante da *stack* LAMP. PHP é uma linguagem de script *server-side* criado por Rasmus Lerdorf em 1994, para construção de *web sites* ou aplicações web dinâmicas que rodam em um servidor. PHP também pode ser “embutido” em códigos HTML para tornar a parte visual da aplicação mais dinâmica, através de ações do usuário por meio de métodos HTTP como GET, POST

que são enviados pela submissão de formulários por exemplo, permitindo a execução de alguma função no servidor, e trazendo a resposta ao cliente. O PHP é muito amigável no sentido de ser fácil de aprender e mescla-lo com HTML e CSS, basta aplicar a lógica para criar aplicações de vários níveis segundo Karanjit (2016). Porém essa mistura do código PHP junto com a linguagem de estruturação, pode deixar o código confuso e desorganizado dependendo do tamanho da aplicação, o que dificulta a manutenção.

Algumas pessoas que utilizam a *stack* LAMP, optam por utilizar a linguagem Perl no lugar de PHP, entretanto isso é mais raro, e o Perl está deixando de ser utilizado conforme pode-se observar na Figura 12.

2.2.2 Stack Java

Java como é uma das linguagens de programação mais utilizadas, possui uma variedade de ferramentas desenvolvidas para esta linguagem. A combinação dessas ferramentas forma a *stack* Java conforme explica Heredia e Sailema (2018). Existem muitas combinações possíveis, mas para citar um exemplo de uma das *stacks* Java mais utilizadas para o desenvolvimento de aplicações modernas é a *stack* Java *Enterprise Edition* (Java EE).

Heredia e Sailema (2018) diz que as tecnologias envolvidas na *stack* trabalham de forma fácil e conjunta. Separando a camada de visualização das demais camadas temos como possibilidade utilizar as seguintes tecnologias:

- *Java Server Pages* (JSP), HTML, CSS no *front-end*, onde o html é responsável pela marcação dos elementos na página, o CSS será responsável pela estilização, e o JSP oferece fácil acesso aos dados, permitindo a separação de níveis dentro da aplicação, além de herdar a portabilidade do Java;
- AngularJs é outra opção para o *front-end* da aplicação. Ele é uma biblioteca JavaScript para desenvolvimento de aplicações *web*, mantida pela Google. É um framework JavaScript que trabalha com os conceitos de *Single Page Application* (SPA). Esse framework segue o padrão MVC, deixando a aplicação escalável, manutenível e padronizada.
- O framework Spring Boot pode ser utilizado no *back-end*, que possui componentes que simplificam as etapas de seleção das dependências de um projeto e sua implementação, deixando o desenvolvedor dedicar seu foco na aplicação em si;
- Postgresql, MongoDB, SQL ou outra base de dados pode ser utilizada para a persistência dos dados;

- Apache/Tomcat no servidor que rodará a aplicação e demais tecnologias envolvidas no desenvolvimento;
- Demais tecnologias envolvidas do desenvolvimento *fullstack* com Java.

Importante ressaltar que segundo estudos feitos por Heredia e Sailema (2018), uma *stack* Java é recomendada para uma aplicação pesada que requer muito processamento computacional, pois as *stack* Java tem capacidade para tal. Entretanto se a aplicação também exige uma alta interação do usuário como uma SPA, alta visualização e alta escalabilidade, a melhor alternativa é uma *stack* mais interativa como as *stacks* JavaScript.

2.2.3 Stack JavaScript

Criada em 1995 por Brendan Eich da Netscape, JavaScript foi de uma linguagem de brinquedo utilizada apenas em *browsers* a uma das linguagens mais populares do mundo para desenvolvimento *full-stack* (KLAUZINSKI; MOORE, 2016). Quando se fala em *stack* JavaScript, existem uma série de variações possíveis, algumas delas são:

- *MongoDB, Express, React, NodeJS* (MERN);
- *Sequelize, Express, React, NodeJS* (SERN);
- *MongoDB, Express, AngularJS, NodeJS* (MEAN);
- *Sequelize, Express, AngularJS, NodeJS* (SEAN);
- *MongoDB, Express, VueJS, NodeJS* (MEVN);
- *Sequelize, Express, VueJS, NodeJS* (SEVN);

Existem muitas outras variações de *stacks* JavaScript, assim como existem variações de *stacks* PHP e Java. Porém algumas delas se destacam por ser uma *stack* totalmente em JavaScript, como é o caso da *stack* MERN por exemplo. Conforme explica Klauzinski e Moore (2016), com a introdução do NodeJS que foi construído em cima do motor V8 do Chrome, tornou possível que desenvolvedores construissem poderosas e performáticas aplicações usando JavaScript. Somando isso com a utilização do MongoDB que é um moderno banco de dados *Not Only Standard Query Language* (NoSQL), pode-se utilizar JavaScript em todas as camadas da aplicação.

Aryal (2020) descreve o desenvolvimento *fullstack* em JavaScript dividido em *front-end*, *back-end* e base de dados. Como opção para o front-end, Aryal (2020) menciona o ReactJS, uma das bibliotecas JavaScript mais utilizadas, de código aberto,

declarativa, eficiente, e baseada em componentes. O React foi desenvolvido e é constantemente atualizado pelos engenheiros do facebook e usuários pelo mundo.

Os componentes do React são pequenos e reusáveis. Eles retornam um elemento do React para ser renderizado na página, e faz isso usando um JavaScript normal, que obtém *inputs* ou *props* (propriedades). Além disso os componentes podem retornar arrays, strings, números, e também outros componentes, criando assim uma árvore de componentes. Os componentes podem ter estados, e passar esses estados como propriedades para os componentes filhos conforme explica Aryal (2020). Os componentes de classe podem ter métodos de ciclo de vida que podem fazer alterações na interface do usuário ou nos dados. Além disso o autor também explica que com o React Hooks podemos reaproveitar também a lógica em outros componentes de forma fácil, utilizando *states* e métodos de ciclo de vida. Vale mencionar também que o React trabalha com o conceito de *two-way data binding*, ou seja, tudo que acontece no *Document Object Model* (DOM), se reflete no *model* JavaScript, dessa forma, se o usuário realizar alguma ação na interface gráfica, isso implicará também na alteração dos dados no JavaScript, e se por alguma ação do sistema houver a mudança nos dados no JavaScript, haverá também a mudança na interface.

Já Aggarwal e Verma (2018) propõe além do uso do React como tecnologia para o *front-end*, o uso do *framework* AngularJS, que é mantido pela Google e pela comunidade, muito utilizado para construção de SPAs, também trabalha com componentes, com *two-way data binding*, facilmente trabalha com o padrão REST e *Model, View, Controller* (MVC).

Além desses frameworks existem outros frameworks e bibliotecas que trabalham com esses conceitos e podem ser utilizados para construção do *front-end* como por exemplo o VueJS que também é muito conhecido.

O *back-end*, o NodeJS é uma plataforma proposta como uma das principais tecnologias para o *back-end* da aplicação conforme descrevem Keinänen (2018) e Aryal (2020). Keinänen (2018) ainda faz uma comparação dizendo que o node pode ser caracterizado como pertencente ao JavaScript assim como *Java Runtime Environment* (JRE) é para Java.

Keinänen (2018) e Aryal (2020) destacam a facilidade de desenvolvimento com node e seu desempenho, pelo fato de ser construído sobre o motor v8 do Chrome. E também enfatizam que a chegada dessa tecnologia possibilitou rodar código JavaScript fora dos *browsers*, onde o node não é apenas usado para aplicações web, mas para desenvolvimento cross-plataforma, *server-side*, e aplicações de rede.

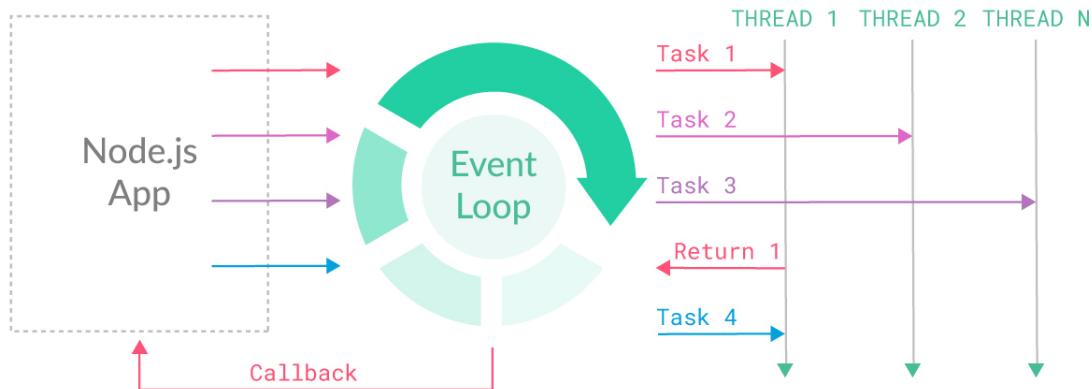
Segundo Keinänen (2018) o node é uma viável alternativa para outras linguagens *server-side* como PHP, ASP ou Java, e o mais importante, permite usar Ja-

vaScript em cada camada da *stack*, seja no servidor, no formato do transporte com *JavaScript Object Notation* (JSON) ou no cliente.

O NodeJS em pouco tempo já ganhou uma boa reputação na indústria de tecnologia. Ele está presente por exemplo no Microsoft Azure, eBay, LinkedIn, PayPal que trocou o Java por Node entre outras empresas conforme comenta Aryal (2020). Netflix, Walmart, Trello, Uber, Medium e Nasa também são citados por Keinänen (2018) por utilizarem o NodeJS para suas aplicações devido sua velocidade, e alta performance.

Segundo Aryal (2020) uma das grandes vantagens do NodeJS, é sua capacidade de executar o código assincronicamente. Ou seja, não precisa esperar uma tarefa terminar para executar a seguinte. Segundo Keinänen (2018) isso é possível por que o NodeJS trabalha com o seu *event-loop* tirando vantagem do sistema *multi-thread* do *kernel*, mantendo o *Input/Output* (I/O) não bloqueado. Aryal (2020) também explica que diferente de linguagens como C ou Java que trabalham de forma síncrona, e por isso precisam esperar uma tarefa terminar para executar a próxima. Com o NodeJS as tarefas podem ser executadas em paralelo umas as outras, e esse é um dos motivos da velocidade dessa tecnologia. Na Figura 13 pode-se observar o funcionamento do *event loop* do NodeJS, como é distribuída as tarefas (*Tasks*) nas *Threads* do sistema, bem como o retorno da tarefa para o event loop que envia os dados retornados para a aplicação através do *Callback* da função requisitante.

Figura 13 – Funcionamento do *event loop*.



Fonte: <https://nexocode.com/blog/posts/behind-nodejs-event-loop/>

Esse assincronismo permitido pelo NodeJS apenas de trazer a vantagem da velocidade, algumas vezes não é desejado, pois caso uma tarefa dependa da execução de uma outra, isso pode causar algum erro no programa. Para contornar essa

situação, quando sabemos que determinada tarefa pode demorar e ela precisa ser síncrona, podemos utilizar alguns artifícios para dizer que o código deve ser executado sincronicamente. Keinänen (2018) menciona como solução o uso de *callbacks* que por padrão retorna dados após a execução da primeira tarefa, para então executar ou não a tarefa subsequente.

Figura 14 – Funções aninhadas formando um "Callback Hell".

```

1 // Callback Hell
2
3
4 a(function (resultsFromA) {
5     b(resultsFromA, function (resultsFromB) {
6         c(resultsFromB, function (resultsFromC) {
7             d(resultsFromC, function (resultsFromD) {
8                 e(resultsFromD, function (resultsFromE) {
9                     f(resultsFromE, function (resultsFromF) {
10                         console.log(resultsFromF);
11                     })
12                 })
13             })
14         })
15     })
16 });
17

```

Fonte: <https://medium.com/@jaybhoyar1997/avoiding-callback-hell-in-node-js-7c1c16ebd4d3>

Entretanto isso pode eventualmente ocasionar um outro problema conforme menciona DUARTE (2020). Esse problema é conhecido como *Callback Hell*, que nada mais é do que o aninhamento de várias funções *Callback* como observado na Figura 14, deixando o código complexo e de difícil manutenção.

DUARTE (2020) explica que esse problema foi resolvido em 2015 com a chegada do *ECMAScript 6* (ES6) o qual introduziu *promises*(promessas), que permite dizer ao JavaScript que determinado código deve ser executado de forma síncrona. Em 2017 entrou também o conceito de *Async/Await* para facilitar ainda mais o uso de sincronismo em funções assíncronas. Na Figura 15 podemos observar uma função assíncrona por natureza da tecnologia, mas que podemos torná-la síncrona com o uso da sintaxe *Async/Await*.

Figura 15 – Uso de *Async/Await* para tornar funções síncronas.



```

async function eatAllSnacks() {
  let snacks;

  try {
    snacks = await getAllSnacks();
  } catch {
    console.log("Oops, you won't get any snacks 😞");
  }

  if (snacks) {
    eat(snacks);
  }
}

```

Fonte: <https://itnext.io/async-await-without-try-catch-in-javascript-6dcdf705f8b1>

Apesar do NodeJS ser a plataforma do seguimento de desenvolvimento *server-side* JavaScript mais popular e robusto, existem algumas outras alternativas a essa tecnologia, como o Vertx e o Deno.

Encontrar bibliotecas e pacotes para a plataforma NodeJS e JavaScript em geral é muito fácil devido ao gerenciador de pacotes do node, o *Node Package Manager* (NPM) conforme menciona Keinänen (2018) a respeito desse gerenciador, o qual é atualmente o maior registrador de software do mundo, ostentando 6000 000 pacotes de código e 3 bilhões de downloads por semana. NPM é um *website* que permite realizar pesquisas e encontrar pacotes que podem servir para suprir alguma necessidade da aplicação. A instalação desse pacote é feita através dos comandos NPM via terminal, facilitando a instalação, desinstalação e atualização de pacotes. Além disso, o NPM é de código aberto e já vem com o NodeJS. Existe também o Yarn desenvolvido pelo Facebook como alternativa ao NPM, onde muitos até o preferem por conta de sua velocidade e pelo fato de poder utilizar os mesmos pacotes com ambos os gerenciadores.

Aryal (2020) menciona também o ExpressJS como parte do *back-end* da aplicação *fullstack* JavaScript, onde o ExpressJS é um framework que trabalha como um *middleware* entre os *requests* e *responses*, e com ele também é possível criar nossos próprios *middlewares*, onde cada *middleware* tem acesso aos dados de *request* e *re-*

sonse, que vai passando por cada *middleware*. O *middleware* pega o *request* executa o código dentro dele, realiza as modificações necessárias no *request* e chama a próxima função declarada na rota, a qual aciona o próximo *middleware*. As rotas também ficam a encargo do ExpressJS, o qual trabalha de forma muito simples e intuitiva, podendo inclusive criar rotas estáticas, para acesso a arquivos pelo endereço literal do servidor.

As rotas no ExpressJS são compostas pela chamada do método HTTP, passando como parâmetro na assinatura do método, o endereço da rota, as funções e *middleware*, todos separados por vírgula conforme exemplifica Aryal (2020). Vale lembrar que assim como as demais tecnologias mencionadas também existem alternativas ao ExpressJS, como por exemplo o Koa e o Hapi. A Figura 16 mostra exemplos de rotas escritas com ExpressJS.

Figura 16 – Exemplos de rotas criadas com ExpressJS.

```
// responde com "Hello World!" na página principal
app.get('/', function (req, res) {
  res.send('Hello World!');
});

// aceita uma request POST na página principal
app.post('/', function (req, res) {
  res.send('Uma requisição POST');
});

// aceita uma request PUT na url /usuario
app.put('/usuario/', function (req, res) {
  res.send('Uma requisição PUT na url /usuario');
});

// aceita uma request DELETE na url /usuario
app.delete('/usuario/', function (req, res) {
  res.send('Uma requisição DELETE na url /usuario');
});
```

Fonte: <https://itnext.io/async-await-without-try-catch-in-javascript-6dcdf705f8b1>

O banco de dados também compõe uma aplicação, e quando se fala em aplicações *fullstack* em JavaScript, Aryal (2020) traz como opção o MongoDB, também conhecido como banco de dados NoSQL, que é um banco de dados de código aberto e cross-plataforma, desenhado para se encaixar em aplicações modernas. E como o MongoDB utiliza JSON como esquema de arquivos, significa que os dados são armazenados em documentos ou coleções, podendo ser *strings*, números, decimais, e até *arrays* de objetos. Ele permite a inserção de dados sem interrupção do serviço, e prove mudanças em tempo real. Entretanto existem algumas observações referente ao Mon-

goDB. Primeiro ele tem um limite de 16 MB por arquivo, e segundo não é permitido um documento com mais de 100 níveis.

Outra observação importante, é que se a aplicação tiver muitos relacionamentos, talvez um banco NoSQL não seja o ideal, apesar do MongoDB conseguir se relacionar através de subdocumentos, quando se tem muitos relacionamentos, isso pode afetar o desempenho.

Aryal (2020) também fala sobre o Mongoose como uma forma de facilitar o uso do MongoDB. Mongoose é uma biblioteca para interagir com o banco de dados MongoDB, permitindo criar, ler, atualizar e deletar dados do banco utilizando uma sintaxe escrita em JavaScript para criar as *querys*, provendo padrões de *schemas*, trabalhando com validações de dados, e garantindo o padrão da estrutura dos dados armazenados.

Existem também outras opções de banco de dados para se trabalhar com JavaScript, como o próprio *Structured Query Language Manager* (MySQL), e utilizando a biblioteca Sequelize para criar as *querys* com a sintaxe em JavaScript. Porém a escolha depende de cada aplicação.

2.3 SOLUÇÕES DE DESENVOLVIMENTO

Seção destinada a tratar sobre algumas soluções e tecnologias de desenvolvimento de software.

2.3.1 Versionamento de código

Oliveira (2018) explica o versionamento de código fazendo um comparativo com a construção de um texto, um livro ou trabalhos acadêmicos, é criada várias versões, incluindo e modificando partes do texto. Geralmente é feito um *backup* da versão antiga e criado um novo documento para fazer as modificações, assim, caso aconteça algum problema, é possível recuperar o texto anterior.

Essa ideia também se aplica ao versionamento de código-fonte. Quando o código é criado, tem-se então a primeira versão. Ao alterar algo no código, é gerado uma segunda versão, e assim por diante. Para automatizar os processos de versionamento de código existem alguns softwares que facilitam esse trabalho, criando versões do programa que está sendo desenvolvido para que não seja preciso ficar criando cópias do nosso projeto manualmente, conforme explica Oliveira (2018).

Palestino (2015) realizou um estudo comparativo sobre as tecnologias de versionamento de código e dentre as várias tecnologias estudadas chegou à conclusão que o *Apache Subversion* (SVN) e o GIT (Juntamente com o GITHUB), são as tec-

nologias mais utilizadas para controle de versão de código, tendo como preferência majoritária o GIT/GITHUB, devido a simples instalação e o fato de não necessitar de um servidor, possibilitando acessar de qualquer lugar.

Conforme explica Palestino (2015), o GIT é um dos softwares existentes para controle de versão de código-fonte, que oferece a possibilidade de se trabalhar sempre em um mesmo diretório, fazendo alterações no projeto, gravar documentação e comentários, sendo tudo registrado pelo programa. Também é possível se trabalhar com o código-fonte em uma área separada, para finalidade de testes por exemplo, assim como também é possível criar quantos repositórios for necessário para quantos projetos desejar. Esse registro de todas as modificações de um repositório, permite que possa ser desfeito qualquer alteração versionada para a última versão estável, conforme explica Palestino (2015).

Segundo Palestino (2015), o GIT também torna possível o trabalho em equipe, permitindo que um arquivo seja editado por várias pessoas, sem o risco de informações serem sobreescritas, pois caso haja algum conflito, quando for tentar registrar as alterações será avisado a respeito desses conflitos, solicitando suas resoluções para depois registrar as alterações.

Um dos diferenciais do GIT sobre os demais softwares de versionamento, é que ele não armazena as alterações em uma lista de mudanças por arquivo, pois considera que todos os arquivos fazem parte de um minisistema. Então para que fique registrado determinada versão, ele cria *snapshots* ou *branchs* que são uma espécie de cópia espelho de um determinado momento do projeto, segundo Palestino (2015), que também faz uma suposição de um projeto que precise ser feita uma modificação, mas essa modificação não deve ficar disponível para mais ninguém além do desenvolvedor, até que tudo esteja pronto. Neste caso Palestino (2015) diz que *snapshot* é uma boa alternativa, pois permitirá que o desenvolvedor trabalhe nesse espelho, e quando tudo estiver pronto, basta realizar um *merge* para o projeto original. Com isso, pode-se dizer que cada vez que alguém salva uma alteração em um projeto, ação essa que é conhecida pelo nome de *commit*, o GIT estará fazendo uma foto de todos os arquivos que compõem o repositório para criar essa cópia espelho, a qual é considerada a versão propriamente.

Você pode manter o repositório principal localmente no computador, criando uma espécie de servidor para o GIT, mas também há disponível o GitHub, que é um local para armazenar os arquivos versionados via GIT na nuvem. Assim você tem acesso ao repositório em qualquer lugar. Para projetos pessoais pode-se utilizar o GitHub gratuitamente, e também quase todos os projetos, frameworks e bibliotecas *open source* estão armazenados no GitHub segundo Palestino (2015).

O funcionamento do GIT é muito simples, conforme explica Palestino (2015). Seu *workflow* básico se resume no seguinte:

- Modificação, inclusão ou exclusão de arquivos no repositório de trabalho;
- Seleção dos arquivos a serem versionados, adicionando seus *snapshots* para a área de preparação. Pode ser utilizado o comando *add* para realizar esta ação;
- Colocar os arquivos da área de preparação para envio com o comando *commit* e adicionando um comentário/descrição da versão;
- Enviar os arquivos para o repositório central (GitHub por exemplo), utilizando o comando *push*.

2.3.2 Gerenciamento de pacotes

Segundo Jacobs (2019), o objetivo de um gerenciador de pacotes é lidar com instalação, dependências, recuperação e atualização de pacotes de software. Gerenciadores de pacotes independentes da linguagem de programação, são desenvolvidos para resolver o mesmo problema. Ao invés de procurar bibliotecas de código e qualquer outra dependência, o gerenciador de pacotes fará isso.

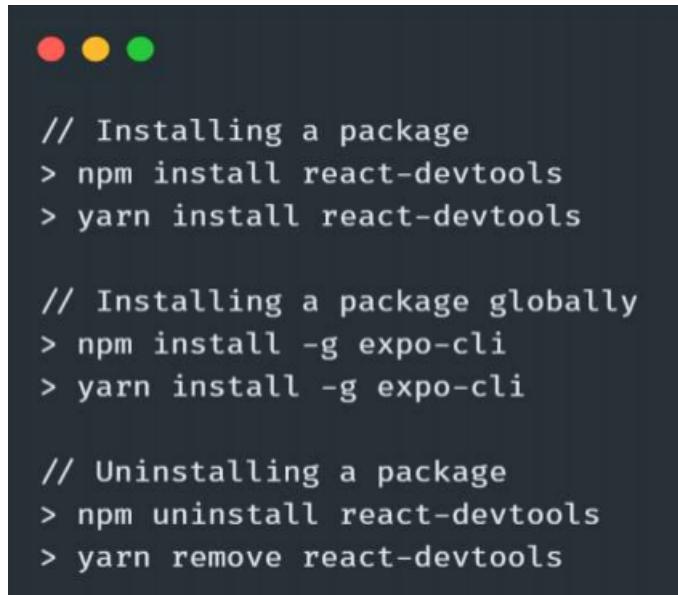
Quando falamos de JavaScript, os gerenciadores de pacotes modernos desta linguagem, são o NPM e Yarn. A primeira *release* do NPM foi distribuída como parte do NodeJS, conforme Jacobs (2019), que também menciona o NPM como sendo o gerenciador de pacotes JavaScript dominante no mercado. Atualmente o NPM é mantido pela NPM Inc.

Segundo Jacobs (2019), o registro NPM é o serviço que hospeda mais pacotes. Possuindo o dobro de pacotes que o segundo colocado, o Apache Maven para pacotes Java.

Jacobs (2019) menciona o Yarn como o segundo gerenciador de pacotes mais popular do mercado, o qual foi desenvolvido por um time do Facebook para resolver alguns problemas do NPM, como performance e segurança. Por este motivo, a velocidade de *download* de pacotes realizados através do Yarn é mais rápido se comparado com o NPM, mesmo o Yarn utilizando o registro NPM por padrão.

Em geral, os comandos tanto do Yarn quanto do NPM são bem parecidos. A Figura 17 demonstra alguns exemplos comparativos de comandos realizados do Yarn e NPM. Pode-se encontrar todas as funcionalidades de ambos gerenciadores, em suas respectivas documentações.

Figura 17 – Comparativos entre comando Yarn e NPM



```

// Installing a package
> npm install react-devtools
> yarn install react-devtools

// Installing a package globally
> npm install -g expo-cli
> yarn install -g expo-cli

// Uninstalling a package
> npm uninstall react-devtools
> yarn remove react-devtools

```

Fonte: (JACOBS, 2019)

Ambos os gerenciadores Yarn e NPM, quando instalam, alteram, excluem ou atualizam algum pacote, registram a modificação em um arquivo chamado *package.json*, o qual fica tem como um dos propósitos justamente armazenar essa informação de versões dos pacotes instalados, facilitando seu gerenciamento, pois de forma automatizada é salva a lista de dependências, conforme explica Jacobs (2019).

É valido mencionar também que existem outros gerenciadores de pacotes em destinados a outras linguagens, como o Composer para *Hypertext Processor* (PHP) e Maven ou Gradle para projetos Java.

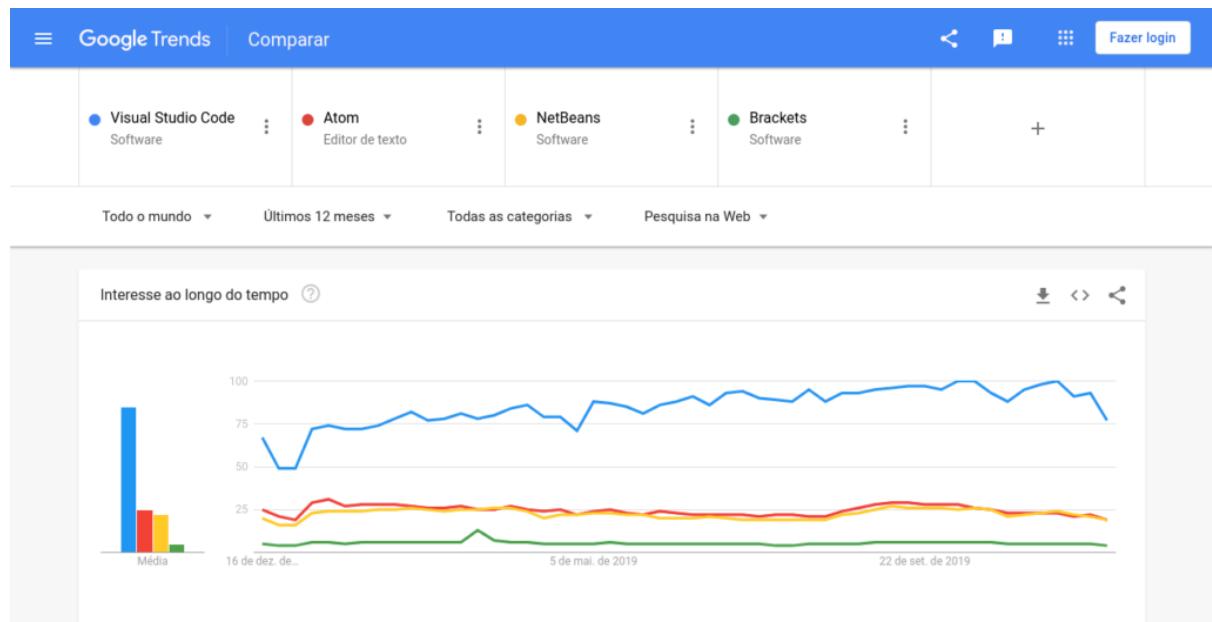
Conforme Armel (2014) o Composer também tem como objetivo gerenciar dependências, desta vez para projetos em PHP. Assim como os gerenciadores de pacotes já mencionados o Composer permite atualizar pacotes de dependências do projeto com um só comando, instalar e atualizar dependências individuais. Ele consegue fazer esse gerenciamento, por também salva em um arquivo chamado *autoload* na raiz do projeto as informações das dependências instaladas. O Composer é instalado como um executável PHP e é adicionado nas variáveis de ambiente. Quando o Composer for devidamente instalado, o desenvolvedor conseguirá utilizá-lo de qualquer lugar do sistema de arquivos utilizando o comando *composer*. As mesmas funcionalidades podem ser alcançadas em projetos Java utilizando algum gerenciador como o Gradle por exemplo.

2.3.3 IDE e editores

Segundo Chaves e Silva (2008), um IDE, é um conjunto de ferramentas que apoiam o desenvolvimento de software com o intuito de agilizar o processo, aproveitando melhor o tempo dos desenvolvedores, proporcionando rapidez e facilidade, e envolve pelo menos 3 ferramentas, o editor, o compilador e o depurador.

Alguns dos editores de texto destinados ao desenvolvimento de software, traz consigo ferramentas de compilação ou depuração, dentre outras ferramentas, ou traz a possibilidade de adicionarmos extensões ou dependências no projeto que farão algum papel específico no desenvolvimento, transformando assim o editor em uma IDE. Conforme pesquisa realizada por FERNANDES (2019) referente aos melhores IDE para 2020, foi listado 4 das mais utilizadas e populares IDE para desenvolvimento multiplataforma, onde o autor não quis considerar IDE específicas para uma plataforma, nem software pagos. As IDE deste estudo são o VS code, Atom, NetBeans e o Brackets. Na Figura 18 pode-se observar a popularidade de cada uma dessas IDE no Google Trends, e nota-se que o VS code ganha no quesito de popularidade.

Figura 18 – Comparativos Google Trends entre IDE



Fonte: (FERNANDES, 2019)

FERNANDES (2019) também cita a importância da participação de cada uma dessas IDE no GitHub, como as estrelas, *issues*, data do último *commit* etc, para descobrir se o projeto está sendo saudável e sendo atualizado. A Tabela 6 mostra os

números dos repositórios do GitHub de cada um dos projetos dessas IDE. Pode-se observar que o VS Code também está a frente quanto aos números do GitHub.

Tabela 6 – Números dos repositórios do Git Hub dos projetos de IDE

Programa	Google Trends	Git Hub stars	Git Issues
VS Code	85	88.3 K	4,043
Atom	25	50.6 K	454
NetBeans	22	1.2 K	—
Brackets	9	30.6 K	2,441

Fonte: Adaptado de FERNANDES (2019)

Segundo FERNANDES (2019), o Visual Studio é a IDE mais popular atualmente. Mantida pela Microsoft e desenvolvida pela comunidade, e por ter sido desenvolvido em outro projeto de código aberto (O Eléctron), o VS Code é compatível com Windows, MacOS e Linux, possuindo suporte para quase todas as linguagens, e por ser altamente customizável através de extensões e *plugins*, permite criar o ambiente de desenvolvimento ideal, seja qual for a linguagem utilizada.

2.3.4 Ferramentas de análise estática de código

Segundo Terra e Bigonha (2008) e Rheinboldt e Chaves (2018), as ferramentas de análise estática de código, são software que vasculham por scripts ou trechos de códigos específicos para encontrar erros, sintaxe mal formada e formatação padronizada.

Terra e Bigonha (2008) diz ainda que o conceito de análise estática de código não é recente. Em 1970 Stephen Johnson da Bell Laboratories criou uma ferramenta chamada de *Lint* que examina código fonte de programas escritos na linguagem C, para verificar erros que pudesse ter escapado do compilador. O sucesso de *Lint* gerou outras ferramentas mais modernas que os desenvolvedores atuais utilizam nas mais diversas linguagens.

Alguns exemplos desse tipo de ferramenta são o PhpLint para a linguagem PHP, Jlint para Java, esLint para JavaScript, Rubocop para Ruby, dentre muitas outras ferramentas.

Dentre os muitos analisadores de código, o EsLint merece um destaque especial, pois conforme menciona Rheinboldt e Chaves (2018), ele é um analisador de código JavaScript, e por conta do ecossistema JavaScript ser muito diversificado, e muitas empresas ditarem boas práticas de escrita de código para esta linguagem, como a Google, e a AirBnB, e também diversos frameworks que seguem determinados padrões, o EsLint então procurou se adaptar a esses diversos padrões, permitindo que seja escolhido o padrão que será executado pelo analisador.

2.3.5 *Templates Engines*

Segundo ANDRADE (2019), um *template engine* é um tipo de ferramenta que auxilia o escape de informações nas interfaces do software. Uma listagem de dados que é resultado de uma consulta no banco de dados, pode ser mesclado essas informações juntamente com o código HTML de um site por exemplo. Com a linguagem PHP que permite essa mesclagem de HTML dentro de scripts PHP ainda assim se tornaria algo maçante dependendo da quantidade de informações ou de repetições necessárias para exibir as informações, pois teria que ser utilizado a sintaxe de abertura e fechamento do script PHP para cada bloco HTML, e para cada instrução de *loop* ou condicional PHP. Já utilizando um *template engine*, a sintaxe das condicionais, *loops*, bem como o escape de informações se tornam bem mais simples e menos verbosa. Pode-se observar na Figura 19 o uso de comando da linguagem PHP sem o uso de *Template Engine*. Já na Figura 20 foi feito o mesmo bloco de código, mas desta vez utilizando a *template engine* padrão do Laravel, o Blade Template.

Figura 19 – Código sem uso de *Template Engine*

```
<?php if ($user->isLoggedIn()): ?>
Welcome back, <strong><?= $user->name; ?></strong>
<?php endif; ?>
```

Fonte: (ANDRADE, 2019)

Figura 20 – Código com uso do *Template Engine* padrão do Laravel (*Blade templates*).

```
@if ($user->isLogged())
    Welcome back, <strong>{{ $user->name }}</strong>
@endif
```

Fonte: (ANDRADE, 2019)

ANDRADE (2019) menciona como sendo as principais *template engines* das principais linguagens de programação as seguintes tecnologias:

- **PHP:** Plates, Blade e Twig;
- **JavaScript:** Mustache, Handlebars, doT, *Embedded JavaScript templates* (EJS), PUG, Jade Language, Squirrely;
- **Python:** Django Template, Genshi, Jinja, Mako;
- **Java:** JSP, ThimeLeaf, FreeMarker.

2.3.6 *Migrations*

Conforme explica BORSATO (2017), uma *migration* cria um controle do histórico de criação e alterações no banco de dados. Com esse histórico, é possível reverter as alterações realizadas quando necessário, criando assim um tipo de controle de versão do banco de dados.

Existem várias tecnologias que realizam *migrations* de banco de dados. Em geral, as *migrations* são realizadas através de comandos simples em um terminal. Em Laravel por exemplo, é utilizado por padrão para realizar *migrations* o comando *Artisan*. Pode ser observado um exemplo desse comando na Figura 21.

Figura 21 – Comando utilizado para criar uma *migration* em Laravel

```
1 | php artisan make:migration criar_tabela_alunos --create=aluno
```

Fonte: (BORSATO, 2017)

Conforme explica BORSATO (2017), uma classe é criada com uma estrutura pré escrita. Esta classe possuirá uma função para avançar a migração, e outra para reverter a migração. No caso das *migrations* do Laravel, as classes são *up* e *down* respectivamente, conforme pode-se observar na figura Figura 22.

Figura 22 – Exemplo de uma *migration* em Laravel

```

1 <?php
2
3 use Illuminate\Support\Facades\Schema;
4 use Illuminate\Database\Schema\Blueprint;
5 use Illuminate\Database\Migrations\Migration;
6
7 class CriarTabelaAlunos extends Migration
8 {
9     public function up()
10    {
11        Schema::create('aluno', function (Blueprint $table)
12        {
13            $table->increments('id');
14            $table->string('nome', 80);
15            $table->string('email', 80);
16            $table->integer('registro');
17            $table->timestamps();
18        });
19    }
20
21    public function down()
22    {
23        Schema::dropIfExists('aluno');
24    }
25 }
```

Fonte: (BORSATO, 2017)

Para executar essa migração, basta executar o comando *php artisan migrate*, que automaticamente as *migrations* criadas serão executadas. Para desfazer uma *migration*, basta executar o comando *php artisan migrate:rollback*, que a última *migration*

executada será desfeita, deixando o banco de dados ao estado anterior dessa migração, conforme explica BORSATO (2017).

Segundo BORSATO (2017), as migrações são quase inevitáveis durante o desenvolvimento de um *software*, mas com o uso das *migrations*, essa tarefa é feita com mais segurança e controle. Como visto, um dos benefícios principais é o fato de se poder ter e alterar versões distintas da mesma base de dados.

Existem outras alternativas, para outras linguagens e tecnologias para se realizar *migrations*, como por exemplo o Node-Migrations para migrar base de dados de sistemas desenvolvidos na plataforma NodeJS, e Flyway para Java, todos com o mesmo conceito básico.

2.3.7 Upload de arquivos

Conforme Lecheta (2015), para haver sucesso no *upload* de arquivos utilizando o método POST do HTTP, deve ser utilizado o *enctype multipart/form-data*, assim como observamos na Figura 23.

Figura 23 – Exemplo de formulário HTML para upload de arquivos.

```
<form enctype="multipart/form-data" action="[sua URL aqui]" method="POST">
    <!-- O tipo "file" cria o botão Browse para escolher o arquivo -->
    <input name="file" type="file" />
    <!-- Botão de submit -->
    <input type="submit" value="Enviar arquivo" />
</form>
```

Fonte: (LECHETA, 2015)

Essa estrutura HTML fará aparecer na tela um botão para adicionarmos arquivos ao formulário que será enviado via POST, conforme explica Lecheta (2015). Mas para que o servidor receba esses arquivos, deve ser preparado um *script* que receba a requisição, faça o devido tratamento, e salve os arquivos.

A maioria das linguagens tem suporte a *upload* de arquivos, porém quando fazemos *upload* de múltiplos arquivos, ou arquivos juntamente com outros campos de formulário, pode ficar um pouco difícil de trabalhar. Lecheta (2015) sugere o uso de uma biblioteca chamada Jersey para a linguagem Java. Essa biblioteca é possuí as

classes IOUtils e FileUtils e outros métodos, que permite manipular os arquivos com mais facilidade.

Em JavaScript também existem bibliotecas para facilitar requisições do tipo *multipart/form-data*. Conforme ALMEIDA (2017), a biblioteca Multer para NodeJS segue o mesmo princípio, trabalhando com as requisições *multipart/form-data*, de modo que se obtém os metadados de cada arquivo enviado, bem como outros campos adicionais dos formulários, e tratando os arquivos diretamente através da biblioteca do Multer, que trabalha como um *Middleware*, interceptando a requisição e manipulando os arquivos recebidos, enviando-os para seu pré-estabelecido repositório.

Pode-se observar a popularidade dessa biblioteca ao abrir a página do Multer na plataforma NPM, que exibe um total de 1.251.058 *downloads semanais*. A instalação e configuração do Multer pode ser feita também através dos comandos NPM, e está documentada também no site [npmjs.com](https://www.npmjs.com/package/multer), bem como sua documentação.

Existem também outras bibliotecas para diversas linguagens, que possui esse mesmo objetivo de facilitar o *upload* de arquivos via método HTTP.

2.3.8 Envio de e-mail

Para Sudana, Qudus e Prasetyo (2019), o e-mail é uma das mais comuns formas de comunicação hoje em dia. Não só para enviar mensagens de texto, mas também áudio, vídeo, e outros anexos. E-mail é um dos mais largos métodos de comunicação na internet, e seu tráfico tem crescido exponencialmente com o advento da *Word Wide Web*. Não importa qual linguagem e tecnologia usada para construir aplicações, provavelmente ela terá que enviar ou receber e-mail, seja para envio de email para troca de senha, boas vindas, relatórios, e-mail marketing etc. Sudana, Qudus e Prasetyo (2019) propõe o uso do PHPMailer, uma biblioteca PHP que facilita o uso de e-mail com esta linguagem. O PHPMailer consegue enviar e-mails automaticamente, sem utilizar outros usuários para enviar e-mail, e rodará em *Simple Mail Transfer Protocol* (SMTP), que é um protocolo de transporte de e-mails para enviar mensagens eletrônicas.

A instalação e utilização do PHPMailer é simples, e ambas estão disponíveis na página do PHPMailer do Git Hub no endereço <<https://github.com/PHPMailer/PHPMailer>>.

Assim como o PHPMailer para PHP, o NodeMailer para NodeJS é uma biblioteca muito fácil de utilizar. Conforme documentação disponível em <<https://nodemailer.com/about/>>, para instalar o NodeMailer basta digitar o comando NPM *npm install nodemailer* e já poderá utilizar em sua aplicação. Segundo CERON (2020) o NodeMailer também trabalha com o SMTP para transporte das mensagens, e sua configuração

para uso básico é similar ao PHP Mailer, basta criar um *script* de configuração, informando servidor, porta e dados de autenticação, e chamar um método para envio do e-mail com o conteúdo do e-mail que deseja enviar. Também é possível enviar anexos, imagens, e conteúdo multimídia nos e-mails.

Importante mencionar que existem muitas outras bibliotecas e tecnologias para o envio de e-mail para as linguagens mencionadas e outras linguagens, como por exemplo o Java Mail para a plataforma Java.

2.4 REQUISIÇÕES AJAX

Resumidamente Niederauer (2007) descreve o AJAX como sendo o uso sistemático de JavaScript e outras tecnologias para tornar o navegador mais interativo com o usuário, utilizando-se solicitações assíncronas. Ou seja, com AJAX é possível fazer solicitações ao servidor sem precisar recarregar a página do navegador, ao contrário do modelo tradicional que carrega a página para cada requisição realizada.

Silva () cita exemplos de uso bem conhecidos, como as pesquisas do Google e Gmail, onde o AJAX é utilizado para fazer requisições a cada caractere digitado, sem precisar recarregar a página. Isso acontece porque com o AJAX é possível buscar dados no servidor de forma assíncrona, sem precisar sair da página atual ou recarregá-la, e quando é recebido o retorno do servidor com os dados, apenas uma pequena área específica da tela é alterada dinamicamente conforme necessário.

Niederauer (2007) fala que existem muitas vantagens para se utilizar o AJAX nas aplicações, entretanto como única desvantagem apontada, ele menciona a possível incompatibilidade com determinadas tecnologias.

2.4.1 XHR

Silva (2009) explica que o objeto responsável pelo funcionamento do AJAX é o *XML HTTP REQUEST* (XHR). Criado pela Microsoft para ser utilizado no *browser* Internet Explorer utilizando ActiveX, foi posteriormente implementado em Java pela Mozilla e denominado XMLHttpRequest. A partir daí outros navegadores surgiram com suporte a tecnologia. Em 2006 a W3C criou especificações para tornar o objetivo XMLHttpRequest uma recomendação oficial e desde então essa tecnologia vem sendo bem sucedida.

O uso básico do objeto XMLHttpRequest se dá pela instância do objeto, abrir uma URL e enviar uma requisição, conforme exemplificado na Figura 24.

Figura 24 – Uso básico do objeto XMLHttpRequest

```
var oReq = new XMLHttpRequest();
oReq.open("get", sua_url, true);
oReq.send();
```

Fonte: Adaptado de
<https://developer.mozilla.org/pt-BR/docs/Web/API/XMLHttpRequest/Usando XMLHttpRequest>

2.4.2 Fetch API

Segundo MOURA (2020) o Fetch é uma API que facilita a requisições AJAX manipulando o objetivo XMLHttpRequest. Ela segue o padrão de *promisse*, ou seja, vai retornar uma promessa. Seu uso básico pode ser observado na Figura 25, onde é declarado uma constante com um endereço de URL, e é chamada a função nativa *fetch*, passando como primeiro parâmetro dessa função a constante que possui como valor a URL, como segundo parâmetro é passado um objeto JSON contendo informações como por exemplo o método HTTP da requisição. Como o Fetch API é baseado em promessa, sempre irá retornar algo, se tiver sucesso na requisição, irá trazer um objeto preenchido como resposta que poderá ser utilizado na função *then*, caso a requisição não tenha sucesso, automaticamente o retorno irá cair na função *catch*, onde poderá ser feito uma tratativa de erro.

Figura 25 – Uso básico do Fetch API

```
1 | const URL_TO_FETCH = 'https://braziljs.org/api/list/events';
2 | fetch(URL_TO_FETCH, {
3 |   method: 'get' // opcional
4 | })
5 | .then(function(response) {
6 |   // use a resposta
7 | })
8 | .catch(function(err) { console.error(err); });
```

Fonte: MOURA (2020)

2.4.3 Axios

Segundo Inácio et al. (2020), O Axios é uma biblioteca para realizar requisições por meio do protocolo HTTP. Silva et al. (2020) complementa dizendo as requisições feitas através dessa biblioteca retornam uma *promisse* compatível com a versão ES6 do JavaScript.

O Axios pode ser instalado com o comando NPM “*npm install axios*” conforme explica Alm (2019). E para realizar requisições com o Axios é bem mais simples, pois ele abstrai o objeto XHR. Basta chamar a função *\$axios*, uma subfunção que será o método HTTP escolhido, e como parâmetros deve ser passado o endereço da requisição, os campos da requisição, e escolher o que acontece com a *promisse* recebida após a requisição. Na Figura 26 pode-se observar esse funcionamento.

Figura 26 – Uso básico da biblioteca Axios

```
this.$axios.$post('myWebserver.com', {
  {
    email: "myEmail@hotmail.com",
    phoneNumber: "+358443005"

  })
  .then(response => {
    console.log("Customer Created");
  }
}
```

Fonte: Alm (2019)

2.5 METODOLOGIAS DE DESENVOLVIMENTO DE SOFTWARE

Segundo Koscienski e Soares (2007), várias metodologias existem para organizar o desenvolvimento de software. Essas metodologias são divididas em metodologias tradicionais, as quais dão ênfase na documentação de cada fase do desenvolvimento (como a metodologia em cascata), e metodologias ágeis, que são consideradas paradigmas do desenvolvimento de software, pois prometem melhorias na produção e qualidade do software. A metodologia de software é grupo de atividades que ajudam e direcionam o desenvolvimento do software.

2.5.1 Metodologia Ágil

Pressman (2010) diz que a metodologia ágil de software é a combinação de filosofias e princípios de desenvolvimento, sendo que as filosofias defendem a satisfação do cliente bem como a entrega incremental prévia. Já os princípios de desenvolvimento prezam por equipes pequenas e motivadas, métodos informais, artefatos de engenharia mínimos, e simplicidade no desenvolvimento em geral. Dessa forma pode-se atender a demanda moderna de software com mais agilidade, velocidade e corretos.

Sommerville (2011) explica que existe alguns métodos ágeis, cada um com suas características e princípios. Mas existem alguns princípios que são compartilhados entre eles. Pode-se observar a relação desses princípios compartilhados entre os métodos ágeis na Tabela 7.

Tabela 7 – Princípios compartilhados entre os métodos ágeis.

Princípios	Descrição
Envolvimento do cliente	Os clientes devem estar envolvidos no processo de desenvolvimento. Seu papel é fornecer e priorizar novos requisitos do sistema e avaliar suas iterações.
Entrega incremental	O software é desenvolvido em incrementos com o cliente, especificando os requisitos para serem incluídos em cada um.
Pessoas, não processos	As habilidades da equipe de desenvolvimento devem ser reconhecidas e exploradas. Membros da equipe devem desenvolver suas próprias maneiras de trabalhar, sem processos prescritos.
Aceitar as mudanças	Deve-se ter em mente que os requisitos do sistema vão mudar. Por isso projete o sistema de maneira a acomodar essas mudanças.
Manter a simplicidade	Focalize a simplicidade tanto do software a ser desenvolvido quanto do processo de desenvolvimento. Sempre que possível trabalhe ativamente para eliminar a complexidade do sistema.

Fonte: Adaptado de Sommerville (2011)

2.5.2 Scrum

Segundo Pressman (2010) o Scrum é um método de desenvolvimento ágil concebido por Jeff Sutherland em 1990. Os seus princípios são assim como o manifesto ágil, consistentes utilizados para orientar as atividades de desenvolvimento, sendo elas:

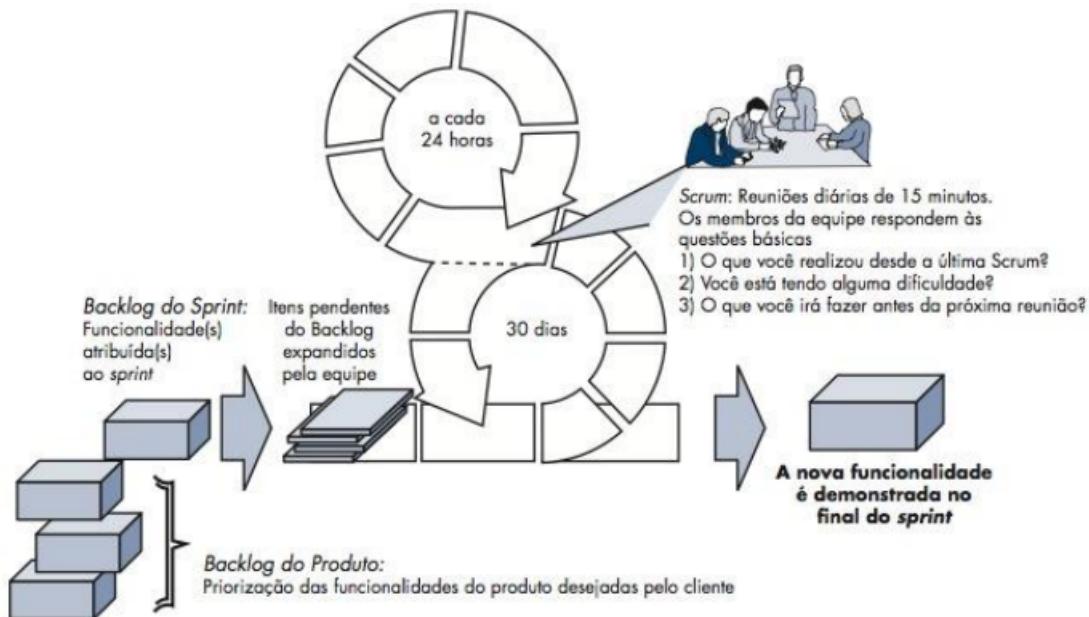
- Requisitos
- Análise
- Projeto
- Evolução
- Entrega

Pressman (2010) explica que em cada uma dessas tarefas, acontecem tarefas que são realizadas em um padrão de processo chamado *sprint*. O trabalho a ser realizado no intervalo dos *sprints* são adaptados conforme o problema em questão, podendo ser modificado em tempo real pela equipe Scrum. O Scrum utiliza um conjunto de padrões de processo de software se provaram eficazes em projetos com prazos apertados, quando os requisitos podem sofrer alterações ao longo do desenvolvimento e projetos críticos de negócio. Esses processos ações de desenvolvimento:

- Registro pendente de trabalhos (*Backlog*): Lista com prioridades dos requisitos e funcionalidades do projeto. Podem ser inseridos itens a essa lista a qualquer momento, pois é dessa forma que novas alterações são introduzidas.
- Urgências (*Sprints*): São unidades de trabalho para alcançar um requisito estabelecido no registro *backlog*, e precisa ser ajustado dentro de um prazo já fechado chamado de janela de tempo, geralmente 30 dias.
- Alterações dos itens do registro de trabalho (*Backlog work itens*): Não são introduzidas durante execução das *sprints*, permitindo assim que a equipe trabalhe em um ambiente estável.
- Reuniões Scrum: Reuniões curtas de geralmente 15 minutos realizadas diariamente pela equipe Scrum, onde são feitas três perguntas-chave que devem ser respondidas por todos os membros da equipe:
 - O que você realizou desde a última reunião da equipe?
 - Quais obstáculos está encontrando?
 - O que planeja realizar até a próxima reunião da equipe?
- Demos: Entrega da parcialidade do software ao cliente para demonstração, teste e avaliação pelo cliente. A demo pode não ter toda a funcionalidade planejada, mas funções que possam ser entregues no prazo estipulado.

Pode-se observar o método descrito na Figura 27.

Figura 27 – Fluxo do processo Scrum



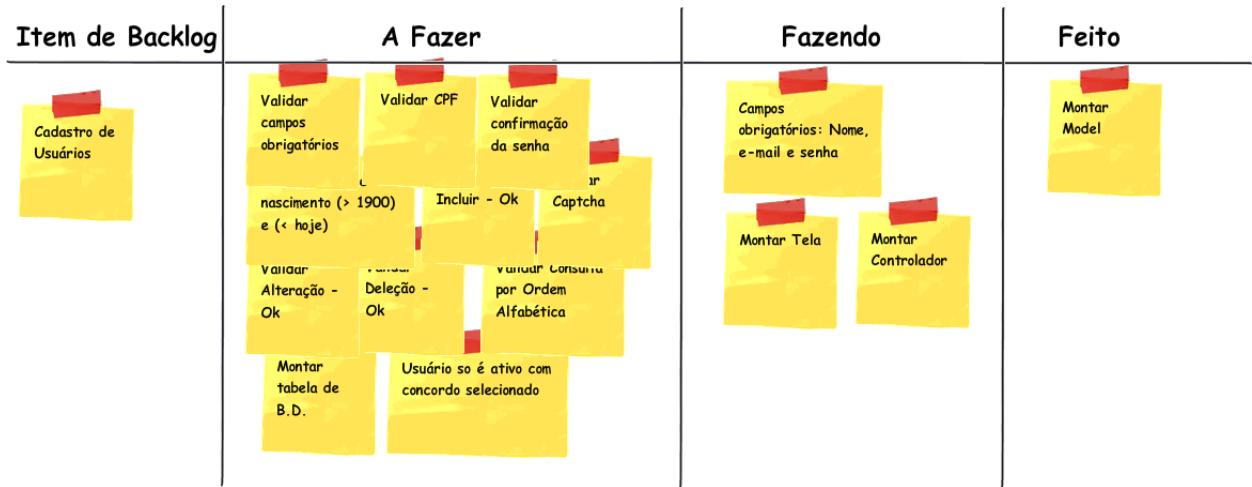
Fonte: Pressman (2010)

2.5.3 Kanban

Segundo Conceição (2015), o Kanban possibilita ter um controle mais detalhado da produção de cada integrante, com informações de quando, quanto e o que produzir. O método Kanban inicialmente foi incorporado em empresas japonesas que utilizavam o conceito *just in time*. A Toyota foi a responsável por introduzir esse método para manter o funcionamento da produção em série. O principal objetivo do Kanban é avaliar o trabalho *WIP Work in Progress*, para mostrar quando uma funcionalidade pode ser arquitetada, codificada, testada etc., isso por meio da visualização.

Conceição (2015) Explica que basicamente o método Kanban permite obter essas informações sobre a produção através de cartões que devem ser preenchidos por todos da equipe e devem ser posicionados em um quadro para sinalizar o andamento das atividades. A Figura 28 exemplifica um quadro tradicionalmente utilizado em projetos ágeis. Esse quadro pode ser modificado conforme necessidade da equipe adicionando colunas como teste por exemplo.

Figura 28 – Exemplo de quadro tradicional Kanban



Fonte: <<https://www.semeru.com.br/blog/wp-content/uploads/2012/09/teste.png>>

2.6 BANCO DE DADOS

Segundo Lima e Moura (2017), banco de dados são estruturas que permitem o armazenamento de informações de forma segura, possibilitando a recuperação, alteração, exclusão e inclusão de dados. Essa estrutura necessita Sistema de Gerenciamento de Banco de Dados (SGBD) para conseguir fazer tal gerenciamento. Dessa forma o sistema de banco de dados é composto pelo banco em si, e o *software* gerenciador SGBD. É esse *software* SGBD que é capaz de realizar as operações de inclusão, exclusão, alteração e recuperação dos dados, o famoso CRUD. Outra função importante do *software* SGBD é o gerenciamento da segurança e integridade dos dados armazenados.

Lima e Moura (2017) explica que atualmente os bancos de dados podem ser separados em duas classificações diferentes. Os bancos SQL e os NoSQL. Essas duas classificações de banco de dados formam dois paradigmas diferentes. Os bancos classificados como SQL formam o paradigma dos bancos relacionais, e os bancos NoSQL formam o paradigma dos bancos não relacionais.

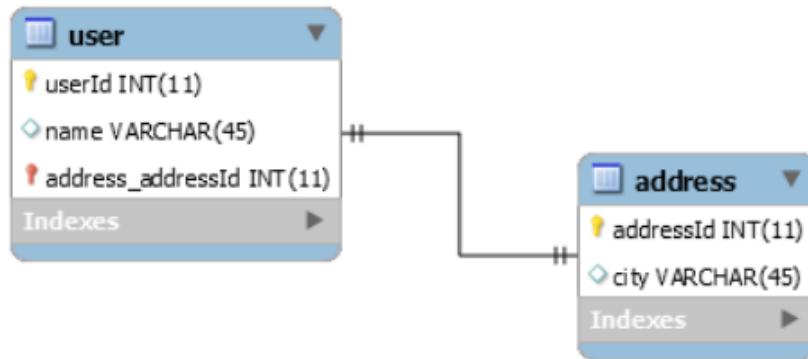
2.6.1 Relacionais

Bancos relacionais são o tipo mais comum de paradigma de banco de dados conforme explica Lima e Moura (2017). Como o próprio nome sugere, os bancos relacionais possuem relacionamento entre suas entidades. Os dados são armazenados em tabelas (entidades), e essas tabelas possuem colunas e os dados armazenados

nessas colunas formam os registros do banco.

Conforme Figura 29 pode-se observar um exemplo de relacionamento entre a entidade *user* e a entidade *address*, onde para cada usuário da tabela *user*, existe um e somente um endereço na tabela *address* formando um relacionamento 1x1.

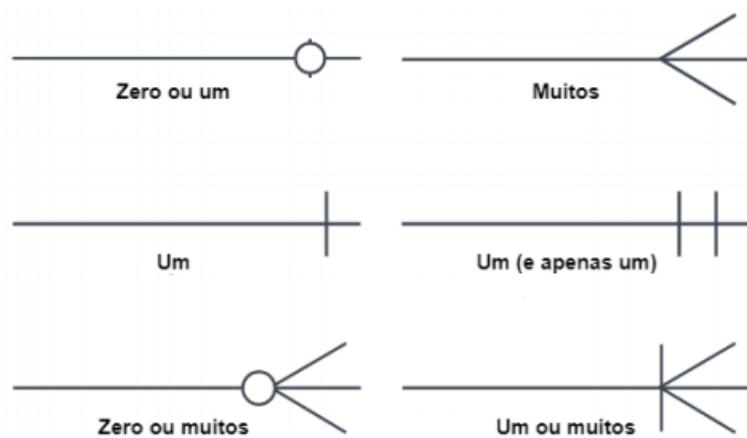
Figura 29 – Representação gráfica de relacionamento entre entidades de um banco SQL.



Fonte: Lima e Moura (2017)

Lima e Moura (2017) explica que esses relacionamentos são descritos de forma gráfica no diagrama através das setas, traços e símbolos que interligam as entidades representando a cardinalidade do relacionamento. A Figura 30 mostra as possíveis representações de cardinalidades em um relacionamento entre entidades de banco SQL.

Figura 30 – Representações indicativas de cardinalidade de relacionamentos entre entidades.



Fonte: Lima e Moura (2017)

2.6.2 Não relacionais

Segundo Lima e Moura (2017) os bancos de dados não relacionais (NoSQL) surgiram para solucionar o problema de armazenamento de grandes volumes de dados. Existem algumas estruturas de banco de dados não relacionais. As principais são:

- **Chave - Valor:** Neste modelo o endereçamento dos dados é feito através de indexação única, onde cada dado tem uma chave única. Os valores armazenados são independentes e isolados, sendo que o relacionamento desses dados acontece apenas por parte da aplicação. A utilização desse modelo de banco de dados foca em aplicações simples baseadas em indexação, e uma das vantagens do modelo é a rápida pesquisa.
- **Orientado a colunas:** Os dados nesse tipo de banco de dados são armazenados em colunas, onde cada coluna compõem um registro. As colunas podem também ser agrupadas em famílias, e o relacionamento entre os dados fica por conta da aplicação. Um exemplo desse tipo de banco é o Apache Cassandra.
- **Banco de dados em Grafos:** Esse tipo de banco de dados, como o próprio nome sugere, tem a base no formato de um grafo. É especializado em dados fortemente conectados, sendo recomendado para aplicações que possuem muitos relacionamentos, devido ao baixo custo de pesquisas contendo muitas conexões. Alguns exemplos desse tipo de banco são o AlegroGraph e o ArangoDB.
- **Orientado a documentos:** Este tipo de banco, organiza os dados no formato de documentos, sendo estruturados em árvores hierárquicas e capazes de coleções, mapas e outros tipos de objetos. No banco orientado a documentos, os documentos não precisam ter estrutura similar, e cada documento possui um ID único para ser identificado na coleção. Os documentos não possuem relacionamento entre si, sendo assim livres de esquemas. Alguns exemplos desse tipo de banco são o MongoDB e o CouchDB.

2.6.3 Relacional X Não relacional

Segundo Lima e Moura (2017) ambos os bancos de dados relacionais e não relacionais possuem vantagens e desvantagens. A Tabela 8 mostra um comparativo entre os dois modelos de banco de dados.

Tabela 8 – Analise comparativa Relacional X Não Relacional

Critério	SQL	NoSQL
Escalabilidade	Possível, mas complexo. Devido à natureza estruturada do modelo, a adição de forma dinâmica e transparente de novos nós no grid não é realizada de modo natural.	Uma das principais vantagens desse modelo. Por não possuir nenhum tipo de esquema pré-definido, o modelo possui maior flexibilidade o que favorece a inclusão transparente de outros elementos.
Consistência	Ponto mais forte do modelo relacional. As regras de consistência presentes propiciam um maior grau de rigor quanto à consistência das informações.	Realizada de modo eventual no modelo: só garante que, se nenhuma atualização for realizada sobre o item de dados, todos os acessos a esse item devolverão o último valor atualizado.
Disponibilidade	Dada a dificuldade de se conseguir trabalhar de forma eficiente com a distribuição dos dados, esse modelo pode não suportar a demanda muito grande de informações do banco.	Outro fator fundamental do sucesso desse modelo. O alto grau de distribuição dos dados propicia que um maior número de solicitações aos dados seja atendido por parte do sistema e que o sistema fique menos tempo não-disponível.

Fonte: Adaptado de Lima e Moura (2017)

2.7 STREAMING

Segundo Gomes e Lourenço (2015), a tecnologia *streaming* envia informações multimídia via transferência de dados entre computadores em rede. Dessa forma é possível visualizar o conteúdo multimídia através da internet por exemplo. Isso se assemelha muito ao tradicional *download*, porém a diferença entre *streaming* e *download* estão basicamente no tipo de servidor e protocolos utilizados para a transmissão. Em um *streaming* os ficheiros visualizados não ficam armazenados em uma pasta como os *downloads*. Além disso a qualidade de um *streaming* depende da conexão e ritmo de transmissão os quais são possíveis alcançar. Hoje em dia, a maior percentagem de tráfego vem dos *streamings* de vídeos. Plataformas como o YouTube são grandes disseminadoras desse tipo de conteúdo.

Por trás de uma transmissão de áudio ou vídeo via *streaming*, acontecem uma série de artifícios para possibilitar a reprodução do conteúdo. Primeiro o cliente se conecta com um servidor que está transmitindo algum conteúdo, isso constrói um *buffer* no cliente, o qual armazena a informação transmitida pelo servidor para que quando o *buffer* estiver cheio, comece a reprodução do conteúdo, conforme explica

Gomes e Lourenço (2015).

2.7.1 Tipos e protocolos de streaming

Conforme explica Santos et al. (2010), existem atualmente três tipos genéricos de abordagens de streaming. São eles:

- ***streaming tradicional***: Utiliza o *Real Time Streaming Protocol* (RTSP) como protocolo. Sua principal característica é o fato de possuir sessões que são iniciadas quando é estabelecida a ligação com o cliente, e termina quando a transmissão dos dados se completa ou é interrompida. Assim, existe apenas uma única transmissão contínua de dados, ao contrário dos outros modelos os quais segmentam em várias transmissões o conteúdo. Existe também o controle por parte do cliente enviando mensagens pré-definidas como *Play* ou *Pause*. Apesar dos dados não serem segmentados ao enviar ao cliente, a informação é codificada em pacotes pequenos do RTSP, que são codificados sobre o protocolo *User Datagram Protocol* (UDP), TCP ou até mesmo o HTTP.
- ***Download progressivo***: Consiste no *download* de arquivos disponibilizados a partir de um servidor web HTTP, e por conta disso é suportado pela maioria das plataformas de reprodução de conteúdo multimídia. O termo “progressivo” refere-se a possibilidade de reproduzir o conteúdo enquanto o *download* está acontecendo. O cliente também pode escolher visualizar parte do conteúdo que ainda não esteja disponível. Isto é feito através da troca de mensagens entre o cliente e o servidor, onde o cliente envia o “*Byte Range Request*”, que é o pedaço do conteúdo o qual ele deseja visualizar, para então o servidor fornecer esse *range* especificado. Esta solução é adotada por vários sites de compartilhamento de vídeos como o Youtube, Vimeo e Myspace. Isso por conta da capacidade desse modelo de continuar a transmissão dos dados até o *download* estar concluído. Se o utilizador suspender a reprodução de um conteúdo que está sendo reproduzido através de *download* progressivo, todo o vídeo poderá ser transferido para o cliente, possibilitando a reprodução completa sem interrupções.
- ***Streaming adaptativo sobre HTTP***: Consiste em um sistema híbrido com funcionamento similar ao *streaming* normal, mas se baseia no *download* progressivo sobre HTTP, pois utiliza o HTTP como protocolo de transporte. Nesse modelo, ao invés de transmitir apenas um único fluxo de dados completo, transmite vários segmentos, que quando são reproduzidos em ordem formam o conteúdo multimídia. Para isto, os arquivos a serem transmitidos devem estar guardados no servidor web já codificados em seguimentos, para que envie esses seguimentos

ao cliente, que receberá e ordenará os segmentos possibilitando formar um fluxo contínuo de reprodução do arquivo.

2.7.2 Protocolos

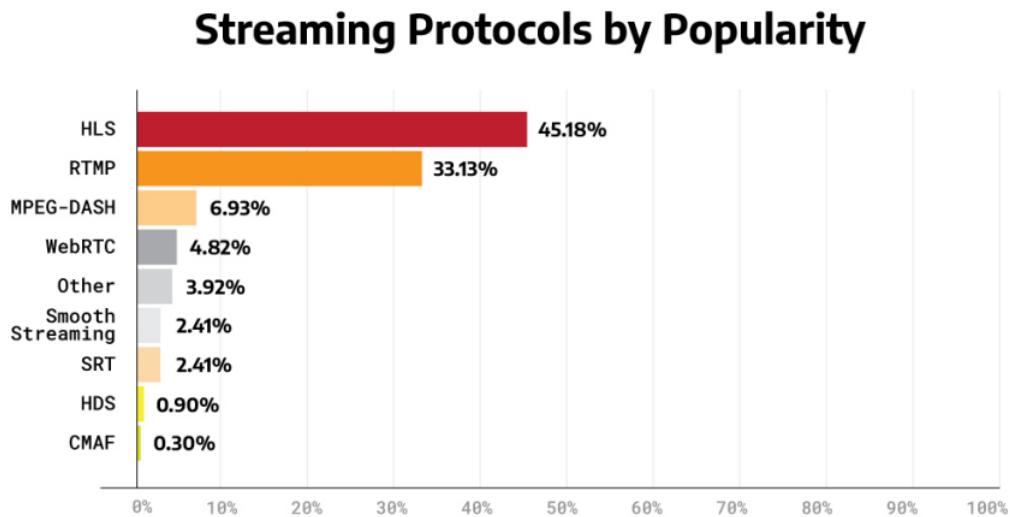
Segundo Gomes e Lourenço (2015), os primeiros protocolos usados para *streaming* foram o RTSP para fornecer o conteúdo a ser transmitido em sequência *unicast* e *Real Time Protocol* (RTP) para transportar esses pacotes ao destinatário. Nesse modelo é o servidor que decide os fragmentos do conteúdo a ser enviado e o cliente apenas recebe o conteúdo e o reproduz.

Mais recentemente o *streaming* utiliza o protocolo HTTP para transmitir o conteúdo multimídia. E o faz a partir de um servidor qualquer, não necessitando um servidor especial para a transmissão. O servidor codifica o conteúdo a ser transmitido com diferentes qualidades, e envia os fragmentos pedidos pelo cliente, podendo ajustar a dinamicamente a qualidade do conteúdo transmitido de acordo com a velocidade da conexão com o cliente, e o cliente consegue escolher o *bit rate* dos fragmentos, conforme explica Gomes e Lourenço (2015).

Com essa capacidade de adaptação dos protocolos de *streaming* surgiram os três principais, Adobe HDS, Apple HLS e Microsoft Smooth *streaming*, cada uma com seu sistema fechado, seus próprios formatos e todas baseadas no modelo HTTP descrito anteriormente.

Uma pesquisa feita pelo Better Software Group, indicou o protocolo *HTTP Live Streaming* (HLS) como sendo o mais popular (conforme pode-se observar na Figura 31) e indicado devido sua adaptabilidade com as mais variadas conexões, e sua compatibilidade, que apesar de inicialmente ter sido desenvolvido para dispositivos Apple, atualmente é compatível com os mais modernos navegadores e sistemas operacionais e *players*.

Figura 31 – Popularidade dos protocolos de *streaming*



Fonte:

<<https://www.bsgroup.eu/what-is-the-hls-protocol-and-why-it-is-popular-in-the-streaming-industry/>>

2.7.3 Qualidade de software

Segundo Sommerville (2011), a avaliação da qualidade de um software é um processo o qual é necessário usar seu julgamento para decidir se foi alcançado um nível aceitável de qualidade. Deve ser considerado se o software é adequado a sua finalidade. Essa avaliação é feita através da respostas de perguntas. O autor menciona seis perguntas que devemos fazer a respeito de características do software:

- Diante o processo de desenvolvimento os padrões de programação e documentação foram seguidos?
- O software foi devidamente testado?
- O software é suficientemente confiável para ser colocado em uso?
- O desempenho do software é aceitável para uso normal?
- O software é útil?
- O software é bem estruturado e comprehensível?

Sommerville (2011) diz que a qualidade de software não implica apenas se a funcionalidade foi implementada. A qualidade de software se baseia principalmente em suas características não funcionais. Isso reflete a experiência prática do usuário. Se a funcionalidade do software não é esperada, os usuários apenas contornam a situação, encontrando outras maneiras de resolver o problema. No entanto se o software for muito lento, ou não confiável, será praticamente impossível aos usuários atingirem seus objetivos. Sommerville (2011) sugere 15 atributos relacionados a confiança, usabilidade, eficiência e a manutenibilidade de software, conforme observado na Tabela 9.

Tabela 9 – Atributos da qualidade de software

Segurança	Compreensibilidade	Portabilidade
Proteção	Testabilidade	Usabilidade
Confiabilidade	Adaptabilidade	Reusabilidade
Resiliência	Modularidade	Eficiência
Robustez	Complexidade	Capacidade de aprendizado

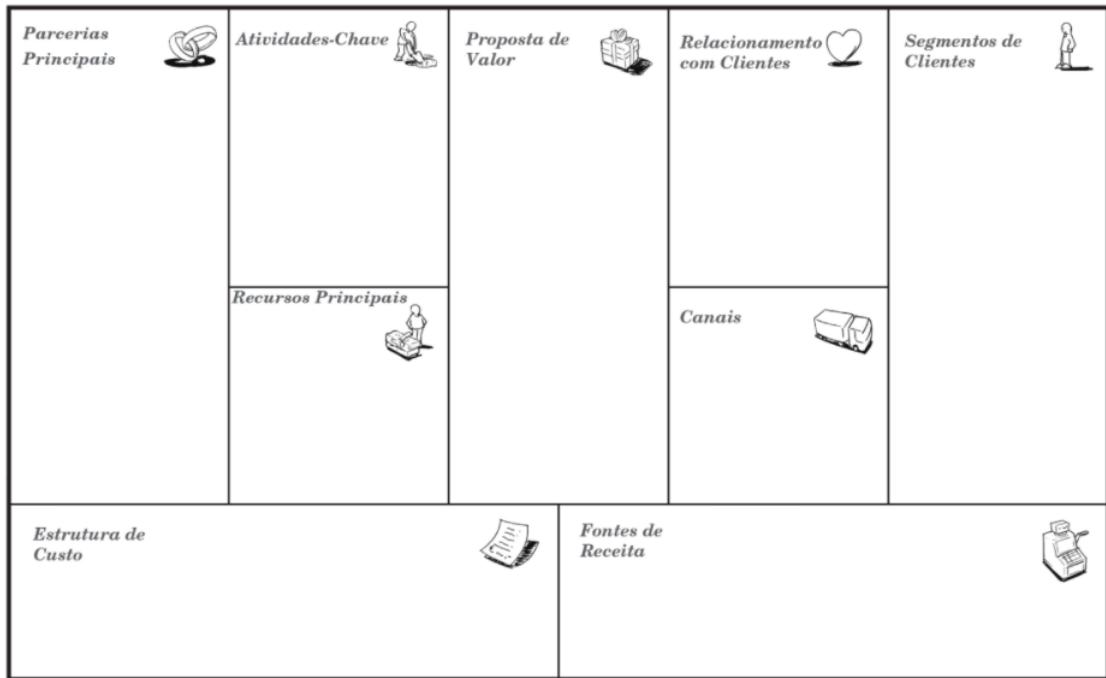
Fonte: Adaptado de Sommerville (2011)

2.8 MODELO DE NEGÓCIO

Segundo Silva e Marciano (2017), para um negócio obter vantagem competitiva e entregar valor, deve-se entender qual é o seu negócio, e uma ferramenta utilizada para ter essa compreensão é o modelo de negócio. O modelo de negócio reflete a lógica da empresa, como funciona e cria valor aos seus *stakeholders*. Dessa forma, o modelo de negócio contempla não somente os valores que serão agregados ao negócio, mas também a cadeia de clientes, parceiros e fornecedores.

Silva e Marciano (2017) menciona que o modelo de negócio é a “*a representação de um sistema construído para estudar algum aspecto daquele sistema ou o sistema como um todo*”, e ainda que o modelo pode ser descritivo, explicativo, de simulação, estático e ou dinâmico. Sempre visando entrega de valor para empresa e clientes e sociedade, os modelos passam por inovações. Uma dessas inovações em modelos de negócios foi proposta por Osterwalder e Pigneur (2020), que apresentou o modelo de negócio CANVAS composto por 9 blocos, conforme visto na Figura 32.

Figura 32 – Exemplo do modelo de negócio CANVAS



Fonte: Osterwalder e Pigneur (2020)

Conforme Silva e Marciano (2017), os 9 blocos desse modelo estão incluso de uma das quatro macro áreas do modelo. Na Figura 33 pode-se observar as 4 macro áreas divididas nos 9 blocos e também a descrição do bloco e perguntas que auxiliam o preenchimento do bloco.

Figura 33 – Macro áreas e os nove blocos do modelo de negócio CANVAS.

GRUPOS		DESCRIÇÃO	PERGUNTAS QUE NORTEIAM O DESENVOLVIMENTO
PRODUTO	Proposta de valor	Conjunto de produtos e serviços que criam valor para um segmento de clientes específico.	Que valor entregou ao cliente? Quais problemas estão ajudando a resolver? Que necessidades estão satisfazendo? Que conjunto de produtos e serviços está oferecendo para cada segmento de cliente?
	Segmentos de cliente	São os diferentes grupos de pessoas a quem uma organização deseja oferecer algo de valor.	Para quem estamos criando valor? Quem são nossos consumidores mais importantes?
	Canais	São os meios empregados pela organização para manter contato com os clientes.	Através de quais canais nossos segmentos de cliente querem ser contatados? Como os alcançamos agora? Como nossos canais se integram? Qual funciona melhor? Quais apresentam melhor custo-benefício? Como estão integrados à rotina dos clientes?
OFERTA DE VALOR	Relacionamento com clientes	Descreve o tipo de relacionamento que a organização estabelece com seus clientes.	Que tipo de relacionamento cada um dos nossos segmentos de clientes espera que estabeleçamos com eles? Qual já estabeleceu? Qual o custo de cada um? Como se integram ao restante do nosso modelo de negócio?
	Recursos principais	Descreve a organização das atividades e recursos que são necessários para criar valor para os clientes.	Que recursos principais nossa proposta de valor requer? Nossos canais de distribuição? Relacionamento com o cliente? Fontes de receita?
	Atividades chave	Habilidades em realizar as ações necessárias mais importantes para criar valor para os clientes.	Que atividades-chave nossa proposta de valor requer? Nossos canais de distribuição? Relacionamento com o cliente? Fontes de receita?
INFRAESTRUTURA	Parcerias principais	Principais redes de fornecedores e os parceiros que fazem o modelo de negócio funcionar.	Quem são nossos principais parceiros? Quem são nossos fornecedores principais? Que recursos principais estão adquirindo dos parceiros? Que atividades-chave os parceiros executam?
	Estrutura de custo	É a descrição de todos os custos envolvidos na operação do modelo de negócio.	Quais são os custos mais importantes em nosso modelo de negócio? Que recursos principais são mais caros? Quais atividades-chave são mais caras?
	Fontes de receita	Descreve a maneira como a organização ganha dinheiro através de cada segmento de clientes.	Quais valores nossos clientes estão realmente dispostos a pagar? Pelo que eles pagam atualmente? Como pagam? Como prefeririam pagar? O quanto cada fonte de receita contribui para o total da receita?
VIABILIDADE FINANCEIRA			

Fonte: Adaptado de Silva e Marciano (2017)

2.9 CONSIDERAÇÕES FINAIS

Neste capítulo, foram sintetizados estudos que ajudaram o planejamento e desenvolvimento do protótipo apresentado neste trabalho.

Entre esses estudos, estão a documentação da aplicação através de artefatos apresentados na subseção 2.1.1, porém como a aplicação trata-se de um protótipo pequeno optou-se em utilizar os diagramas de atividades, diagramas de casos de uso, diagrama de sequência, diagramas de classe. Os quais representam muito bem o protótipo desenvolvido.

Outro ponto o qual o estudo deste capítulo colaborou, foi na tomada de decisão sobre qual arquitetura escolher para o desenvolvimento. Visto que um dos critérios para desenvolvimento do protótipo é a disponibilidade da aplicação via web e do es-

tudo realizado na subseção 2.1.2, definiu-se a utilização da arquitetura no modelo de serviço web. E para a comunicação desta aplicação, optou-se pela arquitetura REST por suas vantagens mencionadas na subseção 2.1.3.

Após a definição da arquitetura, observou-se que ocorreria a separação do código do *back-end* e *front-end*. Diante disso, e com o estudo realizado na seção 2.2, a *stack* que se enquadrava na arquitetura, era a *stack* MERN, por possibilitar a utilização de apenas uma linguagem, com o *back-end* utilizando o NodeJS, o *front-end* com React Native. O banco de dados, utilizando o MongoDB que já salva as informações no formato JSON, e utilizando o Mongoose para realizar as *querys*. Ainda sobre o banco de dados, optou-se por um banco não relacional devido ao pouco relacionamento entre as entidades, conforme estudo descrito na seção 2.6.

Na subseção 2.3.1 foi realizado o estudo sobre formatos de versionamento de código, onde o Git juntamente com o GitHub foi escolhido para ser utilizado por seus diferenciais e facilidades descritos na seção. Instalações de bibliotecas, atualizações e outras manipulações de pacotes foi optado em utilizar o Yarn, devido a escolha da linguagem (JavaScript) e sua maior velocidade se comparado com o NPM, conforme descrito na subseção 2.3.2. A escolha do editor de código, foi baseado na pesquisa de utilização descrita na subseção 2.3.3, onde o Visual Studio Code ganhou com folga.

Outras ferramentas que ajudam no desenvolvimento de código foram estudadas na seção 2.3 como as ferramentas de análise estática de código, onde foi escolhido trabalhar com o EsLint para manter a padronização do código. Outra ferramenta de desenvolvimento como a *template engine* EJS que é a mais conhecida para JavaScript e também outras ferramentas como o Nodemon, foram utilizadas durante o desenvolvimento. Um estudo sobre *upload* de arquivos e seus padrões, os quais foram empregados no protótipo utilizando os métodos da arquitetura REST. Para a realização das requisições foi escolhido o Axios dentre as opções mencionadas na seção 2.4 devido a sua facilidade enviar requisições.

No quesito metodologia de desenvolvimento de software, conforme mencionado na seção 2.5, foi utilizado alguns dos princípios do Scrum como as reuniões semanais, e revisão e correção da produção antes da entrega. Também foi adaptado para uma planilha os conceitos do Kanban, onde se controlava as tarefas pendentes, em andamento e concluídas.

Por conta do tema do protótipo desenvolvido ser um aplicativo de streaming de vídeo na subseção 2.7.1 foi feita uma pesquisa sobre tipos e protocolos de streaming, e foi decidido utilizar o streaming sobre o protocolo HTTP, aproveitando as novas funcionalidades que o NodeJS possui para tratar *stream* de arquivos, quebrando o arquivo em partes e transmitindo em uma requisição continua parte por parte, sob

demandas do cliente, enviando os fragmentos do arquivo via response nos métodos da arquitetura REST. Por fim, foi estudado sobre qualidade de software para realizar uma avaliação do protótipo após seu desenvolvimento, conforme as métricas descritas na subseção 2.7.3.

3 TRABALHOS RELACIONADOS

Nesta seção são apresentados trabalhos relacionados a este. Foi realizada pesquisa de outros trabalhos acadêmicos os quais o tema era o desenvolvimento *full-stack* com JavaScript.

Um dos trabalhos escolhidos como correlato, é a tese desenvolvida por Keinänen (2018), o qual fez um estudo sobre o desenvolvimento de serviços web utilizando a *stack* MERN, onde abordou temas como:

- O uso do MongoDB e armazenamento utilizando MongoDB Atlas;
- Estudo sobre o NodeJS, informações básicas e suas atualizações, compatibilidade, *Event-loop*, Node Package Manager, performance do NodeJS, uso do NodeJS no *Back-end*, *front-end* e em *services*;
- Utilização do *framework* Express, *delivery* de conteúdo estático e roteamento;
- Autenticação;
- Uso do React, informações básicas, atualizações e performance;
- JSX;
- *Virtual DOM*;
- *Data binding*;

Outro trabalho correlato, foi feito por Aggarwal e Verma (2018), o qual diferente de Keinänen (2018) que já tinha uma *stack* definida, este fez um estudo sobre duas *stacks* JavaScript, a MERN e a MEAN. Duas poderosas *stacks* para desenvolvimento de aplicações modernas conforme o autor descreve. O fator chave do sucesso dessas duas stacks segundo Aggarwal e Verma (2018), é que ambas são baseadas em JavaScript. O autor descreve também as vantagens e desvantagens de cada uma das tecnologias utilizadas nas *stacks*, e também traz um quadro sintetizado de atributos, comparando as duas *stacks* conforme visto na Tabela 10.

Tabela 10 – MEAN stack VS MERN stack

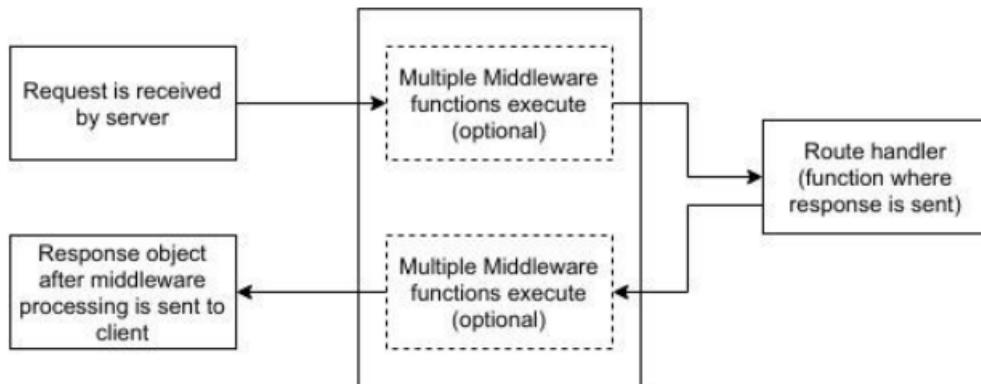
Atributo	MEAN	MERN
Churn	Reduzido	Alto
Tooling	Baixo	Auto
Projeto	JS e HTML	JS somente
Fadiga JS	Menos	Mais
DOM	Regular	Virtual
Complexidade	Alta	Baixa
Empacotamento	Fraco	Forte
Falha	Tempo de execução	Tempo de compilação
Binding	Two-way	Unidirecional
Template	No html	No JSX
Model	Forte	Média
Renderização	No cliente	No servidor

Fonte: Adaptado de Aggarwal e Verma (2018)

Nguyen (2020) realizou um estudo correlato a este trabalho, onde abordou a construção de uma aplicação de ponta-a-ponta utilizando a stack MERN. Assim como Keinänen (2018), ele descreveu todo o processo e tecnologias envolvidas no seu estudo. Alguns dos assuntos abordados por ele foi o padrão *REST*, seus métodos, comunicação com a API, a organização dos *request* e *responses* nas rotas do *back-end* da aplicação utilizando o *framework* Express e *middlewares* conforme exemplificado na Figura 34).

Essa proposta foi também utilizada na construção das rotas da API Meraki. Nguyen (2020) também trata sobre o MongoDB, Mongoose e hospedagem no mongoDB atlas, bem como trouxe explicações sobre o *event loop* do NodeJS dentre outras tecnologias que utilizou em seu projeto, das quais muitas delas foram escolhidas também para o projeto Meraki.

Figura 34 – Roteamento com middlewares



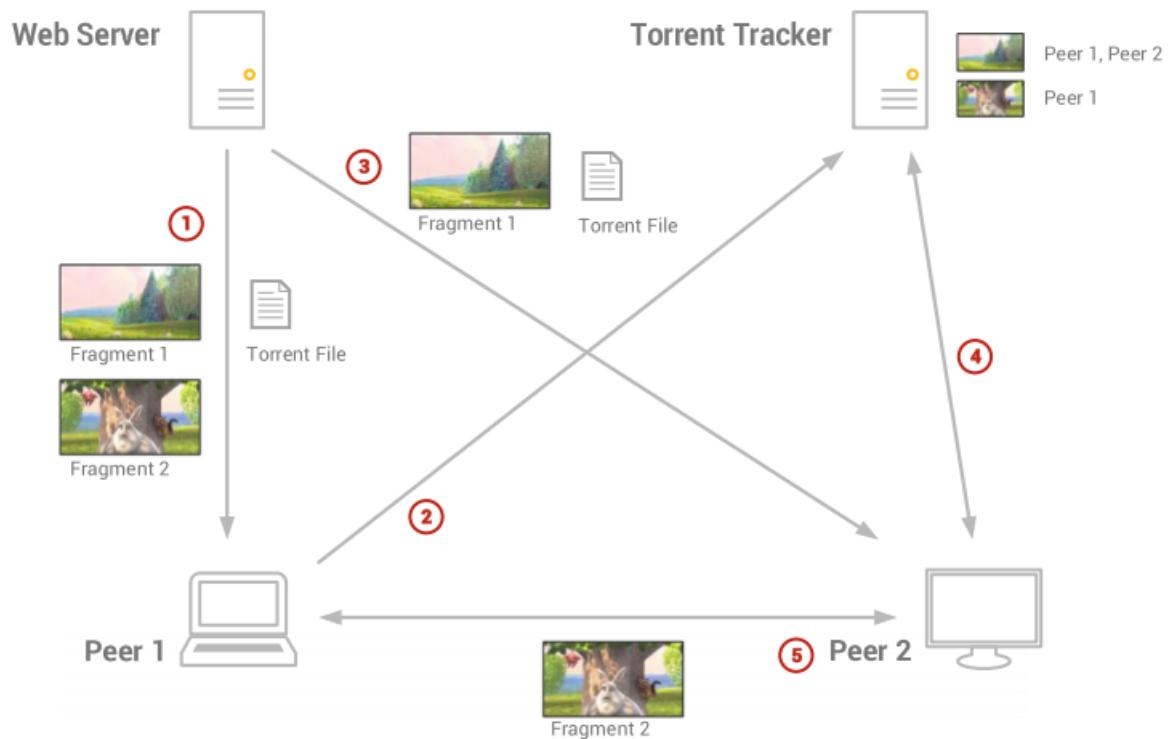
Fonte: Nguyen (2020)

Keinänen (2018) detalhou muito bem as tecnologias utilizadas na *stack* por ele escolhida. Entretanto, não considerou outras possibilidades entre as diversas *stacks* e tecnologias existentes, diferente de Aggarwal e Verma (2018) que em seu estudo conseguiu abstrair as duas *stacks* principais conforme estudo por ele feito, e comparou ambas.

Koren e Klamma (2018) tem relação com este trabalho por também se tratar de um estudo alternativo, barato e de fácil acesso a pequenos desenvolvedores e empresas que necessitam a realizar *streaming* de vídeo sem depender de uma grande infraestrutura de servidores.

Embora assim como a aplicação Meraki, Koren e Klamma (2018) traz como objeto de estudo a transmissão *Peer-to-peer* utilizando javaScript, conforme exemplificado na Figura 35, a aplicação Meraki, diferente de Koren e Klamma (2018) traz uma solução também de baixo custo, e requisitos para a transmissão de vídeo, mas com o advento do node *streams*, a aplicação Meraki pode aproveitar o próprio método HTTP para realizar essa transmissão dos fragmentos dos arquivos de vídeo embutidos no response através método *pipe()* diretamente para o HTML do browser cliente. Está é uma abordagem alternativa ao estudo de Koren e Klamma (2018), onde os fragmentos são transmitidos do browser do cliente para o browser do cliente.

Figura 35 – Arquitetura de stream peer-to-peer e troca de fragmentos



Fonte: Nguyen (2020)

4 DESENVOLVIMENTO DA APLICAÇÃO MERAKI

Este capítulo aborda o desenvolvimento da aplicação Meraki, desde o planejamento, a identidade da aplicação, protótipos, artefatos do projeto, requisitos, construção da API para o *back-end*, a construção do *front-end*, e a distribuição.

4.1 PLANEJAMENTO

No início da construção da aplicação Meraki primeiramente foi definido a temática da aplicação, nomes, cores, e prototipação das interfaces das telas. Também para alinhamento dos avanços do projeto foi feito reuniões semanais com uma metodologia ágil, similar ao *scrum* já mencionado na subseção 2.5.2.

Para a decisão do nome, após uma seleção de possíveis nomes para a aplicação e para o projeto, decidiu-se utilizar o nome “Meraki” por conta de seu significado. Meraki é uma palavra grega que significa “Dar parte de si em algo” e “Fazer algo com a alma”, o que condiz com a proposta da aplicação, já que usuários poderão se dedicar em ensinar e aprender algo.

O logotipo observado na Figura 36 foi desenvolvido em um programa de desenho vetorizado, onde foi desenhado a silhueta de uma coruja ao lado do nome da aplicação como sendo o logotipo. A coruja foi escolhida por sua simbologia relacionada a educação, aulas e ao erudito, já a fonte para o nome da aplicação no logo, foi escolhida uma fonte classificada como sendo do tipo *Script*, que simula um manuscrito (o que remete a escrita em quadro negro, caderno de anotações e estudos em geral).

A cor roxa escolhida para o logo, também está presente em várias partes da aplicação, bem como os subtons dessa cor, pois conforme a psicologia das cores, esse tom transmite emoções e sensações de criatividade, imaginação e sabedoria, que condizem com o intuito da plataforma. Em determinadas telas da aplicação, o logo apresentado será branco, para melhor contraste com o fundo da tela.

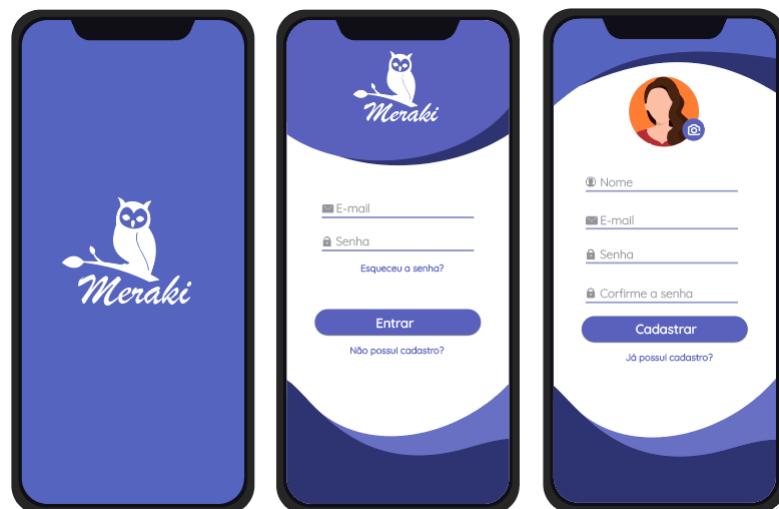
Figura 36 – Logo Meraki



Fonte: Elaborado pelo autor, 2020

Os protótipos das telas foram divididos em dois grupos. As telas de autenticação, que aparecem antes do login do usuário, e as telas da plataforma, quando o usuário já está logado. A prototipação das telas foi feita através de um software de desenho e vetorização, para ser base do desenvolvimento e coleta de requisitos. Na Figura 37 pode ser conferido os protótipos das telas de autenticação e aplicação, onde a tela da esquerda é a tela de *splash screen* utilizada durante o carregamento da aplicação. Na sequência a tela de login e a direita a tela de cadastro.

Figura 37 – Telas de autenticação: *Splash screen*, login e cadastro

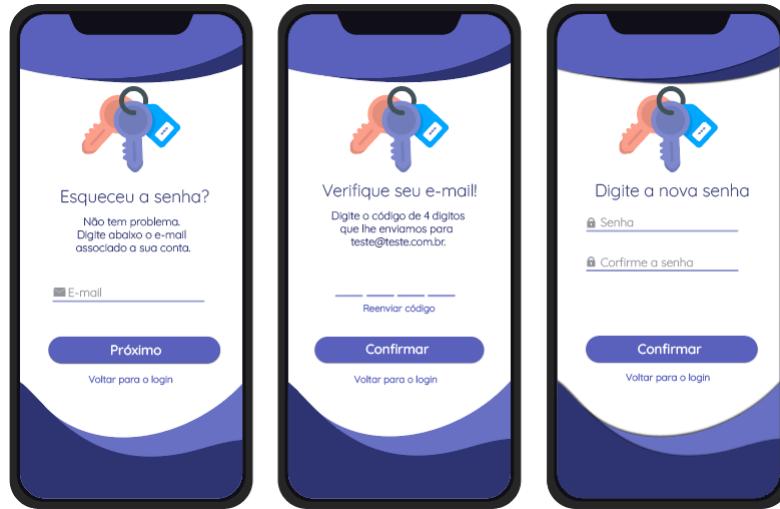


Fonte: Elaborado pelo autor, 2020

Na figura Figura 38 pode ser conferido na sequência a tela de recuperação de senha, onde o usuário informa o e-mail cadastrado para receber o código de autenti-

cação, a tela de validação do código e a tela de criação da nova senha.

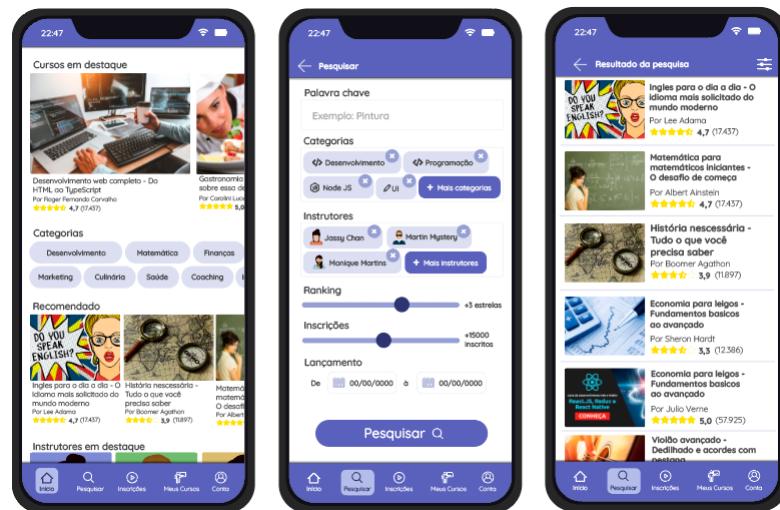
Figura 38 – Telas de autenticação: Esqueci a senha, verificar código e trocar senha



Fonte: Elaborado pelo autor, 2020

É observado na Figura 39 a esquerda a tela inicial da aplicação, onde é apresentado cursos e instrutores em destaque, recomendações, categorias etc. Ao centro, é observado a tela de pesquisa com filtros e a direita a listagem resultante da pesquisa.

Figura 39 – Tela inicial e pesquisar

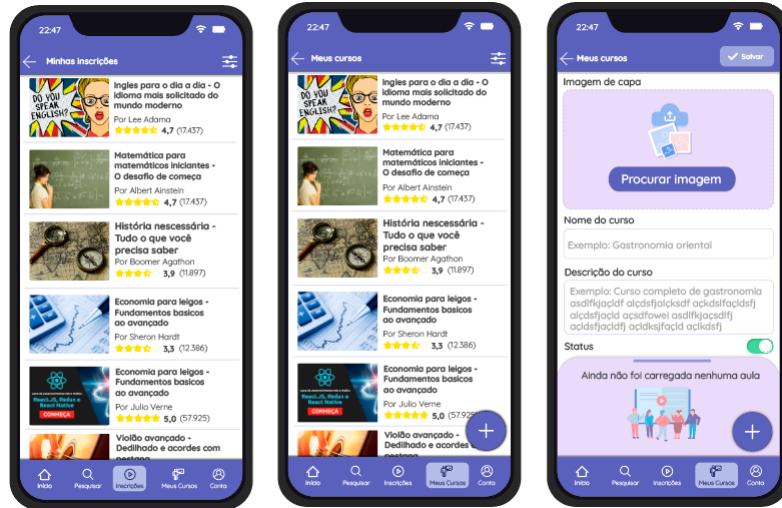


Fonte: Elaborado pelo autor, 2020

Na Figura 39 a esquerda observamos a tela de inscrições onde é listado os

cursos os quais o usuário se inscreveu, no meio a tela dos meus cursos, onde são listados os cursos publicados pelo usuário, e a direita a tela do formulário de upload de novo curso.

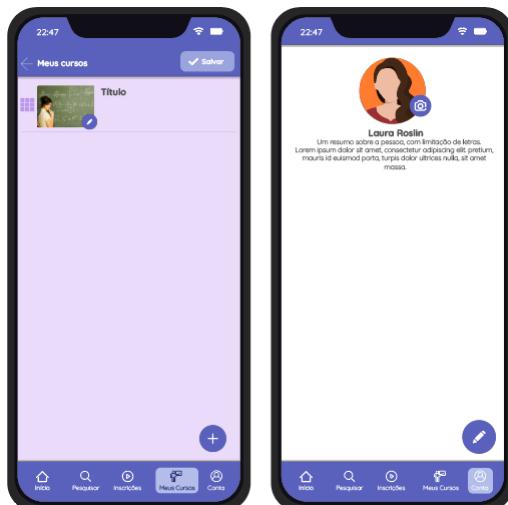
Figura 40 – Tela inscrições e meus cursos



Fonte: Elaborado pelo autor, 2020

Na Figura 39 é observado a esquerda a tela de inclusão de vídeos em um curso criado pelo usuário, e a direita a tela de configurações da conta, onde pode ser feito a troca do nome, foto, resumo, e demais dados cadastrar.

Figura 41 – Tela meus cursos e minha conta



Fonte: Elaborado pelo autor, 2020

Também foi empregado o uso da ferramenta de modelo de negócio CANVAS, conforme descrito na seção 2.8, para criar o modelo de negócio do projeto MERAKI, que pode ser observado na Figura 42.

Figura 42 – Mode de negócio do projeto MERAKI



Fonte: Elaborado pelo autor, 2021

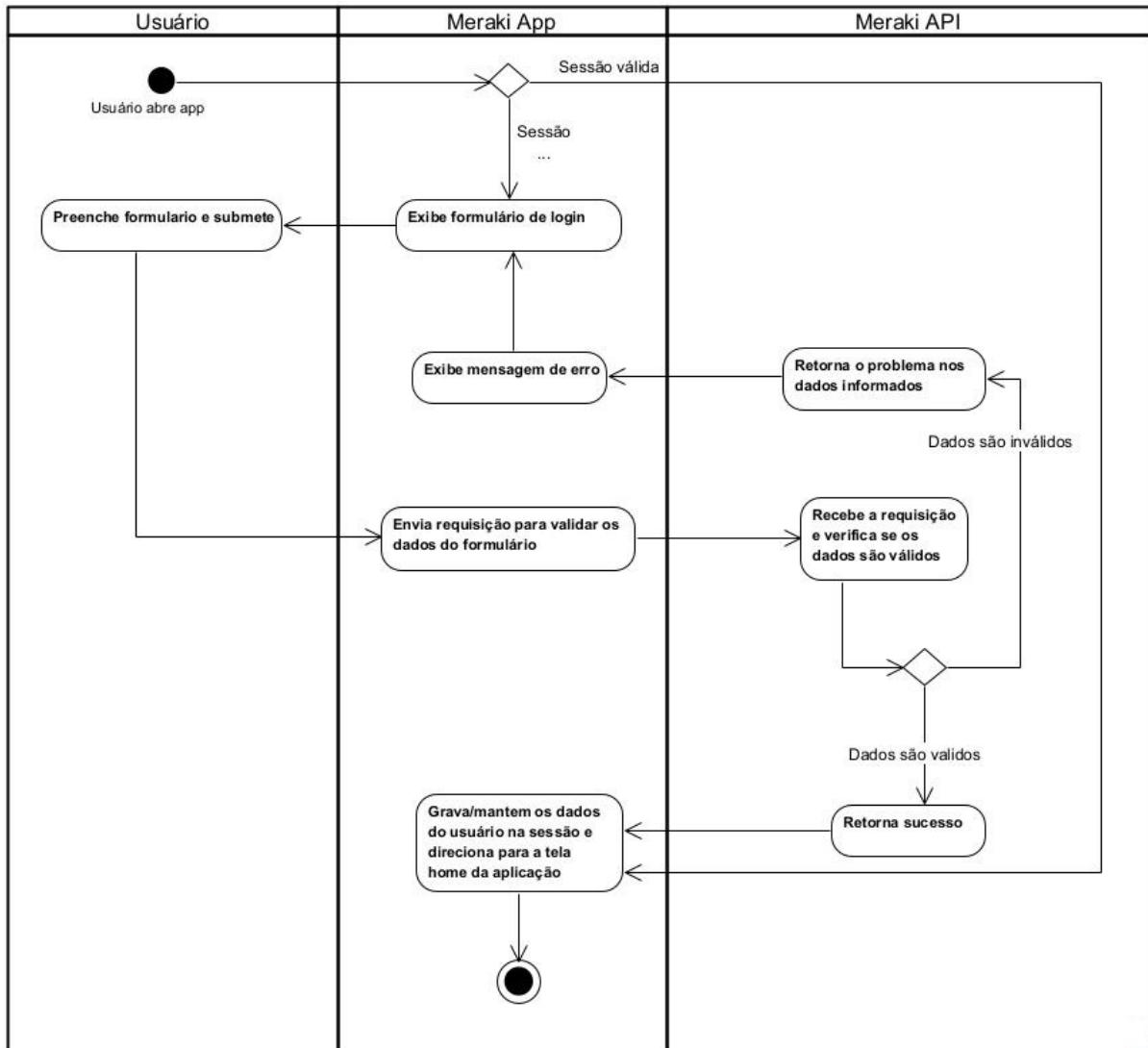
As tecnologias e arquiteturas a serem utilizadas foram decididas na sequência, após o estudo bibliográfico e a criação dos artefatos do projeto dos quais foram obtidos os requisitos. Conforme estudo feito na subseção 2.1.1, existem muitos artefatos que auxiliam o planejamento e desenvolvimento de software. Contudo, existem 5 deles os quais são considerados como sendo os artefatos que representam a essência do sistema pela comunidade de desenvolvimento de software, e estes mesmos artefatos foram escolhidos para representar o projeto Meraki. Os diagramas escolhidos são, o diagrama de atividades, diagramas de caso de uso, diagrama de sequência, diagramas de classe e diagramas de estado.

Para diagramar a utilização dos fluxos de autenticação, cadastro e recuperação de senha pelo aplicativo Meraki, optou-se por utilizar o diagrama de raias, que é uma variação do diagrama de atividades, permitindo descrever e separar as atividades sob responsabilidade de cada ator. Nesses diagramas, as atividades estão divididas sob execução de três atores, o usuário, o APP e a API.

O fluxo de autenticação se inicia quando o usuário abre o aplicativo Meraki, o qual faz uma verificação se existe alguma sessão válida. Caso exista sessão válida, o APP, direcionará o usuário para a tela inicial da aplicação. Caso não possua sessão válida, exibirá o formulário de login. O usuário então deve preencher e submeter o formulário, onde após submete-lo o APP envia uma requisição para a API que validará os dados enviados. caso os dados forem válidos, retornará à requisição com sucesso. Caso os dados forem inválidos, retornará à requisição com o problema no dados informados. Neste caso, o APP ao receber o retorno da API com problema, exibirá uma mensagem na tela referente ao problema nos dados informados e permanecerá na tela do formulário de login, formando um *loop* até que os dados informados pelo usuário sejam validados.

Os detalhes desse diagrama que descreve o fluxo de autenticação são observados na Figura 43.

Figura 43 – Diagrama de Raia descrevendo o fluxo de autenticação

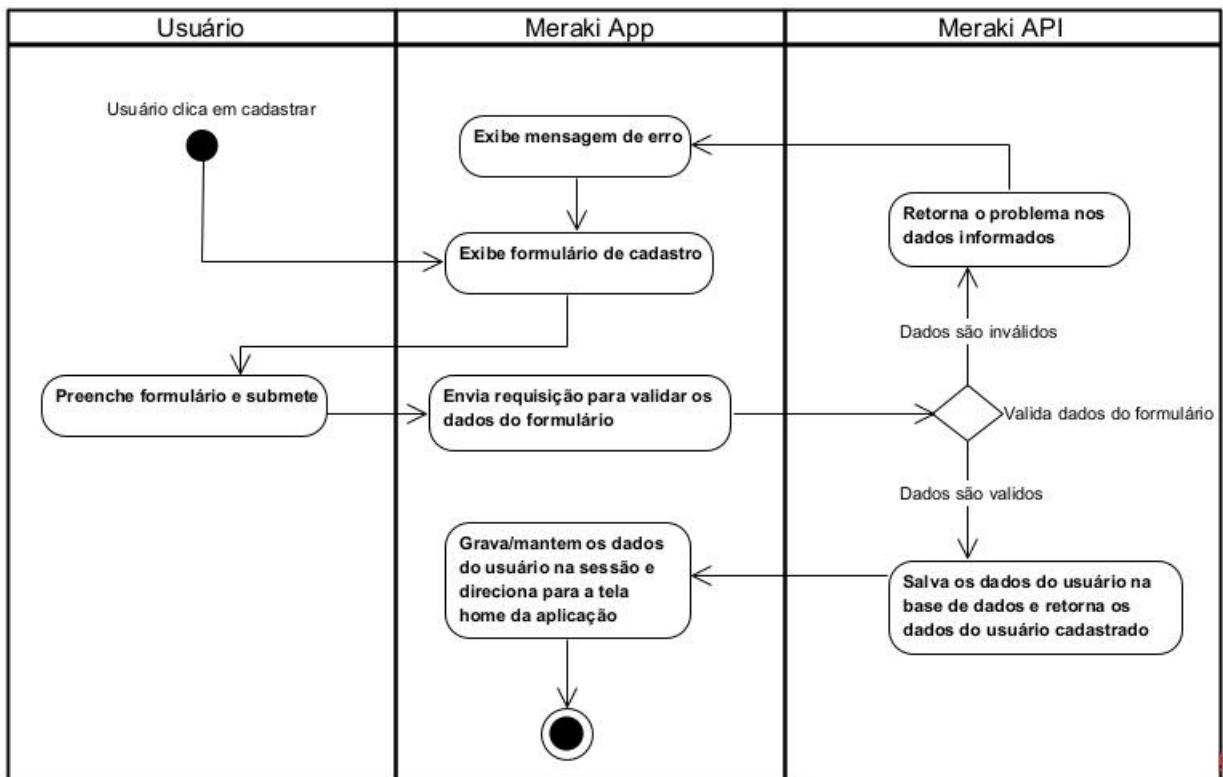


Fonte: Elaborado pelo autor, 2021

Caso o usuário ainda não possua cadastro no APP, haverá na tela de login um link para acessar o formulário de cadastro, onde se inicia o fluxo de cadastro. Após preencher o formulário de cadastro, o usuário deverá submeter os dados informados. Com a submissão dos dados informados, o APP envia para a API uma requisição, e aguardará a validação dos dados e o retorno da API. Caso os dados sejam inválidos, o APP exibirá um aviso informando o usuário os dados inválidos, e permanecerá na tela do formulário de cadastro, até que os dados fornecidos pelo usuário sejam válidos. Caso os dados sejam validos, a API salvará as informações do usuário em banco e retornará para o APP dentro do *response* um JSON com os dados para serem salvos na sessão do usuário.

Tanto no cadastro quanto no login, após submeter o formulário, caso os dados sejam válidos um novo token é criado para autenticar a sessão do usuário, e é enviado no *response* para o APP. Esse fluxo de cadastro de usuário é observado em detalhes na Figura 44.

Figura 44 – Diagrama de Raia descrevendo o fluxo de cadastro



Fonte: Elaborado pelo autor, 2021

Outro fluxo que pode acontecer antes da autenticação do usuário, é o fluxo de troca de senha. Após abrir o APP, o formulário de login é apresentado. Caso o usuário não se lembre da senha, haverá nessa tela um link para o formulário de troca de senha. Quando o usuário clica nesse link dá-se início ao fluxo de recuperação de senha.

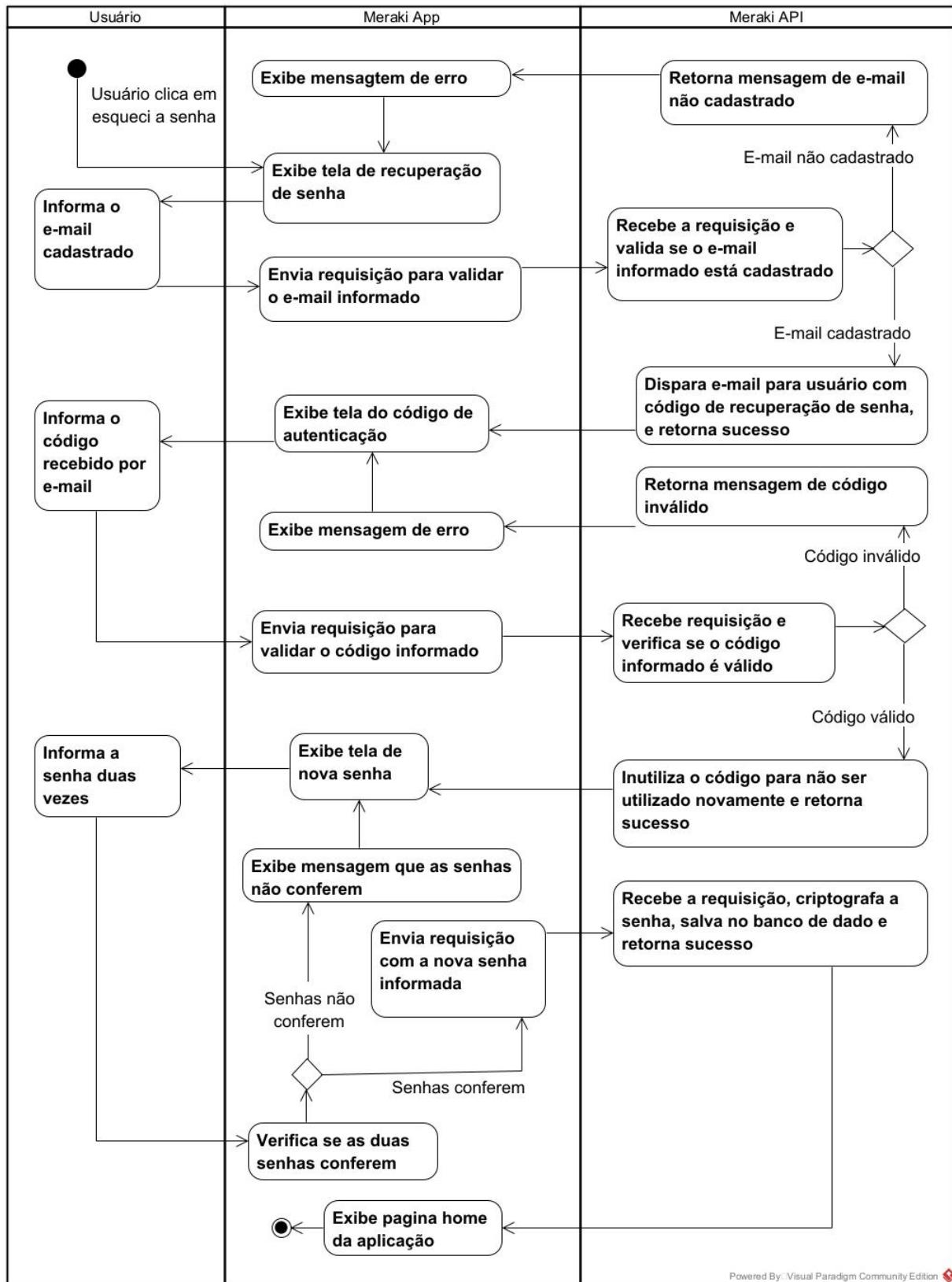
Um formulário é aberto solicitando que o usuário informe o e-mail cadastrado. Após o usuário informar o e-mail e submeter, uma requisição é enviada à API, que fará uma validação se o e-mail informado está contido na base de usuários. Caso não, retornará para o APP que informará para o usuário que o e-mail informado é inválido. Esse processo se repetirá até o usuário informar um e-mail válido.

Após a API validar o e-mail cadastrado, irá disparar um e-mail para o usuário com um código único, criptografado e com prazo de validade, para o usuário informar

no APP. O APP ao receber o retorno da validação do e-mail pela API, irá exibir a tela de validação do código, onde o usuário deverá informar o código que recebeu por e-mail e submeter o formulário. Novamente o APP irá enviar uma requisição à API que validará se o código confere com o armazenado em banco, e se está dentro da validade. Caso o código esteja vencido, o APP exibirá uma mensagem informando o usuário que o código venceu, e que o usuário pode solicitar um novo código clicando no link "Reenviar código".

Após a API validar o código, o mesmo é inutilizado para não ser usado novamente, e retorna sucesso para o APP, que redirecionará o usuário para a tela de nova senha, onde o usuário deverá digitar uma nova senha e repeti-la. A nova senha é enviada para a API, que fará uma nova verificação se a senha confere e se está dentro dos padrões. Caso haja algum problema com as senhas, uma mensagem é exibida para o usuário, que terá que informa-las novamente. Quando as senhas estiverem validadas, a API retornará para o APP no *response* o JSON com as informações do usuário para serem salvos na sessão, bem como o token de autenticação, assim como acontece no fluxo de login e cadastro. Esse fluxo de recuperação de senha pode ser observado na Figura 45.

Figura 45 – Diagrama de Raia descrevendo o fluxo de recuperação de senha



Fonte: Elaborado pelo autor, 2021

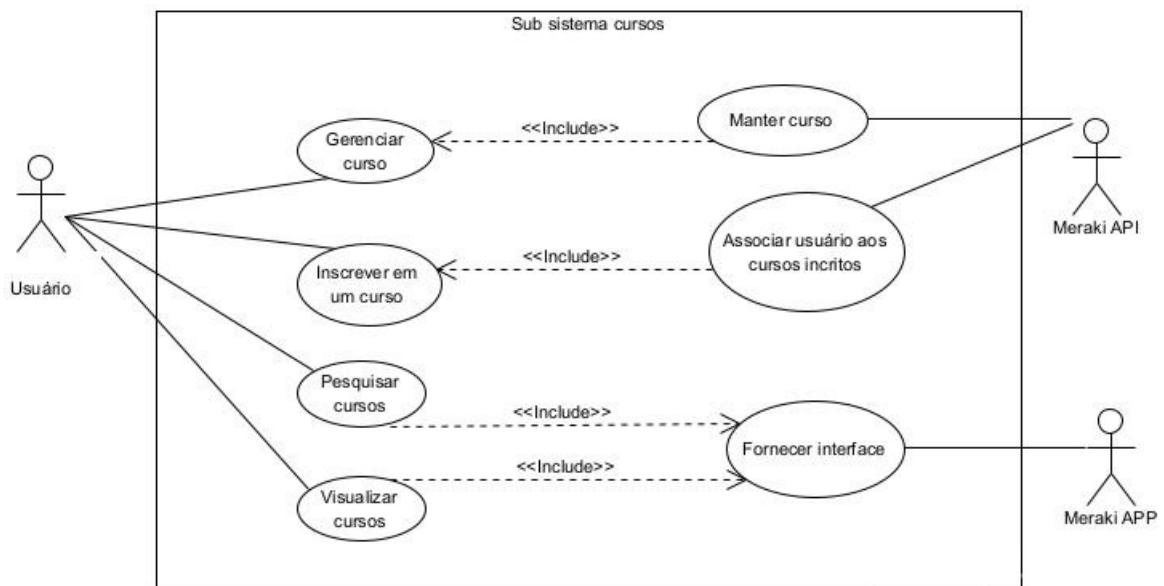
Para representar o principal subsistema da aplicação, que é o subsistema que gira em torno dos cursos, foi escolhido representá-lo utilizando o diagrama de caso de uso, para ter uma visão geral, simples e de fácil entendimento dos procedimentos envolvidos, o que é justamente o objetivo desse diagrama.

São 3 atores envolvidos nesse subsistema. O Usuário, o APP e a API. O ator APP fica encarregado de fornecer a interface para que o ator Usuário realize os procedimentos de pesquisa e visualização de cursos. O usuário também pode se inscrever em cursos, e o ator API se encarregará de persistir essa associação entre usuário e curso.

A API também realizara o procedimento de persistência e controle dos cursos publicados, editados e excluídos pelo usuário, que se encarregará da gerência e publicação dos cursos.

Os procedimentos de cada um dos atores mencionados podem ser observados no diagrama da Figura 46. os relacionamentos entre os procedimentos marcados pelo «Extend» indica um procedimento que pode ser realizado. Já os relacionamentos marcados pelo «include» indicam uma obrigatoriedade na execução de um procedimento para que o outro seja executado.

Figura 46 – Diagrama de caso de uso descrevendo os procedimentos do subsistema Cursos

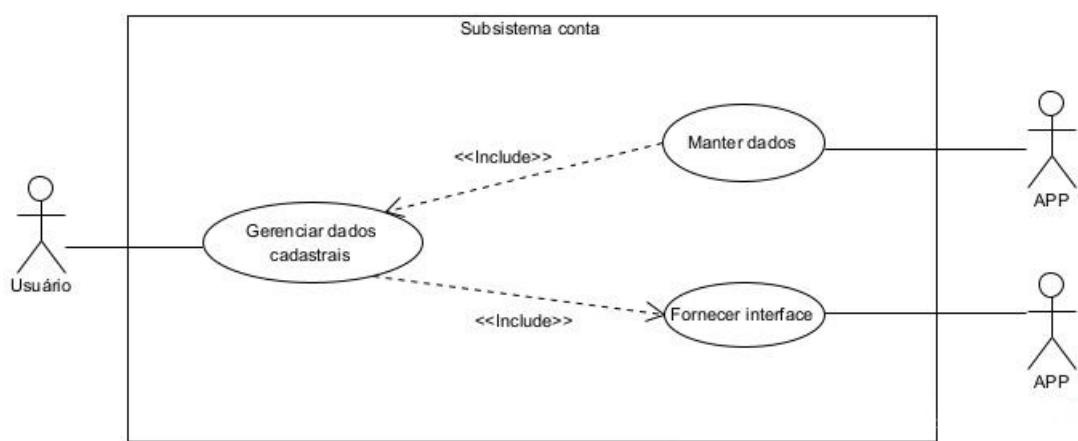


Fonte: Elaborado pelo autor, 2021

Outro subsistema do APP Meraki, envolve as operações de cadastro, edição, e exclusão de dados relacionados a conta do usuário. Esses dados são acessíveis pelo usuário através do menu “conta” do APP Meraki.

O usuário ao abrir a tela de configurações poderá alterar seus dados e submeter o formulário. O APP será responsável por enviar uma requisição para a API com as alterações do usuário, que validará os dados e irá persistir em banco. O fluxo desse procedimento pode ser visto na Figura 47

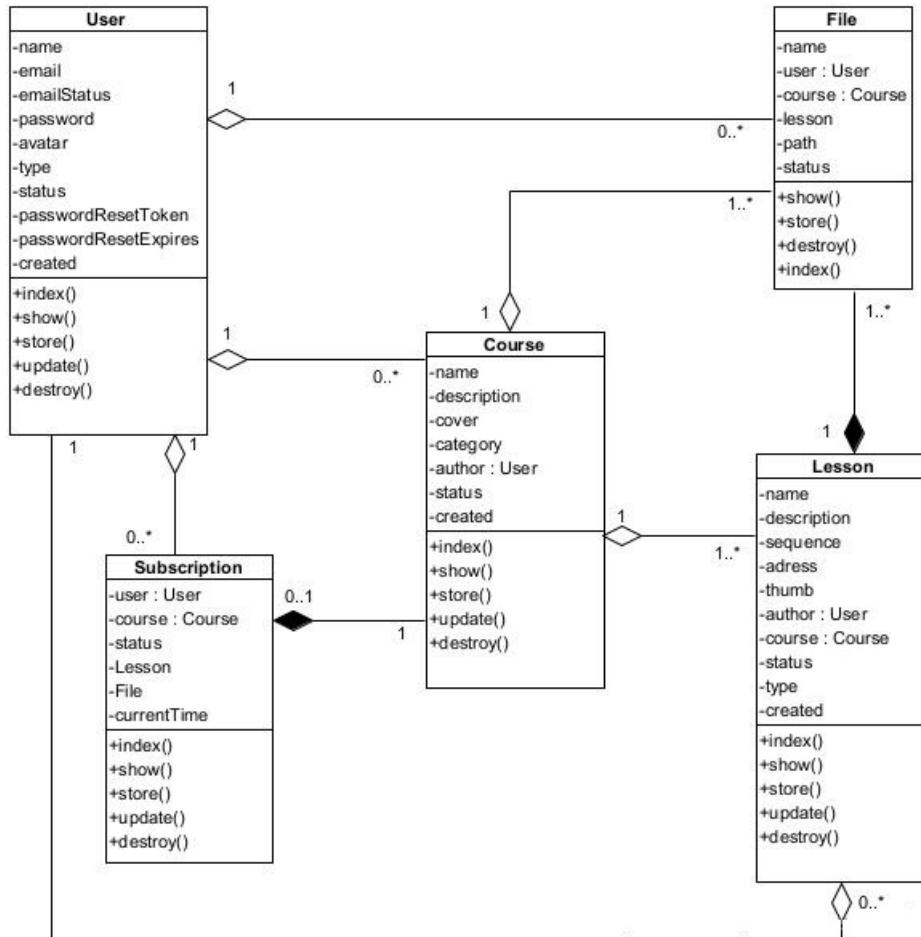
Figura 47 – Diagrama de caso de uso descrevendo os procedimentos de atualização de dados cadastrais



Fonte: Elaborado pelo autor, 2021

Para representar tanto as entidades da API quanto o banco de dados, optou-se por utilizar o diagrama de classe da UML, visto que foi utilizado um banco de dados não relacional. Nesse diagrama, as entidades User, Course, Lesson, File e Subscription são exibidas com seus atributos e operações, bem como os relacionamento de agregação e composição, e a cardinalidade de cada relacionamento.

Figura 48 – Diagrama de classe da UML representando as entidades da aplicação



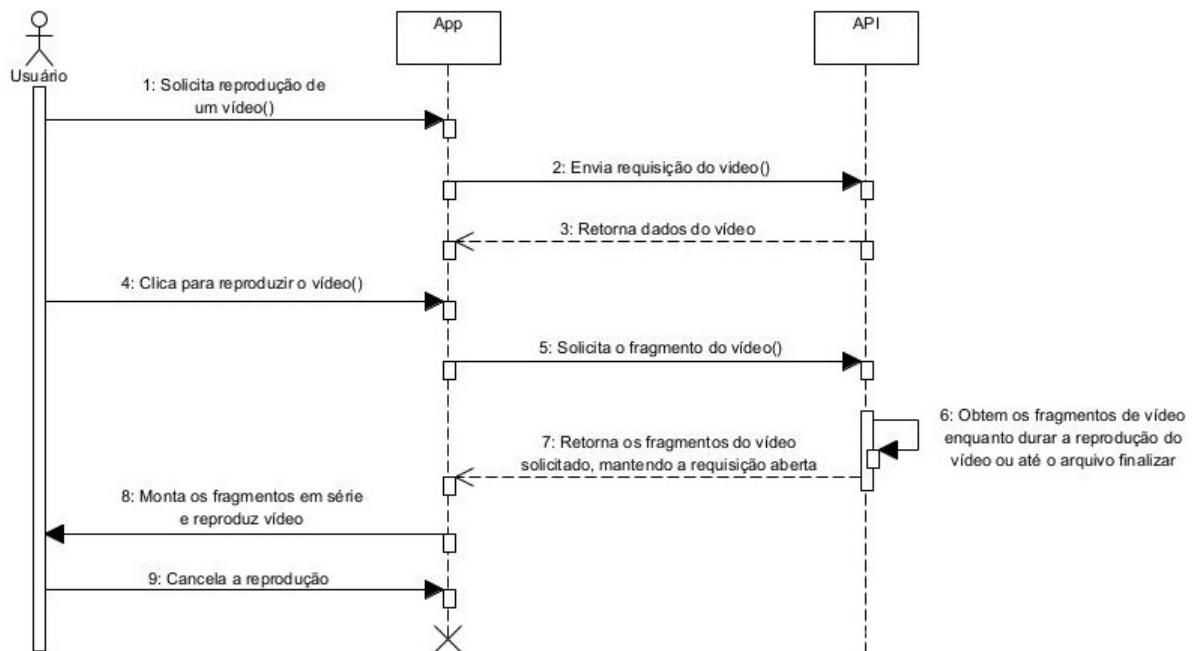
Fonte: Elaborado pelo autor, 2021

Para finalizar os diagramas necessários para documentar a aplicação, utilizou-se de um diagrama de sequência para representar o fluxo de *streaming* de vídeo. Nesse diagrama, o fluxo é iniciado pelo ator usuário, que solicita a reprodução de um vídeo. O APP recebe essa solicitação e envia uma requisição para a API, que irá retornar no *response* os dados gerais do vídeo. O usuário, ao clicar em reproduzir o vídeo, fará com que o APP envie uma mensagem para a API solicitando os fragmentos do vídeo, e passando o tempo inicial do fragmento (*range*) solicitado pelo usuário e ou APP. A API irá obter o intervalo do vídeo, conforme o *range* solicitado pelo APP e ou usuário.

A requisição continuará aberta, de modo a conseguir continuar enviado os fragmentos de vídeo conforme o *player* avança a reprodução. Isso ocorre automaticamente, onde o próprio *player* solicita um novo fragmento do vídeo, antes do fragmento atual finalizar a reprodução, dando a impressão ao usuário que se trata de um arquivo

de vídeo contínuo, quando na verdade são fragmentos de vídeos em série. Isso é possível utilizando o código de status 206 do HTTP, que indica conteúdo parcial, fazendo o método HTTP manter a conexão aberta para que a requisição continue sendo transmitida. Após a reprodução do vídeo chegar ao fim, ou se o usuário cancelar a reprodução do vídeo, a requisição é terminada. Esse fluxo é observado na Figura 49.

Figura 49 – Diagrama de sequência representando o fluxo de *streaming* de vídeo



Fonte: Elaborado pelo autor, 2021

Este fluxo de *streaming* dos vídeos se inicia no aplicativo, onde o script responsável por renderizar a página do *player*, realiza uma requisição para a rota de *stream* da API, enviando na requisição, o *token* e o id do vídeo solicitado pelo usuário, conforme Figura 50.

Figura 50 – Solicitação do vídeo pelo *player* do APP Meraki.

```

EXPLORER          ... course.js ✘
OPEN EDITORS
MERAKI
.env_example      u
.eslintrc.json
.gitignore
index.js
package.json      M
readme.md
yaml-error.log
.yaml.lock
Meraki-Web
MerakiApp
__tests__
android
ios
node_modules
src
assets
components
contexts
pages
account
course
courses
Forgot-pass
home
include
Main
reset-password
search
Signin
SignUp
Token-code
routes
services
styles
App.tsx          u
.buckconfig        u

MerakiApp > src > pages > course > course.js > onBuffer
1 import * as React from 'react';
2 import { View, Text, StyleSheet } from 'react-native';
3 import VideoPlayer from 'react-native-video-controls';
4
5 // styles
6 import mainStyles from '../styles/mainStyles';
7
8
9 function course({ navigation, courseId }) {
10
11     let url = localStorage.getItem('@meraki-app/url-api-stream');
12     let token = localStorage.getItem('@meraki-app/token');
13     let src = `${url}/${courseId}/${token}`;
14
15     return (
16         <View style={mainStyles.mainGrid}>
17             <Text style={mainStyles.text}>Nome do curso</Text>
18             <VideoPlayer
19                 source={{ uri: src }}
20                 ref={(ref) => {
21                     this.player = ref
22                 }}
23                 onBuffer={this.onBuffer}
24                 onError={this.videoError}
25                 style={styles.backgroundVideo}
26                 tapAnywhereToPause="true"
27                 showOnStart={true}
28             >
29         </View>
30     );
31 }
32
33 > function onBuffer() {
34     ...
35 }
36
37 > function videoError() {
38     ...
39 }
40
41 > var styles = StyleSheet.create({
42     ...
43 });
44
45
46
47
48
49
50
51 export default course;

```

Fonte: Elaborado pelo autor, 2021

A API ao receber a requisição, redirecionará a requisição para a rota de *stream*, conforme visto na Figura 51, a qual inicialmente é interceptada pelo *middleware* “token-Middleware”.

Figura 51 – Encaminhamento da requisição do vídeo para a rota *stream*.

```

EXPLORER          routes.js M X
OPEN EDITORS
MERAKI
  Meraki-API
    assets
    config
    node_modules
  routes
    routes.js M
      src
      templates
    test
    uploads
    utils
      env
      .env_example U
      .eslint.json
      .gitignore
      index.js
    package.json M
      README.md
      .yam-error.log
      .yam.lock
  Meraki-Web S
  MerakiApp
  MerakiMobile
  .gitmodules

routes.js M X
Meraki-API > routes > routes.js ...
1 const express = require("express");
2 const { resolve } = require("path");
3
4 const authController = require("../src/controllers/authController");
5 const userController = require("../src/controllers/userController");
6 const courseController = require("../src/controllers/courseController");
7 const lessonController = require("../src/controllers/lessonController");
8 const subscriptionController = require("../src/controllers/subscriptionController");
9 const fileController = require("../src/controllers/fileController");
10 const streamController = require("../src/controllers/streamController");
11
12 const authMiddleware = require("../src/middlewares/authMiddleware");
13 const uploadMiddleware = require("../src/middlewares/uploadMiddleware");
14 const tokenMiddleware = require("../src/middlewares/tokenMiddleware");
15
16 const routes = express.Router();
17
18 // Root route
19 routes.get("/", (req, res) => res.send({ message: "Você acessou o app Meraki!" }));
20
21 // Auth routes
22 routes.post("/registration", authController.registration);
23 routes.post("/authenticate", authController.authenticate);
24 routes.post("/forgot_password", authController.forgotPassword);
25 routes.post("/check_token", authController.checkToken);
26 routes.post("/reset_password", authController.resetPassword);
27
28 // Allow public access to assets folder
29 routes.use("/static", express.static(resolve(__dirname, "..", "assets")));
30
31 // Stream routes
32 routes.get("/stream/:id/:authorization", tokenMiddleware, authMiddleware, streamController.stream); // Line highlighted by a red oval
33
34 // Use the auth middleware to below routes
35 routes.use(authMiddleware);
36
37 // User routes
38 routes.get("/users", userController.index);
39 routes.post("/users", userController.store);
40 routes.get("/users/:id", userController.show);
41 routes.put("/users/:id", userController.update);
42 routes.delete("/users/:id", userController.destroy);
43
44 // Course routes
45

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```

Você foi conectado ao banco de dados com sucesso!
[nodemon] restarting due to changes...
[nodemon] starting `node index.js`
Iniciando a API em ambiente dev

```

Fonte: Elaborado pelo autor, 2021

O *middleware* “tokenMiddleware” é responsável por deixar o token no seu lugar (*headers*), interceptando a requisição e verificando se existe o *token* dentro dos cabeçalhos da requisição e liberar a requisição para o próximo *middleware* se encontrar o *token*. Caso não encontre o *token* no *header*, ele irá verificar se ele está informado como parâmetro da requisição, colocando o *token* no *header* e prosseguindo a requisição para o próximo *middleware*. Caso não encontre o *token* nem no *header* nem nos parâmetros da requisição, ele não fará nenhuma manipulação na requisição, apenas deixará seguir para o próximo *middleware*, conforme visto na Figura 52.

Figura 52 – *Token middleware.*

```

EXPLORER ... routes.js M tokenMiddleware.js M
Meraki-API > src > middlewares > tokenMiddleware.js ...
1 module.exports = async (req, res, next) => {
2   ...
3   if (req.headers.authorization) {
4     return next();
5   }
6   if (req.params.authorization) {
7     req.headers.authorization = `bearer ${req.params.authorization}`;
8   }
9   return next();
10 };
11

```

Fonte: Elaborado pelo autor, 2021

O próximo *middleware* que intercepta a requisição, é o “*middlewareAuth*”, que é responsável por verificar se no cabeçalho da requisição existe o *token* e se ele é válido. Se não houver o *token* informado no *headers*, ele manipulará o *response* para retornar o *status code 401* informando o erro “Token não informado”. Se o *token* existir dentro do *headers*, ele irá verificar se o *token* é válido. Caso o *token* for válido, irá prosseguir com a requisição. Caso não for válido, irá manipular o *response*, retornando o *status code 400*, informando que o *token* não é válido.

Figura 53 – *Token authentication.*

```

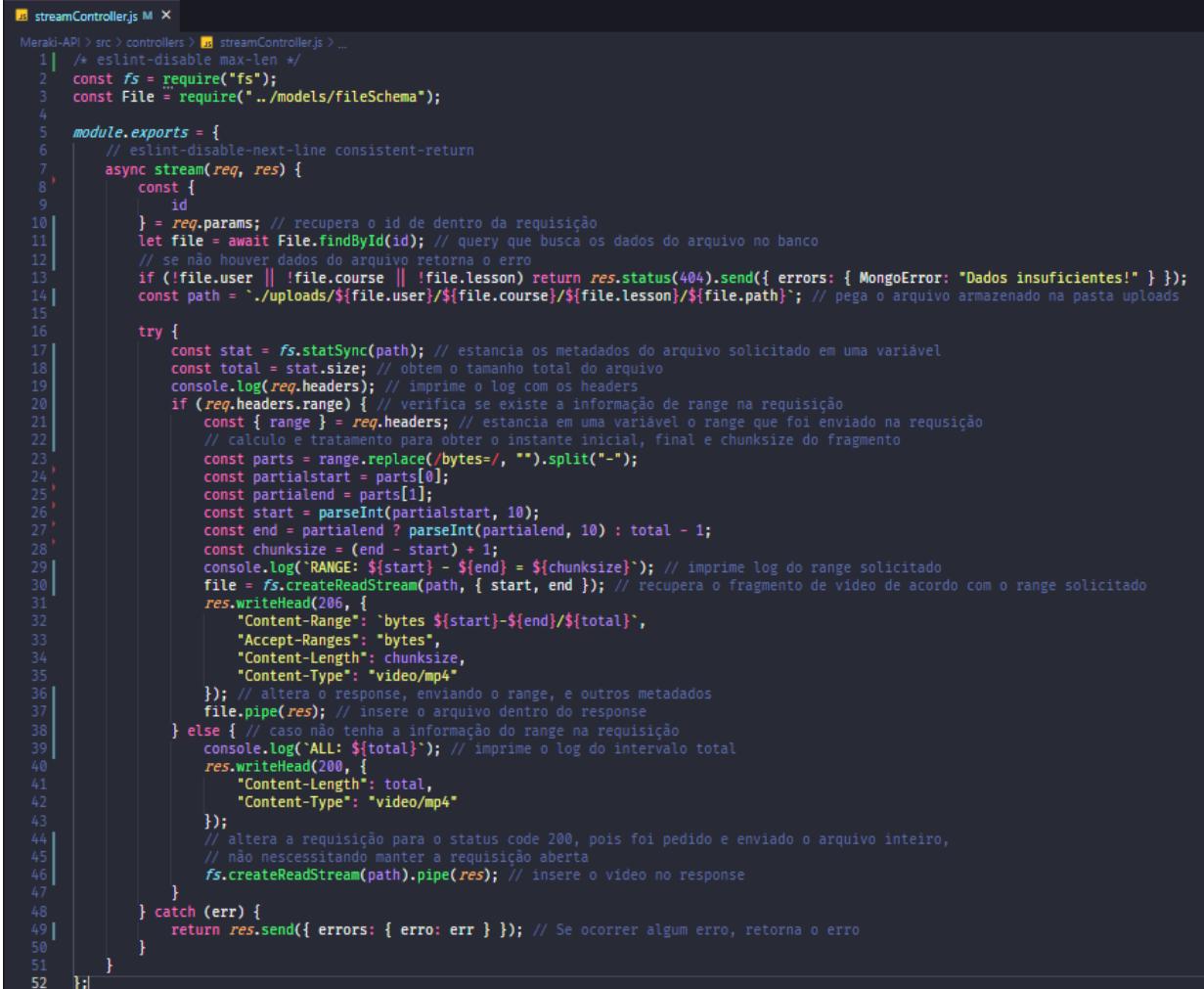
EXPLORER ... routes.js M tokenMiddleware.js M authMiddleware.js X
Meraki-API > src > middlewares > authMiddleware.js ...
1 const jwt = require("jsonwebtoken");
2 const { promisify } = require("util");
3
4 const config = require("../config/config");
5
6 module.exports = async (req, res, next) => {
7   const authHeader = req.headers.authorization;
8
9   if (!authHeader) return res.status(401).json({ errors: { MongoError: "Token não informado!" } });
10
11   const [, token] = authHeader.split(" ");
12
13   try {
14     const decoded = await promisify(jwt.verify)(token, config.TOKEN_SECRET);
15     req.id = decoded.id;
16     return next();
17   } catch {
18     return res.status(400).json({ errors: { MongoError: "Token inválido!" } });
19   }
20 };
21

```

Fonte: Elaborado pelo autor, 2021

Depois de validado o *token*, a requisição chega no *controller de stream*, responsável por interceptar a requisição, obter os dados do arquivo salvos em banco, recuperar o fragmento do vídeo, inserir o fragmento no *response* junto com outros metadados, retornar erros caso ocorra algum, conforme exemplificado na Figura 54

Figura 54 – Controller streams



```

1  /* eslint-disable max-len */
2  const fs = require('fs');
3  const File = require("../models/fileSchema");
4
5  module.exports = {
6    // eslint-disable-next-line consistent-return
7    async stream(req, res) {
8      const {
9        id
10       } = req.params; // recupera o id de dentro da requisição
11       let file = await File.findById(id); // query que busca os dados do arquivo no banco
12       // se não houver dados do arquivo retorna o erro
13       if (!file.user || !file.course || !file.lesson) return res.status(404).send({ errors: { MongoError: "Dados insuficientes!" } });
14       const path = `./uploads/${file.user}/${file.course}/${file.lesson}/${file.path}`; // pega o arquivo armazenado na pasta uploads
15
16      try {
17        const stat = fs.statSync(path); // instancia os metadados do arquivo solicitado em uma variável
18        const total = stat.size; // obtém o tamanho total do arquivo
19        console.log(req.headers); // imprime o log com os headers
20        if (req.headers.range) { // verifica se existe a informação de range na requisição
21          const { range } = req.headers; // instancia em uma variável o range que foi enviado na requisição
22          // cálculo e tratamento para obter o instante inicial, final e chunksize do fragmento
23          const parts = range.replace(/bytes=/, "").split("-");
24          const partialstart = parts[0];
25          const partialend = parts[1];
26          const start = parseInt(partialstart, 10);
27          const end = partialend ? parseInt(partialend, 10) : total - 1;
28          const chunkszie = (end - start) + 1;
29          console.log(`RANGE ${start} - ${end} = ${chunkszie}`);
30          file = fs.createReadStream(path, { start, end }); // recupera o fragmento de vídeo de acordo com o range solicitado
31          res.writeHead(206, {
32            "Content-Range": `bytes ${start}-${end}/${total}`,
33            "Accept-Ranges": "bytes",
34            "Content-Length": chunkszie,
35            "Content-Type": "video/mp4"
36          }); // altera o response, enviando o range, e outros metadados
37          file.pipe(res); // insere o arquivo dentro do response
38        } else { // caso não tenha a informação de range na requisição
39          console.log(`ALL: ${total}`); // imprime o log do intervalo total
40          res.writeHead(200, {
41            "Content-Length": total,
42            "Content-Type": "video/mp4"
43          });
44          // altera a requisição para o status code 200, pois foi pedido e enviado o arquivo inteiro,
45          // não necessitando manter a requisição aberta
46          fs.createReadStream(path).pipe(res); // insere o vídeo no response
47        }
48      } catch (err) {
49        return res.send({ errors: { erro: err } });
50      }
51    }
52  };

```

Fonte: Elaborado pelo autor, 2021

No terminal, é possível observar o *log* dos ranges solicitados cada vez que o usuário clica na barra de progresso do *player do vídeo*, conforme visto na Figura 55.

Figura 55 – *Ranges stream*

```

roger@LAPTOP-0RD2DAOH MINGW64 /c/Git/Meraki/Meraki-API (master)
$ npm start

> meraki@1.0.0 start C:\Git\Meraki\Meraki-API
> node index.js

Iniciando a API em ambiente dev
Você foi conectado ao banco de dados com sucesso!
RANGE: 557056 - 65474780 = 64917725
RANGE: 131072 - 65474780 = 65343709
RANGE: 19693568 - 65474780 = 45781213
RANGE: 28606464 - 65474780 = 36868317
RANGE: 34504704 - 65474780 = 30970077
RANGE: 40763392 - 65474780 = 24711389
RANGE: 45842432 - 65474780 = 19632349
RANGE: 48824320 - 65474780 = 16650461
RANGE: 54067200 - 65474780 = 11407581
RANGE: 58982400 - 65474780 = 6492381
RANGE: 62062592 - 65474780 = 3412189
|

```

Fonte: Elaborado pelo autor, 2021

4.2 DESENVOLVIMENTO BACK-END

Para desenvolver o *back-end* da aplicação, primeiramente foi criado repositórios no GIT HUB, feito o clone desses repositórios localmente e iniciado o projeto em NodeJS. Todo o projeto foi desenvolvido em cima da versão 12.18.3 do NodeJS.

O editor de código utilizado foi o Visual Studio Code da Microsoft, devido a suas facilidades para instalar complementos junto ao editor, bem como a vantagem de possuir terminal integrado.

Foi iniciado o projeto utilizando o gerenciador de pacotes Yarn, devido a sua maior velocidade se comparado com seu concorrente principal, conforme já descrito na subseção 2.3.2. Para isso bastou executar no terminal, dentro da pasta clonada do projeto o comando *yarn init*.

Este comando criou o arquivo *package.json* e também a pasta *node-modules*, já com as configurações e dependências escolhidas durante a criação do projeto.

Também foram instaladas outras dependências que se viu necessário durante o desenvolvimento da API. As dependências instaladas para a API são divididas em duas partes dentro do arquivo *package.json*, as dependências da própria aplicação, e as dependências de desenvolvimento. A lista das dependências da aplicação pode ser observada abaixo:

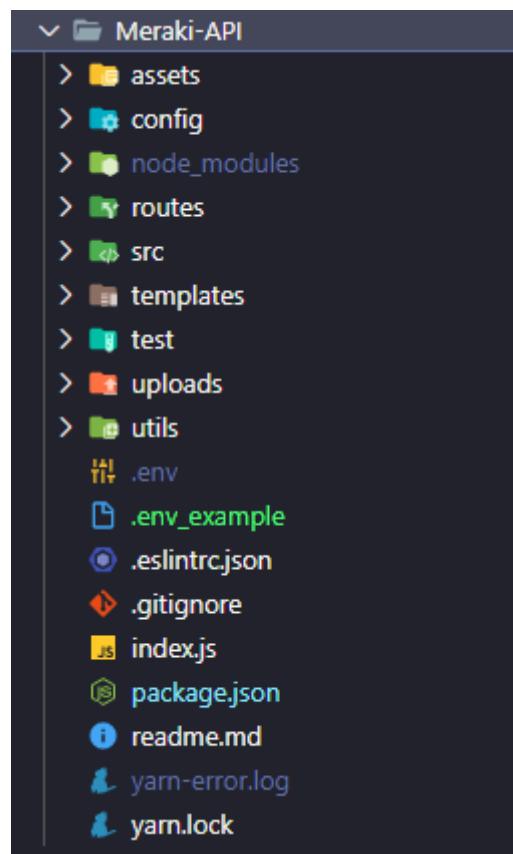
- **bcrypt** - Utilizado para criptografar e descriptografar dados sensíveis salvos em banco e códigos de autenticação.
- **dotenv** - Utilizado para criar variáveis de configurações dos ambientes de produção e desenvolvimento, permitindo maior segurança dos dados contidos nessas configurações de ambientes, como por exemplo, usuário e senha de banco de dados, usuário e senha de e-mail, entre outros.
- **ejs** - *Template engine* utilizada para interpolar e escapar valores nos templates de e-mail enviado pela API.
- **express** - Framework utilizado para navegação por rota (roteamento da API), configuração, manipulação, recebimento e envio de *requests* e *responses*.
- **jsonwebtoken** - Utilizado para tratar o token de autenticação do usuário.
- **mongoose** - Utilizado para realizar as buscas (*querys*) no banco de dados MongoDB e utilizando a sintaxe em JavaScript.
- **multer** - Utilizado para tratar as requisições do tipo *multipartformdata* com arquivos.
- **nodemailer** - utilizado para realizar o envio dos e-mails.

Algumas dependências são restritas para uso durante o desenvolvimento da aplicação, e não é uma dependência da aplicação em si. Abaixo é observado a lista de dependências de desenvolvimento:

- **eslint** - Utilizado identificar padrões de código fora dos padrões de mercado automaticamente.
- **nodemon** - Utilizado para subir o servidor automaticamente após salvar.
- **prettier** - Utilizado para auto formatar o código, conforme padrões estabelecidos nas configurações do Eslint.

Sobre a estrutura de pastas do projeto, foi organizado da seguinte forma, como visto na Figura 56.

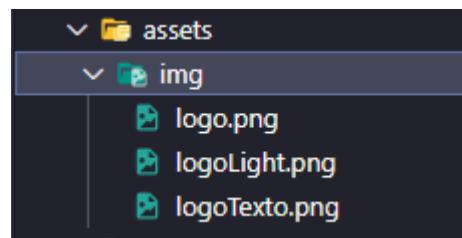
Figura 56 – Estrutura de pastas do projeto



Fonte: Elaborado pelo autor, 2021

Na raiz do projeto, existe uma pasta chamada *assets*, responsável por armazenar arquivos de *assets*, como imagens por exemplo. Por enquanto dentro desta pasta existe apenas uma subpasta chamada *imgs* que armazena os logos da aplicação, conforme visto na Figura 57

Figura 57 – Estrutura da pasta *assets*

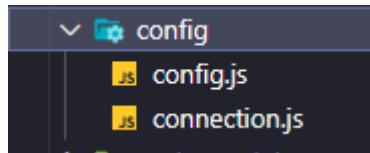


Fonte: Elaborado pelo autor, 2021

A próxima pasta, a *config*, armazena arquivos de configuração. Como pode ser

observado na Figura 58, existem atualmente apenas dois arquivos dentro dessa pasta, o *config.js* responsável por importar as configurações do Dotenv sobre o ambiente, e o *connection.js* responsável por realizar a conexão com o banco de dados.

Figura 58 – Estrutura da pasta *config*

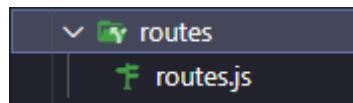


Fonte: Elaborado pelo autor, 2021

Na sequência, é observado a pasta *node modules*, onde ficam todas as dependências do projeto, e as dependências das dependências. Todas controladas pelo arquivo *package.json* e pelo *yarn.lock*.

Outra pasta que é vista na estrutura do projeto, é a pasta *routes*, vide Figura 59, a qual é responsável por armazenar os arquivos de rotas da aplicação. Por conta de a aplicação não possuir muitas rotas, um único arquivo de rota foi suficiente para essa função, porém caso necessário, pode ser feia divisões em vários arquivos com rotas agrupadas.

Figura 59 – Estrutura da pasta *routes*

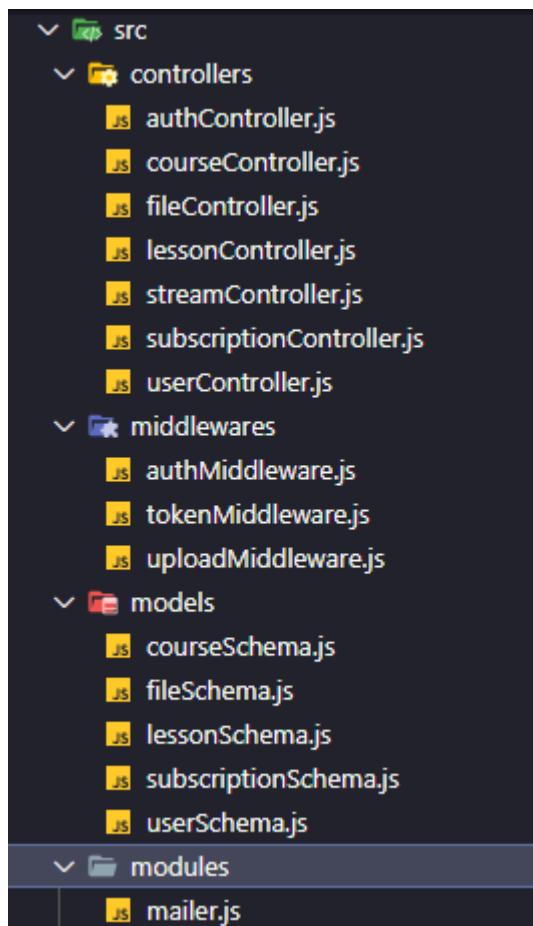


Fonte: Elaborado pelo autor, 2021

Na pasta *src*, foi armazenado o código da aplicação propriamente dita. Essa pasta possui algumas subpastas para melhor controle do da aplicação, garantindo escalabilidade e manutenibilidade do código. A subpasta *controllers* é responsável por armazenar os *controllers* das entidades da aplicação. A subpasta *middlewares* armazena alguns *middleware* necessários para tratar algumas questões no recebimento e retorno das requisições, como verificação do *token*, e tratamento dos arquivos enviados para upload. Na subpasta *models*, ficam os modelos, com as propriedades dos arquivos do banco, bem como suas regras. Por fim, a subpasta *modules* foi criada para armazenar arquivos com funções genéricas que são chamadas em vários *controllers* diferentes, a fim de realizar o reaproveitamento de código, como é o caso do envio do

e-mail, o qual é enviado por diversas partes da API. As subpastas são observadas na Figura 60.

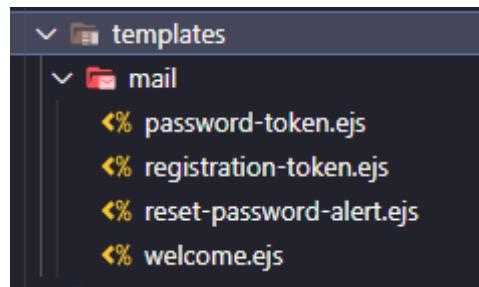
Figura 60 – Estrutura da pasta src



Fonte: Elaborado pelo autor, 2021

Voltando para a raiz do projeto, temos na sequência a pasta *templates*, o qual foi criada para armazenar *templates* em geral, como os *templates* dos e-mails enviados. A estrutura da pasta *templates* é observada na Figura 61.

Figura 61 – Estrutura da pasta *templates*



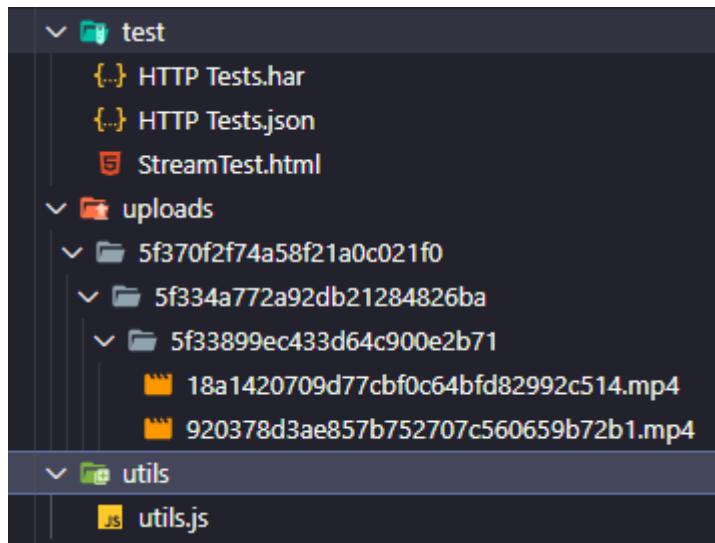
Fonte: Elaborado pelo autor, 2021

As próximas pastas da estrutura do projeto são na sequência as pastas *test*, *uploads* e *utils*. A pasta *test* basicamente contém alguns arquivos de testes feitos em uma ferramenta de testes de API (Insomnia), o qual está armazenado toda as configurações feitas para realizar os testes.

Já a pasta *uploads*, contém uma estrutura de subpastas, montada dinamicamente pelo próprio sistema, utilizando os códigos de identificação do usuário e dos cursos para armazenar os arquivos. Essa pasta *uploads* foi armazenada dentro do projeto a caráter de exemplo, pois a mesma poderia estar em um servidor dedicado a armazenamento de arquivos.

A pasta *utils* reúne scripts genéricos utilizados por todo o sistema. Apesar de já ter sido deixado reservado esse local para essa funcionalidade, não se viu até o momento a necessidade de utiliza-lo. Porém, conforme o produto for ganhando escala, a estrutura já estará pronta. Essas três pastas mencionadas podem ser observadas na Figura 62.

Figura 62 – Estrutura das pastas *test*, *upload* e *utils*



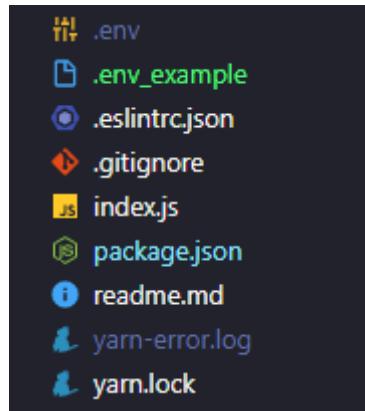
Fonte: Elaborado pelo autor, 2021

Ainda na raiz do projeto, existem alguns arquivos importantes para serem mencionados, conforme visto na Figura 63. O arquivo *.env*, é o arquivo de configuração das variáveis do ambiente de produção e desenvolvimento, o qual armazena os dados de login do banco de dados, e-mail entre outros. Esse arquivo não é versionado junto ao Git Hub, pois tem informações sensíveis. Porém, para que o repositório do projeto possa ter um exemplo de quais variáveis devem ser configuradas para executar o projeto, foi criado um outro arquivo chamado *.env example*, que este sim é versionado junto ao Git Hub como exemplo de configuração de ambiente.

Existe também o arquivo *.gitignore*, o qual foi mencionado nele tudo o que não é preciso versionar para o Git Hub, como a pasta *node modules* e o próprio arquivo *.env*. O arquivo *readme.md* presente no projeto, serve para disponibilizar um *how to*, com instruções a quem desejar configurar e executar o projeto. No arquivo *.eslintrc.json*, possui algumas configurações referente a como o Eslint deve tratar algumas padronizações de código. Basicamente é um arquivo JSON, com uma serie de parâmetros.

E para finalizar temos o arquivo principal do projeto, o arquivo *index.js*. Ao executar a API no terminal, o primeiro arquivo a ser consultado é o arquivo *index.js*. Isso porque no arquivo *package.json*, existe uma configuração, para que ao dar *start* na API, esse arquivo seja executado, conforme visto na Figura 64.

Figura 63 – Arquivos da raiz do projeto



Fonte: Elaborado pelo autor, 2021

Figura 64 – Scripts do package.json

```

package.json M X
Meraki-API > package.json > {} scripts
1  {
2    "name": "meraki",
3    "version": "1.0.0",
4    "description": "Back-end API for Meraki app. A online course platform",
5    "main": "app.js",
6    "scripts": [
7      "test": "echo \\\"Error: no test specified\\\" && exit 1",
8      "dev": "nodemon index.js",
9      "start": "node index.js",
10     "build": "node index.js",
11     "lint": "npx eslint . --ext .js --ignore-pattern node_modules",
12     "lintFix": "npx eslint . --ext .js --fix --ignore-pattern node_modules"
13   ],

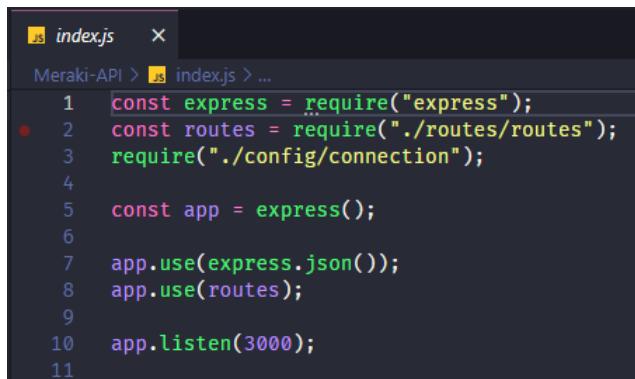
```

A screenshot of a code editor showing the `package.json` file. The `scripts` section is highlighted. The code defines several scripts: `test`, `dev`, `start`, `build`, `lint`, and `lintFix`. The `start` script is set to run `node index.js`.

Fonte: Elaborado pelo autor, 2021

Assim sendo, ao rodar o comando `yarn start`, o Yarn irá executar o script `start` do `package.json`, que irá direcionar para o arquivo principal (`index.js`). O arquivo `index.js`, irá importar o arquivo de rotas da aplicação, o Express e o arquivo de conexão ao banco de dados. Irá instanciar o Express na variável “ap”, e instruir o Express a utilizar as rotas. Por fim, irá executar a aplicação na porta 3000. Todas essas instruções contidas no arquivo `index.js` são observadas na Figura 65

Figura 65 – Arquivo index.js



```

index.js  x
Meraki-API > index.js > ...
1 const express = require("express");
2 const routes = require("./routes/routes");
3 require("./config/connection");
4
5 const app = express();
6
7 app.use(express.json());
8 app.use(routes);
9
10 app.listen(3000);
11

```

Fonte: Elaborado pelo autor, 2021

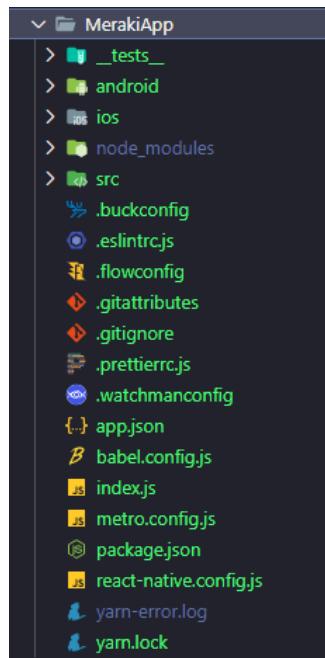
Como o arquivo principal (*index.js*), faz a importação do arquivo de rotas, ele tem acesso a todos os métodos HTTP disponível pelo Express no arquivo de rotas, os quais trabalham com os *controllers* e estes com os *models*. Dessa forma, ao acessar o endereço da API, passando alguma rota válida na URL, o arquivo *index.js* que possui importado todas as rotas da aplicação, através do Express, irá executar a funções destinadas a rota específica, resumindo assim o fluxo de acesso a informação pela API.

4.3 DESENVOLVIMENTO FRONT-END

Para a construção do APP Meraki, foi escolhido utilizar a biblioteca React Native, devido a suas vantagens, como a possibilidade de escrever um único código tanto para a plataforma Android, quanto para a plataforma IOS, e funcionar de forma nativa. Na pasta clonada localmente a partir do repositório do Git Hub, foi executado o comando *react-native init MerakiApp*, para que o React Native se encarregar de criar a estrutura do projeto.

Automaticamente uma chamada *android* será criada na raiz do projeto, com os arquivos necessários para o *build* do APP na plataforma Android. O mesmo acontecerá para a plataforma IOS. Na raiz do projeto também foi criado automaticamente a pasta *node modules*, responsável por armazenar as dependências do projeto. Uma pasta chamada *tests* também foi criada, para manter os testes da aplicação, caso necessário. Por fim, foi criada manualmente uma pasta chamada *src*, para armazenar o código fonte do APP. Além das pastas, o projeto possui na raiz alguns arquivos de configurações também. Toda essa estrutura de pastas mencionada é possível observar na figura Figura 66.

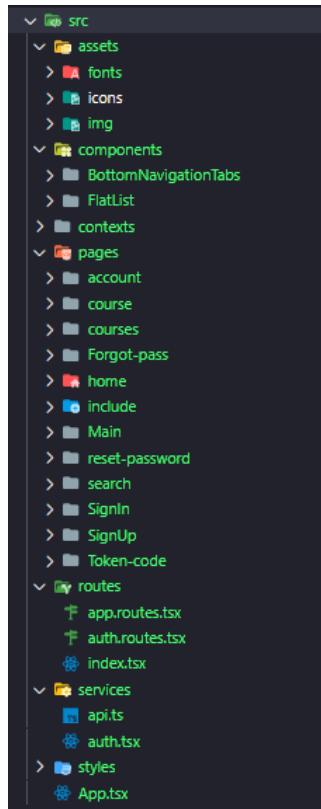
Figura 66 – Estrutura de pastas do aplicativo Meraki



Fonte: Elaborado pelo autor, 2021

A pasta principal (src), contém subpastas para melhor organização e aproveitamento do código. A subpasta *assets* contém arquivos de ícones, fontes e imagens (logos). A subpasta *componentes* contém alguns componentes que são utilizados pela aplicação. Na subpasta *contexts*, ficam os arquivos de contexto da aplicação, o contexto de autenticação, e o contexto geral da aplicação. Em *pages*, ficam todas as páginas da aplicação. Na pasta *routes*, ficam os arquivos de rotas da aplicação, que foram divididos em dois arquivos (rotas de autenticação e rotas do APP). Na pasta *services*, foram criados dois serviços, um que é responsável por armazenar parâmetros utilizados pela aplicação, e outro responsável por disponibilizar as funções de conexão do APP *front-end* com a API *back-end*. Na última pasta, *styles*, ficam os temas e estilos gerais do APP. E por fim, dentro da pasta *src* ainda existe um arquivo principal chamado *app.tsx*, o qual declara o *navigation container* que renderizará as telas da aplicação conforme a rota que for solicitada. Toda essa estrutura de pastas é observada na Figura 67.

Figura 67 – Estrutura da pasta SRC



Fonte: Elaborado pelo autor, 2021

Voltando para a pasta raiz do APP, existe alguns arquivos importantes, como *yarn.lock* e o *package.json* que registram as dependências e suas versões. Mas o arquivo principal da raiz, é o *index.js*. Esse arquivo é o primeiro a ser acionado após executar a aplicação. Ele é responsável por importar o arquivo *app.js* de dentro da pasta *src*, o qual tem acesso as rotas e declara o *navigation container* que renderiza todas as telas.

O projeto do APP Meraki possui algumas dependências. Assim como o projeto da API Meraki, as dependências do APP também são divididas entre dependências da aplicação e dependências de desenvolvimento. Abaixo a lista dependências do APP:

- **@react-native-community/async-storage** - API native do React Native usada no APP Meraki para armazenar dados (usuário logado, *token* etc.) localmente no dispositivo do usuário (similar aos *cookies* dos navegadores).
- **@react-navigation/native** - Biblioteca usada para navegação entre telas e transporte de informações entre telas.

- **@react-navigation/stack** - Biblioteca responsável por empilhar as telas navegadas pela aplicação, permitindo voltar para as telas anteriores.
- **axios** - Cliente HTTP usado no APP Meraki para realizar as requisições a API.
- **react** - Biblioteca usada para criar interfaces, sendo ela uma dependência da biblioteca *react-native*.
- **react-native** - Biblioteca responsável por criar as interfaces do APP de forma nativa.
- **react-native-dash** - Biblioteca que permite desenhar linhas tracejadas ou pontilhadas nas páginas da aplicação.
- **react-native-dismiss-keyboard** - Biblioteca que fornece algumas funções para cancelar/esconder dinamicamente o teclado virtual do dispositivo do usuário. Usado para esconder o teclado nos formulários, após o usuário já ter digitado a quantidade de caracteres necessária em um *input*.
- **react-native-gesture-handler** - Biblioteca responsável por reconhecer gestos do usuário na tela. Utilizado na API Meraki para
- **react-native-masked-text** - Biblioteca que fornece configurações de máscaras para campos de formulários, como e-mail, cep, telefone entre outros.
- **react-native-tab-view** - Biblioteca utilizada para criar visualizações de páginas navegáveis através de abas.
- **react-native-vector-icons** - Biblioteca de ícones em vetor.
- **react-native-video** - Biblioteca que fornece um *player* reprodutor de vídeo, configurável e personalizável.
- **react-native-video-controls** - Biblioteca que permite estender as funcionalidades do react-native-video, permitindo adicionar controles ou personalizar os já existentes.

Abaixo pode ser conferido a lista de dependências de desenvolvimento do APP Meraki:

- **@babel/core** - Biblioteca responsável por transpilar o código escrito em ES6 em código *ECMAScript 5* (ES5) reconhecível pelos navegadores atuais.
- **@babel/runtime** - Pacote utilizado para injetar *helpers* no código transpilado.

- **@react-native-community/eslint-config** - Utilizado para adicionar regras e padrões ao EsLint.
- **eslint** - Utilizado para definir os padrões de código do projeto, conforme definido pelo mercado.
- **jest** - Responsável por transpilar as sintaxes JavaScript para um formato suportado pelo ES5.
- **babel-jest** - Transpilador fornecido pelo Jest, que transforma arquivos TypeScript em JavaScript automaticamente.
- **metro-react-native-babel-preset** - Utilizado para realizar customizações no arquivo *babel.config.js* a transpilação do código.
- **react-test-renderer** - Biblioteca que renderiza componentes React em objetos JavaScript puros.

4.3.1 Considerações do capítulo

O Meraki, trata-se de uma aplicação *mobile* de *streaming* de cursos online que segue os padrões de uma *Application Programming Interface* (API) em seu *back-end*. A construção dessa API, se deu em cima das facilidades do NodeJS, construído sobre o motor V8 do Chrome, o qual é possível desenvolver aplicações poderosas totalmente em JavaScript (KLAUZINSKI; MOORE, 2016)

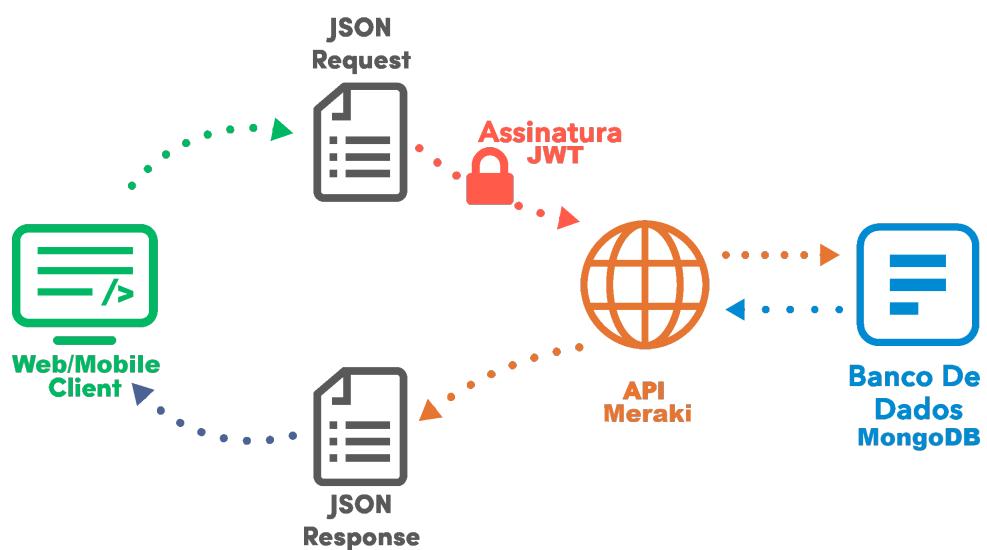
Já o *front-end*, foi construído uma aplicação *mobile* nativa, utilizando React Native, onde apesar de todo o desenvolvimento acontecer em JavaScript, o componente nativo é renderizado de acordo com a plataforma que está a rodar (Android ou IOS), permitindo o uso de recursos nativos do dispositivo como por exemplo o *Global Position System* (GPS) e notificações dos celulares. Conforme afirmam Cruz, Petrucci e Sotto (2018) o JavaScript é uma opção viável para o desenvolvimento *mobile* com performance nativa e aproveitamento de código.

A aplicação desenvolvida para este estudo segue a ideia de desacoplamento, pois sua API é totalmente separada do *front-end*. Uma das vantagens nesse caso é poder criar vários *front-ends* que consomem essa mesma API, permitindo um reuso de código da API para cada plataforma a ser desenvolvida. Além do aplicativo *mobile* que irá consumir a API desse projeto, pode ser criado uma aplicação *web*, uma aplicação *desktop* Linux, Windows ou MacOS que consuma a mesma API. Ou seja, o trabalho será de criar apenas as interfaces gráficas para cada uma dessas plataformas, pois os dados, regras de negócios e segurança são fornecidos pela API.

Segundo KENLON (2020), uma API é uma interface, porém ao invés de ser uma interface para o usuário com botões e elementos gráficos, é uma interface para

uma ou mais aplicações. Ela é responsável por fornecer a essas aplicações todas as informações que elas precisam para funcionar. Uma API consiste em uma aplicação *back-end* que funciona de forma independente, onde o usuário através da interface gráfica (*front-end*) faz requisições (*request*) à API, que receberá a requisição, trabalhará com toda a lógica e regras de negócio necessárias, e retornará uma resposta (*response*) ao *front-end* que por sua vez irá exibir informações ao usuário. É muito comum o uso de chaves de autenticação nas chamadas de requisições entre o *front-end* e a API, para garantir que a requisição vem de uma origem autêntica. O funcionamento do Meraki pode ser exemplificado na Figura 68.

Figura 68 – Funcionamento básico da API Meraki



Fonte: O próprio autor.

5 RESULTADOS E DISCUSSÕES FINAIS

Conforme mencionado na seção 1.2, foi definido alguns objetivos a serem alcançados no projeto. Abaixo é observado a lista desses itens que foram considerados como metas a serem atingidas.

- **Roteamento** - Empregado na API utilizando o Express para criar os *end-points*, e também no APP para navegar entre as páginas com a ajuda do *stack navigator*. Entendendo este requisito como cumprido com sucesso.
- **Autenticação** - Implementado tanto na API quanto no APP, os quais foi empregado o JsonWebToken, cumprindo assim com o objetivo de autenticação do projeto.
- **Disparo de e-mail** - Empregado na API e solicitado APP, o disparo de e-mail é realizado no momento que é solicitado troca de senha, na confirmação código de autenticação e quando a senha é trocada. Atendendo assim esse item do escopo.
- **Criptografia** Utilizada para criptografar dados sensíveis antes de salva-los ao banco, bem como o código de validação enviado pelo e-mail do usuário. Dessa forma, o item criptografia definido no escopo foi atendido.
- **CRUD** Foi implementado o CRUD do usuário, cursos, arquivos, lições e de inscrições, atendente também a este item previsto no escopo.
- **Persistência em banco** - Foi utilizado um banco não relacional, o MongoDB para persistir os dados da aplicação, atendente também a este item.
- **Upload e transmissão de arquivos** - Foi implementado o upload dos arquivos de vídeo aulas e também das imagens de capa dos cursos e dos vídeos, bem como a foto do usuário. Também foi implementado a transmissão dos arquivos de vídeo por **streaming**, utilizando o próprio método HTTP, deixando mais este item do escopo atendido.
- **Artefatos de modelagem e arquitetura** - Foram desenvolvidos alguns artefatos para representar o projeto. Cada um dos artefatos representa parte do projeto no formato de diagrama da UML. Dessa forma, este item também foi atendido.
- **Versionamento** - Conforme estudo realizado, foi escolhido utilizar a tecnologia GIT em conjunto com o Git Hub para versionar o código fonte a medida que ele foi desenvolvido. Este item também foi atendido.

Dado os 9 itens definidos no escopo do projeto como alcançados, defino como alcançado os objetivos desse projeto, pois foi concluído com sucesso a construção de uma aplicação a nível comercial, escalável, manutenível, e utilizando tecnologias já consolidadas no mercado, escolhidas com base nos estudos aqui apresentados.

A aplicação Meraki tem complexidade suficiente para realizar as implementações desejadas e os estudos em cima delas, e foi constatado que o JavaScript é sim uma opção para desenvolvimento *software* capaz de concorrer igualmente com outras tecnologias. Tanto pela facilidade na implementação de scripts utilizando uma linguagem única, e pela facilidade de aprendizado, que resulta em velocidade de entregas, quanto pelo poder por traz da tecnologia, que conta com muitas bibliotecas já prontas para utilizar.

De maneira geral, o desenvolvimento do projeto foi facilitado por conta das muitas documentações e bibliografias já existentes sobre as tecnologias utilizadas que facilitam o entendimento. A ressalva vai para a tecnologia utilizada para o *streaming* de vídeo, a qual foi um desafio, por conta de ser um tipo de *streaming* mais recente, e sem muitas documentações, o que resultou em um esforço e tempo maior dedicado ao aprendizado da tecnologia. Aprendido o funcionamento dessa tecnologia, foi possível implementa-la, e concluo que a sua facilidade de funcionamento para transmissão dos vídeos compensou o tempo e esforço empregado na aprendizagem.

Como pontos positivos do desenvolvimento do projeto e protótipo com as tecnologias escolhidas, considera-se:

- **Vasta documentação** - Tanto em documentações oficiais, quanto em fóruns e sites na web é possível conseguir informações sobre as tecnologias empregadas na construção do Meraki.
- **Uso difundido das tecnologias** - Com altas taxas semanais de downloads, as tecnologias e bibliotecas utilizadas já estão difundidas no desenvolvimento de software, trazendo maior segurança quanto ao emprego de tecnologias as quais já caíram no gosto dos desenvolvedores.
- **Código limpo, escalável e manutenível** - Resultado das abstrações de tecnologias e técnicas utilizadas no projeto.
- **Facilidade de linguagem** - Utilizando apenas JavaScript para implementar os códigos, não se teve dificuldade em questões de sintaxe, ou preocupações com diferentes linguagens.
- **Compatibilidade** - Tanto nos quesitos de plataforma, quanto no quesito de compatibilidade de código, o projeto Meraki foi bem atendido, visto que são atendidas

tanto as plataformas Android, IOS e WEB, podendo inclusive se estende para plataformas de sistemas *desktop*. Isso utilizando apenas JavaScript no desenvolvimento, o que facilita a compatibilidade de dependências e relacionamento entre a própria aplicação, como a transferência de informações via *JSON* entre banco, *API* e *APP*.

- **Velocidade** - Tanto no desenvolvimento da aplicação, devido a outros pontos positivos mencionados, quanto ao desempenho que a tecnologia promete.

Como pontos negativos do desenvolvimento do projeto e protótipo com as tecnologias escolhidas, considera-se

- **Overloading de abstração** - Algumas das tecnologias utilizadas, percebeu-se um certo *overloading* de abstração, o que pode deixar o código não muito fácil de entender inicialmente, podendo requerer alguma pesquisa ou explicação.
- **Atualização de tecnologias** - Devido a velocidade de atualização das tecnologias utilizadas, uma nova versão de uma biblioteca, ou novas funcionalidades de um protocolo podem surgir a qualquer momento, podendo deixar a aplicação defasada caso não seja atualizada, e por ser funcionalidades novas, as documentações podem ser mais escassas.

5.0.1 Trabalhos futuros

Com o intuito de melhorar esta pesquisa e dar continuidade ao projeto, propõe-se como trabalhos futuros:

- Construir o Meraki utilizando outras linguagens e realizar um estudo comparativo entre elas, podendo ser tanto qualitativo, quanto quantitativo, comparando o esforço de se produzir a aplicação, bem como a análise de outras métricas de desempenho, segurança, carga, tráfego, etc.
- Analisar o desempenho do APP utilizando um banco relacional, a fim de medir o desempenho em diversos cenários e constatar se realmente para esta aplicação, o banco não relacional é a melhor opção.
- Alterar o armazenamento dos arquivos para um servidor dedicado como o S3 da Amazon, é algo que também pode ser feito futuramente para melhorar tanto o acesso, velocidade, capacidade de carga e tráfego, quanto a segurança dos arquivos.
- Aplicar testes automatizados (testes unitários, de integração, segurança, escalabilidade, etc.) a fim de realizar uma análise de confiabilidade da aplicação.

- Desenvolver uma versão web da aplicação Meraki.
- Implementar a transmissão dos vídeos utilizando outra tecnologia e protocolo, a fim de compara questões de desempenho.
- Adicionar um chat de mensagens, que permita a usuários e autores dos cursos possam trocar mensagens.
- Adicionar uma área de notificações para que os usuários possam ficar a par das últimas atualizações dos cursos os quais está inscrito.
- Criar um sistema para avaliar, curtir, realizar comentários e compartilhamentos nos cursos e videoaulas, tornando a aplicação mais interativa.
- Implementar a possibilidade de colocar materiais complementares como arquivos de texto, pdf ou planilhas, para acompanhar os cursos caso o autor julgue necessário.
- Realizar a conversão do vídeo para um formato padrão ao fazer upload, e a inclusão de legendas e *caption*.
- Implementar o uso de um novo *token* a cada requisição, salvando o novo *token* recebido no *storage* local do dispositivo para ser utilizado na próxima requisição.

REFERÊNCIAS

- AGGARWAL, S.; VERMA, J. Comparative analysis of mean stack and mern stack. 2018.
- ALM, C. S. Web application for customer-, account-and card register: The enfuce portal. 2019.
- ALMEIDA, F. **Express, realizando upload com multer.** 2017. <<http://cangaceirojavascript.com.br/express-realizando-upload-multer/>>. Accessed: 2020-11-30.
- ANDRADE, A. P. d. **O que é Template Engine?** 2019. <<https://www.treinaweb.com.br/blog/o-que-e-template-engine/>>. Accessed: 2020-11-30.
- ARMEL, J. Web application development with laravel php framework version 4. Metropolia Ammattikorkeakoulu, 2014.
- ARYAL, S. Mern stack with modern web practices. 2020.
- BORSATO, E. C. **Desenvolvendo migrations utilizando Laravel.** 2017. <<https://imasters.com.br/back-end/desenvolvendo-migrations-utilizando-laravel>>. Accessed: 2020-11-30.
- CERON, V. **Como enviar e-mail com Nodemailer.** 2020. <<https://programandosolucoes.dev.br/2020/10/27/enviar-email-com-nodemailer/>>. Accessed: 2020-12-01.
- CHAVES, A. M.; SILVA, G. da. proposta de uma arquitetura de software e funcionalidades para implementação de um ambiente integrado de desenvolvimento para a linguagem php. **I Jornada Científica e VI FIPA do CEFET Bambuí**, v. 31, p. 32–36, 2008.
- CONCEIÇÃO, J. d. Aplicação de metodologias ágeis para desenvolvimento de software: um estudo de caso na empresa alliance software. Universidade Federal de Santa Maria, 2015.
- CRUZ, V. da S.; PETRUCELLI, E. E.; SOTTO, E. C. S. A linguagem javascript como alternativa para o desenvolvimento de aplicações multiplataforma. **Revista Interface Tecnológica**, v. 15, n. 2, p. 39–49, 2018.
- DAIGNEAU, R. **Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services.** Addison-Wesley, 2012. (Addison-Wesley signature series Service design patterns). ISBN 9780321544209. Disponível em: <https://books.google.com.br/books?id=tK3_vB304bEC>.
- DUARTE, L. **Programação assíncrona em Node.js – Callbacks e Promises.** 2020. <https://www.luiztools.com.br/post/programacao-assincrona-em-nodejs-callbacks-e-promises/?gclid=CjwKCAiAv4n9BRA9EiwA30WND7BsqAph_ZCYbqoKJDjyJ6JW0rMeM9syTfq_pKv2saZu9YV9pmzfNxoCbyQAvD_BwE>. Accessed: 2020-11-30.

- FERNANDES, H. M. **Melhores editores de texto (IDEs) para Desenvolvimento Web em 2020.** 2019. <<https://marquesfernandes.com/desenvolvimento/melhores-editores-de-texto-ides-para-desenvolvimento-web-em-2020/>>. Accessed: 2020-11-30.
- GARBADE, M. **15 JavaScript frameworks and libraries.** 2016. <<https://opensource.com/article/16/11/15-javascript-frameworks-libraries>>. Accessed: 2020-08-15.
- GOMES, A.; LOURENÇO, R. Internet live streaming. 2015.
- HEREDIA, J. S.; SAILEMA, G. C. Comparative analysis for web applications based on rest services: Mean stack and java ee stack. **KnE Engineering**, p. 82–100, 2018.
- INÁCIO, B. H. C. et al. Calf manager-sistema para gerenciamento da produção de bezerros. Instituto Federal Goiano, 2020.
- JACOBS, A. Comparison of javascript package managers. 2019.
- KARANJIT, A. Mean vs. lamp stack. 2016.
- KEINÄNEN, M. Creation of a web service using the mern stack. Metropolia Ammatti-korkeakoulu, 2018.
- KENLON, S. **What is an API?** 2020. <<https://opensource.com/resources/what-api>>. Accessed: 2020-09-11.
- KLAUZINSKI, P.; MOORE, J. **Mastering JavaScript Single Page Application Development.** [S.I.]: Packt Publishing Ltd, 2016.
- KOREN, I.; Klamma, R. Peer-to-peer video streaming in html5 with webtorrent. In: SPRINGER. **International Conference on Web Engineering.** [S.I.], 2018. p. 404–419.
- KOSCIANSKI, A.; SOARES, M. dos S. **Qualidade de Software-2ª Edição: Aprenda as metodologias e técnicas mais modernas para o desenvolvimento de software.** [S.I.]: Novatec Editora, 2007.
- LECHETA, R. R. **Web Services RESTful: Aprenda a criar web services RESTful em Java na nuvem do Google.** [S.I.]: Novatec Editora, 2015.
- LIMA, J. F. d.; MOURA, L. P. d. **Análise de desempenho sobre camadas de persistência em banco de dados relacional e não-relacional.** Dissertação (B.S. thesis) — Universidade Tecnológica Federal do Paraná, 2017.
- MACHADO, G. B. et al. Uma arquitetura baseada em web services com diferenciação de serviços para integração de sistemas embutidos a outros sistemas. Florianópolis, SC, 2006.
- MOURA, F. N. **Fetch API e o JavaScript.** 2020. <<https://braziljs.org/artigos/fetch-api-e-o-javascript>>. Accessed: 2020-12-01.
- NGUYEN, H. End-to-end e-commerce web application, a modern approach using mern stack. 2020.

NIEDERAUER, J. **Web Interativa com Ajax e PHP-1^a Edição.** [S.I.]: Novatec Editora, 2007.

OLIVEIRA, M. F. d. **WSDL: Simplifique a integração de dados via Web Service.** 2014. <<https://www.devmedia.com.br/wsdl-simplifique-a-integracao-de-dados-via-web-service/30066>>. Accessed: 2020-09-24.

OLIVEIRA, W. **O universo da programação: Um guia de carreira em desenvolvimento de software.** Casa do Código, 2018. ISBN 9788594188915. Disponível em: <<https://books.google.com.br/books?id=Z1J1DwAAQBAJ>>.

OSTERWALDER, A.; PIGNEUR, Y. **Business model generation: inovação em modelos de negócios.** [S.I.]: Alta Books, 2020.

PALESTINO, C. M. C. Estudo de tecnologias de controle de versões de software. 2015.

PRESSMAN, R. **Software Engineering: A Practitioner's Approach.** McGraw-Hill Education, 2010. (McGraw-Hill higher education). ISBN 9780073375977. Disponível em: <<https://books.google.com.br/books?id=y4k\AQAAIAAJ>>.

RHEINBOLDT, R. L.; CHAVES, R. d. A. Uma abordagem para simplificar utilização de analisadores estáticos e ferramentas de transformação de código. 2018.

SANTOS, J. F. P. d. et al. Mobile streaming-qualidade de experiência. 2010.

SAUDATE, A. **REST: Construa API's inteligentes de maneira simples.** Casa do Código, 2014. ISBN 9788566250985. Disponível em: <<https://books.google.com.br/books?id=u0-CCwAAQBAJ>>.

SAUDATE, A. **SOA aplicado: Integrando com web services e além.** Casa do Código, 2014. ISBN 9788566250978. Disponível em: <<https://books.google.com.br/books?id=OHCCCwAAQBAJ>>.

SILVA, F. Meller-da; MARCIANO, P. de O. Modelo de negócio inovador: a empresa netflix. **Revista Eletrônica Científica do CRA-PR-RECC**, v. 4, n. 1, p. 79–97, 2017.

SILVA, M. B. R. Ajax desmistificado.

SILVA, M. D. d. S. et al. Shivaradar: aplicação para geolocalização de estabelecimentos gamers. Florianópolis, SC., 2020.

SILVA, M. S. **Ajax com Jquery: Requisições AJAX com a simplicidade de JQuery.** [S.I.]: Novatec Editora, 2009.

SILVEIRA, P. et al. **Introdução à Arquitetura de Design de Software: Uma Introdução à Plataforma Java.** Elsevier Brasil, 2011. ISBN 9788535250305. Disponível em: <<https://books.google.com.br/books?id=qQU17gFaofgC>>.

SOMMERVILLE, I. **Engenharia de software.** PEARSON BRASIL, 2011. ISBN 9788579361081. Disponível em: <<https://books.google.com.br/books?id=H4u5ygAACAAJ>>.

SUBRAMANIAN, V. **Pro MERN stack: full stack web app development with Mongo, Express, React, and Node.** [S.I.]: Springer, 2017.

SUDANA, I.; QUDUS, N.; PRASETYO, S. Implementation of phpmailer with smtp protocol in the development of web-based e-learning prototype. In: IOP PUBLISHING. **Journal of Physics: Conference Series**. [S.I.], 2019. v. 1321, n. 3, p. 032027.

TERRA, R.; BIGONHA, R. S. Ferramentas para análise estática de códigos java. **Mônografia, Universidade Federal de Minas Gerais (UFMG), Belo Horizonte**, 2008.