UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ENGENHARIA DE SÃO CARLOS

André Pereira Cavalcante

Desenvolvimento de uma plataforma para compartilhamento de materiais de estudos entre docentes e discentes da USP

São Carlos

2021

André Pereira Cavalcante

Desenvolvimento de uma plataforma para compartilhamento de materiais de estudos entre docentes e discentes da USP

Monografia apresentada ao Curso de Engenharia Mecatrônica, da Escola de Engenharia de São Carlos da Universidade de São Paulo, como parte dos requisitos para obtenção do título de Engenheiro Mecatrônico.

Orientadora: Profa. Dra. Kalinka Regina Lucas Jaquie Castelo Branco

São Carlos

2021

Ficha catalográfica elaborada pela Biblioteca Prof. Dr. Sérgio Rodrigues Fontes da EESC/USP com os dados inseridos pelo(a) autor(a).

Cavalcante, André Pereira
S518d Desenvolvimento de uma plataforma paracompartilhamento de materiais de estudos entredocentes e discentes da USP / André Pereira Cavalcante; orientadora Kalinka Regina Lucas Jaquie Castelo Branco, 2021.

Monografia (Graduação em Engenharia Mecatrônica) -- Escola de Engenharia de São Carlos da Universidade de São Paulo, 2021.

1. Plataforma web. 2. Trabalho de Conclusão de Curso. 3. Node js. 4. Angular. 5. MongoDB. 6. AWS. I. Título.

Eduardo Graziosi Silva - CRB - 8/8907

FOLHA DE AVALIAÇÃO

Candidato: André Pereira Cavalcante

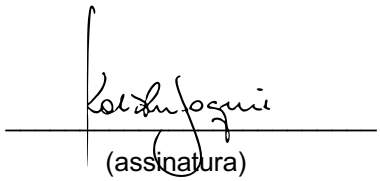
Título: Desenvolvimento de uma plataforma para compartilhamento de materiais de estudos entre docentes e discentes da USP

Trabalho de Conclusão de Curso apresentado à Escola de Engenharia de São Carlos da Universidade de São Paulo Curso de Engenharia Mecatrônica.

BANCA EXAMINADORA

Professor Kalinka R. L. J. Castelo Branco (Orientador)

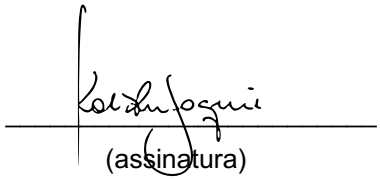
Nota atribuída: 9.5 (Nove, Cinco)



(assinatura)

Professor Maíra M. da Silva

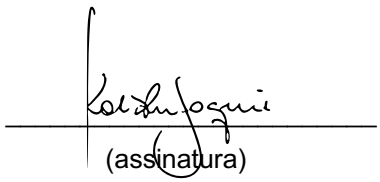
Nota atribuída: 9.5 (Nove, Cinco)



(assinatura)

Mestre Isadora Ferrão

Nota atribuída: 9.5 (Nove, Cinco)



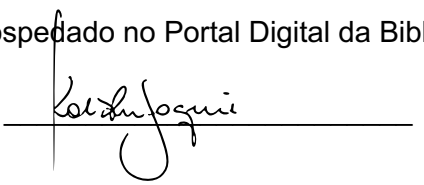
(assinatura)

Média: 9,5 (Nove, Cinco)

Resultado: Aprovado por unanimidade

Data: 02/08/2021.

Este trabalho tem condições de ser hospedado no Portal Digital da Biblioteca da EESC

SIM ☒ NÃO ☐ Visto do orientador 

AGRADECIMENTOS

Agradeço muito à minha família por todo o apoio, à minha mãe Patricia, por todos os sacrifícios, à Georgiana, que há muito tempo cuida de mim como filho, ao meu irmão Fábio Filho, que me inspira a tentar sempre ser uma pessoa melhor, à Anna, que me ofereceu grande apoio durante os últimos anos da trajetória da minha graduação. Agradeço, especialmente, ao meu pai Fábio, que me proporcionou tudo o que foi necessário em educação, na vida, para que eu pudesse estar aqui hoje.

Agradeço às amizades que fiz durante a graduação, sendo alguns dos maiores amigos: Davi, Christian, Marques, Shuji, Túlio e Arthur, que me ajudaram emocionalmente e profissionalmente de inúmeras formas.

Agradeço aos meus amigos Lucas, Flávia, Mateus e Rafa pela amizade de tantos anos, tantos desabafos e conversa jogada fora e à Marina, que contribuiu muito para que eu iniciasse essa jornada.

Agradeço a toda a equipe do CDCC, principalmente, à Silvia Aparecida Martins dos Santos e ao Sidney Carlos Rigo Júnior, que me acolheram e motivaram a enxergar problemas e necessidades na sociedade de uma forma mais ampla e a resolvê-los com mais curiosidade e criatividade.

Agradeço ao João Batista Betoni, que me orientou e salvou diversas vezes durante o período de graduação.

Agradeço também aos professores Maíra Martins da Silva e Rodrigo Nicoletti por me inspirarem em tantos momentos, profissionalmente e pessoalmente.

Agradeço à professora Kalinka, minha orientadora, por todo o tempo, conhecimento, muita paciência, por sempre se mostrar disposta a ajudar e orientar da melhor forma possível.

Agradeço também a todas as outras pessoas que participaram desta trajetória, direta ou indiretamente.

"Nós só podemos ver um pouco do futuro, mas o suficiente para perceber que há muito a fazer."

Alan Turing

RESUMO

ANDRÉ, P. C. **Desenvolvimento de uma plataforma para compartilhamento de materiais de estudos entre docentes e discentes da USP**. 2021. 69p. Monografia (Trabalho de Conclusão de Curso) - Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2021.

Este trabalho tem como objetivo a criação de uma plataforma com a função de guardar em sistema de nuvem anotações, sendo estas, fotos importadas via web e, posteriormente, convertidas em arquivo PDF, dos alunos ou professores da USP. Após o envio dos materiais, qualquer discente ou docente da USP cadastrado no aplicativo poderá acessar o material. Os materiais podem ser buscados com base no conteúdo ou disciplina desejada. Este trabalho tem o intuito de contribuir para a integração dos alunos entre cursos, entre professores e entre unidades da USP e melhorar a eficiência no aprendizado dos alunos. Sendo assim, o sistema permite que alunos que entram na USP possa, desde o início utilizarem materiais já disponibilizados por alunos de anos anteriores, em um local para busca focado e repleto de materiais de estudo. Nesta monografia são descritas as tecnologias e ferramentas utilizadas no projeto, no planejamento, no desenvolvimento e os resultados obtidos. As principais tecnologias utilizadas foram Node.js, Angular, MongoDB e AWS.

Palavras-chave: Plataforma web; Trabalho de Conclusão de Curso; Node js; Angular; MongoDB; AWS.

ABSTRACT

ANDRÉ, P. C. **Development of a platform to share study materials between USP students and professors**. 2021. 69p. Monografia (Trabalho de Conclusão de Curso) - Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2021.

This work aims to create a platform with the function of saving notes in a cloud system, which are photos imported via web and, later, converted into a PDF file, of the students or professors at USP. After sending the materials, any USP student or professor registered in the application will be able to access the material. Materials can be searched based on content or desired subject. This work aims to contribute to the integration of students between courses, between professors and between USP units and improve the efficiency of student learning in the years after the start of use, as they already have a place for focused search of study materials. This work describes the technologies and tools used in the project, planning, development and results obtained. The main technologies used were Node.js, Angular, MongoDB and AWS.

Keywords: Web platform; Node js; Undergraduate Capstone project; Angular; MongoDB; AWS.

LISTA DE FIGURAS

Figura 1 – Funcionamento do Node.js 20
Figura 2 – Exemplo de modelo UML de banco de dados MongoDB 24
Figura 3 – MongoDB em formato de tabela - Coleção Users 24
Figura 4 – MongoDB em formato de tabela - Coleção Notebooks 24
Figura 5 – MongoDB em formato BSON - Coleção Users 24
Figura 6 – MongoDB em formato BSON - Coleção Notebooks 25
Figura 7 – Casos de uso 30
Figura 8 – Visão Visitante 31
Figura 9 – Visão Usuário 31
Figura 10 – Visão Administrador 32
Figura 11 – Modelo UML de classes do banco de dados 33
Figura 12 – Escolha do ambiente Front-end 37
Figura 13 – Fluxo de estudo e utilização do versionamento e do deploy 45
Figura 14 – Migração do banco de dados local para plataforma remota 49
Figura 15 – Escolha do armazenamento remoto das imagens 50
Figura 16 – Escolha entre S3 ou EC2 na AWS 51
Figura 17 – Página de *login* 63
Figura 18 – Página de registro 63
Figura 19 – Página inicial do usuário 64
Figura 20 – Busca e *download* de materiais 64
Figura 21 – Modelo UML Inicial 66

SUMÁRIO

1 INTRODUÇÃO 17
1.1 Motivação e Contextualização 17
1.2 Objetivo 17
1.3 Organização 17

2 REVISÃO DA LITERATURA 19
2.1 *Responsive web design* 19
2.2 *Back-end* 20
2.2.1 Node Js 20
2.2.2 Módulos do Node.Js 21
2.2.2.1 Express 21
2.2.2.2 Express-Session 21
2.2.2.3 CORS 21
2.2.2.4 Body-Parser 22
2.2.2.5 Dotenv 22
2.2.2.6 Express-validator 22
2.2.2.7 Mongoose 22
2.2.2.8 Multer 22
2.2.2.9 AWS-SDK 22
2.2.2.10 Multer-S3 22
2.2.2.11 Bcrypt 23
2.2.2.12 Sharp 23
2.2.2.13 ImagesToPDF 23
2.2.3 Banco de Dados MongoDB 23
2.2.4 *AWS - Amazon Web Service* 25
2.3 *Front-end* 25
2.3.1 HTML 26
2.3.2 CSS 26
2.3.3 JavaScript 26
2.3.4 TypeScript 26
2.3.5 Módulos e *Framework - Front-end* 26
2.3.5.1 Angular 26
2.3.5.2 HttpClient 27
2.3.5.3 *Toaster* 27
2.3.5.4 *Routes* 27
2.4 Ferramentas e *Sofwares* Utilizados 27

2.4.1 VS Code . . . 27
2.4.2 GitHub . . . 27
2.4.3 Heroku . . . 27
2.4.4 Postman . . . 28
2.4.5 Robo3t . . . 28
2.4.6 MongoDB Atlas . . . 28
2.4.7 AWS - Amazon S3 . . . 28
2.5 Considerações finais . . . 28

3 PLATAFORMA DE COMPARTILHAMENTO DE MATERIAIS DE ESTUDO . . . 29
3.1 Funcionalidades . . . 29
3.2 Arquitetura . . . 30
3.2.1 Visão Visitante . . . 30
3.2.2 Visão Usuário . . . 31
3.2.3 Visão Administrador . . . 32
3.3 Modelo Banco de Dados . . . 32
3.4 Considerações finais . . . 33

4 DESENVOLVIMENTO . . . 35
4.1 Etapas de aprendizado . . . 35
4.1.1 *Node.js* . . . 35
4.1.2 *Express* . . . 35
4.1.3 Entendendo o *Servidor* . . . 36
4.1.4 *Angular, React ou Vue* . . . 36
4.1.5 *Angular* . . . 36
4.1.6 *Routing* . . . 39
4.1.7 *HTTP Requests* . . . 40
4.1.8 Enviando dados do *back-end* para o *front-end* . . . 41
4.1.9 *File Upload* . . . 43
4.1.10 Salvando Arquivos Localmente . . . 44
4.1.11 *Heroku* . . . 44
4.1.12 *Git* . . . 45
4.1.13 Integração MongoDB-Servidor . . . 45
4.1.14 *Atlas - MongoDB* . . . 49
4.1.15 Escolha do armazenamento remoto das imagens . . . 49
4.1.16 *AWS - S3 ou EC2* . . . 50
4.1.17 Integração AWS-Servidor . . . 50
4.1.18 Nomeando Imagens . . . 52
4.1.19 Renderizando Informação do Banco de Dados . . . 53

4.1.20 Buscando Arquivos no S3-Bucket e baixando as imagens 56
4.1.21 Convertendo Imagens em PDF 58
4.1.22 Baixando o Arquivo PDF 59
4.1.23 *Hash* nas senhas 61
4.1.24 Utilizando *session* para permanecer logado 61
4.1.25 Aplicação 62
4.2 Resultados 62
4.3 Considerações Finais 63

5 CONCLUSÃO 65
5.1 Trabalhos futuros 65
5.2 Problemas encontrados 66

REFERÊNCIAS 69

1 INTRODUÇÃO

1.1 Motivação e Contextualização

O curso de graduação, por si só, demonstra várias dificuldades para o discente, sendo elas de cunho acadêmico e/ou pessoal considerando o extenso tempo necessário para a formação do aluno. Dentre todos os empecilhos, a falta de norte em buscar opções de bons materiais de estudo para cada uma da matérias se mostra um dos maiores.

Quando o discente encontra um bom material para estudos, que encaixa com o seu tipo de comunicação, seja por livros ou anotações de um professor específico, colegas ou veteranos, aquele já se sente mais motivado e seguro no estudo da disciplina, consequentemente, criando um interesse maior por esta e estudando com afinco, muitas vezes, até se aprofundando mais na disciplina do que o ensinado em sala de aula ou cobrado nas provas e trabalhos. Em outros casos, se sente mais a vontade para estudar matérias extra-curriculares por lhe sobrar tempo, tempo este que seria gasto exaustivamente em pesquisa por bons materiais.

Assim, uma estratégia interessante é a utilização de uma plataforma onde os alunos e professores que já passaram pelo processo de pesquisa, busca e desenvolvimento por materiais possam compartilhá-los com outros alunos, para que a curva de aprendizado dos alunos da universidade, como um todo, possa ser acelerada. Assim, os alunos poderão passar por um período de graduação melhor aproveitado, seja estudando mais a fundo as disciplinas, seja participando de grupos de extensão, tendo mais tempo para cuidar de assuntos pessoais, lazer e saúde mental ou mesmo aproveitando o tempo extra para estudo de conteúdo extra-curricular, como uma língua estrangeira.

1.2 Objetivo

Este trabalho tem como objetivo o desenvolvimento de uma plataforma web, voltada para estudantes e docentes da USP, com todas as funcionalidades que possibilitem o compartilhamento de materiais de estudo com toda a comunidade da universidade, bem como os meios de busca pelos materiais na plataforma. Nela, discentes e docentes podem se cadastrar e postar as próprias anotações, enquanto os estudantes da graduação tem um local único com grande variedade de materiais de estudo destinados à graduação.

1.3 Organização

Esta monografia está dividida em 5 capítulos. No Capítulo 2 são apresentadas as tecnologias e ferramentas utilizadas para o desenvolvimento do trabalho. Em seguida, no Capítulo 3, são apresentados aspectos relacionados ao projeto que foram definidos antes

do início do desenvolvimento da plataforma e se fazem necessários para o entendimento da mesma. No Capítulo 4 são discorridas as etapas de aprendizado que foram essenciais para implementar as funcionalidades da plataforma e os resultados obtidos. Por fim, no Capítulo 5, são apresentadas as conclusões e recomendações para trabalhos futuros.

2 REVISÃO DA LITERATURA

Neste capítulo são apresentados conceitos sobre a plataforma a ser desenvolvida no projeto e as tecnologias e ferramentas utilizadas, juntos a uma breve explicação do motivo pelas quais foram escolhidas.

2.1 *Responsive web design*

O *Responsive web design* (RWD) é uma abordagem ao *design* da Web que faz as páginas da Web renderizarem bem, automaticamente, reorganizando-se e redimensionado-se de acordo com o tamanho da tela do dispositivo que o acessa. Conteúdo, *design* e desempenho são necessários em todos os dispositivos para garantir usabilidade e satisfação. Essa adaptabilidade se mostra de grande importância considerando o enorme número de usuários de *smartphones* e *tablets* atualmente.

Outras opções para desenvolver esse projeto seriam aplicativos nativos ou *web apps*.

Aplicativos nativos são aplicativos que podem trabalhar *off-line* no *smartphone* e são desenvolvidos especificamente para uma plataforma. Podem se apropriar de uma integração maior com os recursos do dispositivo como câmera, GPS (*Global Position System*), lista de contatos, etc.

Web apps podem ser considerados sites otimizados especificamente para dispositivos móveis com o intuito de trazer ao usuário a experiência de um aplicativo nativo. Embora aplicativos nativos e *wep apps* apresentem aspectos mais modernos em relação aos sites, cada um possui suas vantagens e momentos de utilização (NOGUEIRA et al., 2015).

Aplicativos nativos apresentam aspectos negativos em relação ao seu tempo de desenvolvimento, considerando a necessidade de serem projetados para cada sistema operacional a fim de atender a todos os usuários. Como vantagens apresentam o desempenho e o uso de recursos de hardware do dispositivo, porém estas vantagens não se destacam para este projeto.

Web apps possuem um tempo menor de desenvolvimento se comparados aos aplicativos nativos, mas são projetados tendo em mente especificamente dispositivos móveis. Como as funções principais desse projeto são o compartilhamento de imagens e o *download* de materiais em PDF, acredita-se que a plataforma será tão acessada a partir de ambiente *desktop* quanto de *smartphone* ou *tablet*.

Assim, foi o desenvolvimento deste projeto foi um site responsivo (ZEMEL, 2015) visando a flexibilização do uso para estudantes e professores, bem como o atendimento de todas as necessidades para a implementação de suas funcionalidades.

2.2 *Back-end*

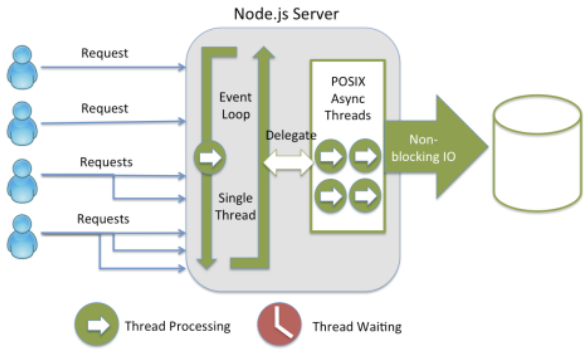
Back-end é a parte da aplicação que roda no servidor, responsável pelas funcionalidades, comunicação com o banco de dados, módulos de segurança e integração com o sistema de nuvem. Para isso foram utilizados o ambiente de execução Node.js, o banco de dados MongoDB e o sistema de nuvem AWS.

2.2.1 Node Js

Para o desenvolvimento das funcionalidades da plataforma, foi escolhido o Node.js[1]. O Node.js é um ambiente de execução para JavaScript, de comportamento assíncrono e orientado a eventos, além de ser de código aberto, que permite aos desenvolvedores utilizar JavaScript para programar o *server*. O Node.js usa um modelo de I/O (*Input/Output*) orientado a eventos, com funcionamento assíncrono não bloqueante, o que o torna uma ótima opção para aplicações em tempo real com troca intensa de dados por meio de dispositivos distribuídos.

Na Figura 1 é ilustrado o funcionamento das requisições com o Node.js.

Figura 1 – Funcionamento do Node.js



Fonte: <https://strongloop.com/strongblog/node-js-is-faster-than-java/>

Com base na Figura 1 entende-se então que o processo ocorre da seguinte forma:

1. Cliente faz a requisição;
2. Servidor passa a requisição para processamento e se disponibiliza para atender a próxima requisição;
3. Servidor continua atendendo requisições a medida que elas vão acontecendo;
4. Servidor responde as solicitações a medida que seus processamentos vão sendo concluídos.

[1] Node.js <https://nodejs.org/en/about/>

A escolha do Node.js para este projeto foi baseada no desempenho oferecido pelo ambiente, aliado ao comportamento assíncrono não bloqueante, também pensou-se na linguagem de programação utilizada por ele, o Javascript, a qual é a linguagem mais próxima do Typescript, sendo esta a linguagem utilizada para o desenvolvimento do *front-end* da aplicação. Isso permite a diminuição da complexidade do projeto como um todo e otimiza o tempo de aprendizado das linguagens.

2.2.2 Módulos do Node.Js

Durante o desenvolvimento do projeto foram utilizados os módulos e *frameworks* apresentados nas subseções que seguem, e que auxiliaram na criação da aplicação.

2.2.2.1 Express

Express[2] é um *framework* do Node.js que auxilia na construção das aplicações web, fornecendo mecanismos para:

- Gerenciar requisições HTTP (*Hypertext Transfer Protocol*) de diferentes tipos e rotas e URLs *(Uniform Resource Locator*);
- Combinar com mecanismos de renderização de *views* para gerar respostas inserindo dados em modelos;
- Definir as configurações comuns da aplicação web, como a porta a ser usada para conexão e a localização dos modelos que são usados para renderizar a resposta;
- Adicionar em qualquer ponto da requisição um *middleware* para interceptar, processar ou pré-processar e tratar a mesma.

2.2.2.2 Express-Session

As requisições HTTP são tratadas como transações independentes de qualquer requisição anterior, fazendo com que informações buscadas em uma requisição possam não estar disponíveis em outra requisição. O *Express-Session*[3] ajuda a sempre manter as informações do cliente em diferentes páginas ou requisições, auxiliando a manter o usuário logado, por exemplo.

2.2.2.3 CORS

O CORS (*Cross-origin resource sharing*)[4] é o módulo que lida com o compartilhamento de recursos de origem cruzada, sendoum mecanismo que permite que recursos

[2] Express <https://expressjs.com/>
[3] Express-Session <https://www.npmjs.com/package/express-session>
[4] CORS <https://www.npmjs.com/package/cors>

restritos em uma página da web sejam solicitados de outro domínio fora do domínio do qual o primeiro recurso foi servido. Uma página da web pode incorporar livremente imagens de origem cruzada, folhas de estilo, *scripts*, *iframes* e vídeos.

2.2.2.4 Body-Parser

O *Body-Parser*[5] é responsável por analisar os corpos da solicitação de entrada em um *middleware* antes de manipulá-lo.

2.2.2.5 Dotenv

O módulo Dotenv[6] é um módulo de dependência zero que carrega variáveis de ambiente de um arquivo *.env* em *process.env*.

2.2.2.6 Express-validator

O módulo *Express-validator*[7] é uma biblioteca de *middleware* para o *Express* que você pode incorporar em seus aplicativos para validação de dados do lado do servidor.

2.2.2.7 Mongoose

O módulo *Mongoose*[8] gerencia relacionamentos entre dados, fornece validação de esquema e é usado para traduzir entre objetos no código e a representação desses objetos no MongoDB.

2.2.2.8 Multer

O módulo *Multer*[9] é um *middleware* node.js para lidar com *multipart/form-data*, que é usado, principalmente, para fazer *upload* de arquivos. Ele é escrito baseado no *busboy* para máxima eficiência.

2.2.2.9 AWS-SDK

O módulo *AWS-SDK*[10] é uma coleção de ferramentas de software para a criação de aplicativos e bibliotecas que usam recursos do *Amazon Web Services* (AWS).

2.2.2.10 Multer-S3

O módulo Multer-S3[11] é um *middleware* que possui a mesma função do módulo *Multer*, porém, com o intuito de fazer a comunicação com o *bucket* S3 da AWS.

[5] Body-Parser <https://www.npmjs.com/package/body-parser>
[6] Dotenv <https://www.npmjs.com/package/dotenv>
[7] Express-validator <https://www.npmjs.com/package/express-validator>
[8] mongoose <https://mongoosejs.com/>
[9] Multer <https://www.npmjs.com/package/multer>
[10] AWS-SDK <https://www.npmjs.com/package/aws-sdk>
[11] Multer-S3 <https://www.npmjs.com/package/multer-s3>

2.2.2.11 Bcrypt

Bcrypt[12] é uma biblioteca que permite fazer um *hash* em *strings*. Uma função *hash* é um processo de conversão que leva um bloco arbitrário de dados e retorna uma cadeia de *bits* de tamanho fixo. Diferente da encriptação, o *hash* é um processo irreversível. Com isso pode-se salvar as senhas dos usuários no banco de dados em uma forma ilegível, deixando as informações mais seguras.

2.2.2.12 Sharp

Sharp[13] é um módulo de alto desempenho do Node.js utilizado no processamento e redimensionamento de imagens JPEG, PNG, WebP and TIFF.

2.2.2.13 ImagesToPDF

ImagesToPDF[14] é um módulo de alto desempenho do Node.js utilizado para converter imagens, de extensões JPEG e PNG, em arquivos PDF. Ela foi projetada para ser simples, tornando um processo complexo, de gerar documentos em algo tão simples quanto fazer algumas chamadas de função.

2.2.3 Banco de Dados MongoDB

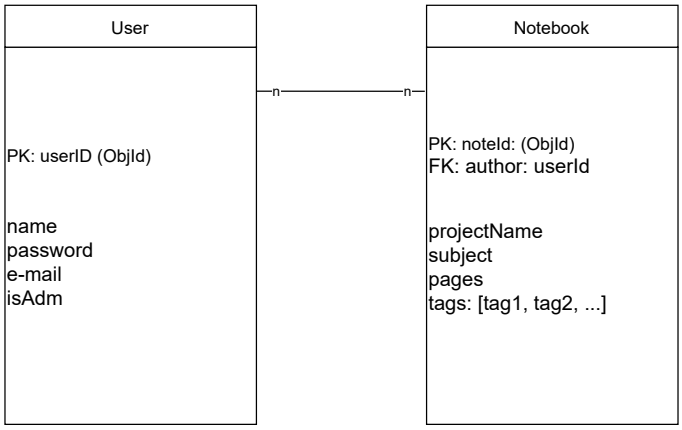
MongoDB[15] é um sistema de gerenciamento de banco de dados classificado como *NoSQL* (*Not Only Structured Query Language*), sendo *source-available cross-platform* orientado a documento que utiliza a linguagem *Javascript* por meio de documentos parecidos com JSON, os BSON, com esquemas opcionais, para a manipulação dos dados. Sua estrutura é baseada em coleções e objetos dentro destas. A estrutura do banco de dados se torna extremamente livre e flexível devido ao tipo de organização, não necessitando ter relações diretas entre todos os itens armazenados, porém, também permitindo se modelar o banco de dados tal qual um banco relacional. (HOWS, 2015)

Na Figura 2 é ilustrado um exemplo do modelo UML (*Unified Modeling Language*) para o banco de dados. Nas Figuras 3 e 4 são ilustradas as coleções do banco de dados em forma de tabela. Nas Figura 5 e Figura 6 as coleções estão representadas em formato BSON, o formato nativo do MongoDB.

Neste exemplo há duas coleções, *Notebooks* e *Users*, de dados armazenados no MongoDB, uma figura em formato de Tabela e outra no formato padrão de armazenamento do Mongo, BSON. A coleção Notebooks possui elementos com os dados: *noteId*, *author*, *projectName*, *subject* e um *array* de *tags*. Já em *Users*, temos os dados *userID*, *name*,

[12] Bcrypt <https://www.npmjs.com/package/bcrypt>
[13] Sharp <https://sharp.dimens.io/en/stable/>
[14] ImagesToPDF <https://www.npmjs.com/package/images-to-pdf>
[15] MongoDB <https://www.mongodb.com/>

Figura 2 – Exemplo de modelo UML de banco de dados MongoDB



Fonte: Elaborado pelo autor.

Figura 3 – MongoDB em formato de tabela - Coleção Users

	_id	name	password	email	isAdmin
1	ObjectId("60886644330b137eb3c72d66")	Andre	1234	andre.cavalcante@usp.br	true

Fonte: Elaborado pelo autor no software Robo3t.

Figura 4 – MongoDB em formato de tabela - Coleção Notebooks

	_id	tags	projectName	subject	pages	author	__v
1	ObjectId("608886fa11edce0937de6b39")	[1 element]	A	tcc1	11	6088664...	0
2	ObjectId("6088872d11edce0937de6b3a")	[1 element]	B	tc2	4	6088664...	0

Fonte: Elaborado pelo autor no software Robo3t.

Figura 5 – MongoDB em formato BSON - Coleção Users

```
/* 1 */
{
    "_id" : ObjectId("60886644330b137eb3c72d66"),
    "name" : "Andre",
    "password" : "1234",
    "email" : "andre.cavalcante@usp.br",
    "isAdmin" : "true"
}
```

Fonte: Elaborado pelo autor no software Robo3t.

password, *isAdm*. No UML, a *Primary Key* (PK) ou *chave primária* indica que este valor é único na coleção e que pode e deve ser utilizado como referência para elementos de outras coleções. A *Foreign Key* (FK) ou *chave estrangeira* indica que o parâmetro é uma PK de outra coleção, ou seja, uma referência para elementos de outra coleção. Nas Figura 5 e Figura 6, os valores de cada item dos objetos representam os dados. O UML indica que as coleções têm uma relação de multiplicidade $n - n$, o que significa que vários usuários podem estar associados a vários cadernos e que vários cadernos podem estar associados a vários usuários.

Figura 6 – MongoDB em formato BSON - Coleção Notebooks

```
/* 1 */
{
    "_id" : ObjectId("608886fa11edce0937de6b39"),
    "tags" : [
        "tag1"
    ],
    "projectName" : "A",
    "subject" : "tcc1",
    "pages" : 11,
    "author" : "60886644330b137eb3c72d66",
    "__v" : 0
}

/* 2 */
{
    "_id" : ObjectId("6088872d11edce0937de6b3a"),
    "tags" : [
        "tag2"
    ],
    "projectName" : "B",
    "subject" : "tc2",
    "pages" : 4,
    "author" : "60886644330b137eb3c72d66",
    "__v" : 0
}
```

Fonte: Elaborado pelo autor no software Robo3t.

Dado que esse projeto tem como público-alvo o corpo docente e discente da USP, não se espera um alto volume de dados e tráfego tal qual há em redes sociais. A escolha do banco de dados, então, não possui grandes efeitos no desempenho dessa aplicação, sendo realizada de acordo com a preferência do desenvolvedor. Porém, devido às condições de contorno da modelagem feita, percebeu-se que um modelo *NoSQL* apresentou ganhos no processo de escrita e diminuição na quantidade de coleções necessárias, optando-se assim por um modelo "quasi-relacional". Entre os motivos da escolha do *MongoDB* estão a sua baixa curva de aprendizagem e a sua popularidade, dentre as opções *NoSQL*, que faz com que a maioria dos provedores de hospedagem já tenha suporte para o MongoDB.

2.2.4 *AWS - Amazon Web Service*

AWS (*Amazon Web Services*) é uma plataforma de computação em nuvem abrangente e em evolução fornecida pela Amazon que inclui uma mistura de ofertas de infraestrutura como serviço (IaaS), plataforma como serviço (PaaS) e pacote de software como serviço (SaaS).

2.3 *Front-end*

O *front-end* está relacionado à parte visual da aplicação, o que será visto pelos usuários. Inclui a criação do *layout* das páginas e o processo de torná-las responsivas.

2.3.1 HTML

O HTML (*HyperText Markup Language*) ou Linguagem de Marcação de Hipertexto é uma linguagem de marcação utilizada na construção de páginas na Web. Documentos HTML podem ser interpretados por navegadores.

2.3.2 CSS

O CSS (*Cascading Style Sheets*) é utilizado para definir o estilo dos elementos de uma página HTML, ou seja, serve para personalizar a aparência visual da aplicação.

2.3.3 JavaScript

JavaScript é uma linguagem de programação que permite implementar funcionalidades mais complexas em páginas web, tornando-as mais dinâmicas e interativas.

2.3.4 TypeScript

TypeScript é a linguagem utilizada pelo *Framework* Angular e é uma linguagem de programação desenvolvida e mantida pela *Microsoft*. É um superconjunto sintático estrito em *JavaScript* e adiciona tipagem estática opcional à linguagem. O *TypeScript* foi projetado para o desenvolvimento de grandes aplicativos e transcompilações para *JavaScript*.

2.3.5 Módulos e *Framework* - *Front-end*

Durante o desenvolvimento do *front-end* foram utilizados os módulos e *frameworks* que auxiliam na criação da aplicação nas seção que seguem.

2.3.5.1 Angular

O Angular[16] é um *framework open-source* baseado em *TypeScript* desenvolvido e contribuído pelo *Angular Team* na Google e pela comunidade. Angular é um relançamento do AngularJS, que era todo construído em *Javascript*, *open source*, e é utilizado para o desenvolvimento de interface de usuário ou para componentes utilizados por esta. Angular pode ser utilizado como base em um projeto *single page* ou aplicação *mobile*, bem como em sites responsívos, já possuindo poder para gerenciamento de estado e renderização para o *DOM (Document Object Model)* dos elementos da página, bem como capacidade de *routing*.

[16] Angular <https://angular.io//>

2.3.5.2 HttpClient

O HttpClient[17] é um módulo Angular que permite que o aplicativo se comunique com serviços de *back-end* pelo protocolo HTTP. O módulo permite realizar todas as solicitações HTTP, incluindo GET, POST, PUT, PATCH e DELETE.

2.3.5.3 *Toaster*

O *Toaster*[18] é um módulo que facilita o envio de notificações à tela do usuário.

2.3.5.4 *Routes*

O *Routes*[19] é um módulo que facilita o roteamento de páginas no Angular para que ele possa fazer a renderização na troca de página sem necessidade de atualização da mesma.

2.4 Ferramentas e *Sofwares* Utilizados

Para o desenvolvimento do projeto foram utilizados os softwares apresentados nas próximas subseções.

2.4.1 VS Code

VS Code (ou *Visual Studio Code*)[20] é um editor de texto e código-fonte multiplataforma.

2.4.2 GitHub

GitHub[21] é uma plataforma de hospedagem de código-fonte com controle de versão. Permite a divulgação do código do projeto e o trabalho em conjunto. O código fonte desse projeto encontra-se no repositório do GitHub: <https://github.com/tcc-usp/tcc>.

2.4.3 Heroku

Heroku[22] é uma solução de Plataforma como Serviço que permite ao desenvolvedor criar e subir rapidamente suas aplicações para a nuvem. A aplicação desenvolvida nesse projeto encontra-se hospedada no Heroku: <https://tcc-usp.herokuapp.com/>.

[17] HttpClient <https://angular.io/api/common/http/HttpClient>
[18] Toaster <https://www.npmjs.com/package/ngx-toastr>
[19] Routes <https://angular.io/api/router/Routes>
[20] VS Code <https://code.visualstudio.com/>
[21] GitHub <https://github.com/>
[22] Heroku <https://www.heroku.com/>

2.4.4 Postman

Postman[23] é uma aplicação que permite enviar e receber dados via requisições HTTP. Com ele é possível automatizar testes para sua aplicação, evitando a necessidade de fazer testes manualmente.

2.4.5 Robo3t

Robo 3T (anteriormente chamado de Robomongo)[24] é uma interface gráfica de usuário (GUI), para desktop, popular às implantações de hospedagem do MongoDB que permite que você interaja com seus dados por meio de indicadores visuais em vez de uma interface baseada em texto.

2.4.6 MongoDB Atlas

MongoDB Atlas[25] é um banco de dados totalmente gerenciado em nuvem desenvolvido pelas mesmas pessoas que criam o MongoDB. O Atlas lida com toda a complexidade de implantação, gerenciamento e recuperação de suas implantações.

2.4.7 AWS - Amazon S3

AWS (*Amazon Web Services*)[26] é uma plataforma de computação em nuvem abrangente e em evolução fornecida pela Amazon que inclui uma mistura de ofertas de infraestrutura como serviço (IaaS), plataforma como serviço (PaaS) e pacote de software como serviço (SaaS). Amazon S3 ou *Amazon Simple Storage Service* é um serviço oferecido pela *Amazon Web Services* que fornece armazenamento de objetos por meio de uma interface de serviço da web. O Amazon S3 usa a mesma infraestrutura de armazenamento escalonável que a Amazon.com usa para executar sua rede global de comércio eletrônico.

2.5 Considerações finais

Neste capítulo foram abordados conceitos sobre as tecnologias e ferramentas utilizadas no desenvolvimento deste trabalho. No próximo capítulo é abordado o projeto da plataforma, esse se faz necessário antes do desenvolvimento para que se tenha uma melhor compreensão e visão do produto gerado.

[23] Postman <https://www.getpostman.com/>
[24] Robo 3T <https://robomongo.org/>
[25] MongoDB Atlas <https://www.mongodb.com/cloud/atlas>
[26] AWS <https://aws.amazon.com/>

3 PLATAFORMA DE COMPARTILHAMENTO DE MATERIAIS DE ESTUDO

Neste projeto foi desenvolvido o protótipo de uma plataforma web responsiva que permite que estudantes e professores da USP possam armazenar e compartilhar materiais de estudo com toda a comunidade da universidade. Estes materiais contribuirão para o aprendizado e melhor aproveitamento do discente durante o período de graduação, se tornando uma das principais fontes de pesquisa que propicia o processo de aprendizagem. Funciona como um grande armazenador e gerenciador de dados em nuvem.

Neste capítulo serão detalhados aspectos referentes ao projeto, definidos antes da criação da aplicação, a fim de que se tenha uma visão do produto que será criado. Inclui, então, a definição das funcionalidades da aplicação, o fluxo do site e o modelo do banco de dados.

3.1 Funcionalidades

Antes do início do desenvolvimento foram definidas as funcionalidades da aplicação. Foi feita inicialmente uma lista com as ações que os usuários poderiam realizar, sendo eles professores ou alunos. Como é usual no desenvolvimento de software, durante o decorrer desse projeto foram identificadas necessidades de modificação, implementação de novas funcionalidades ou retirada de algumas funcionalidades devido à relação complexidade/tempo.

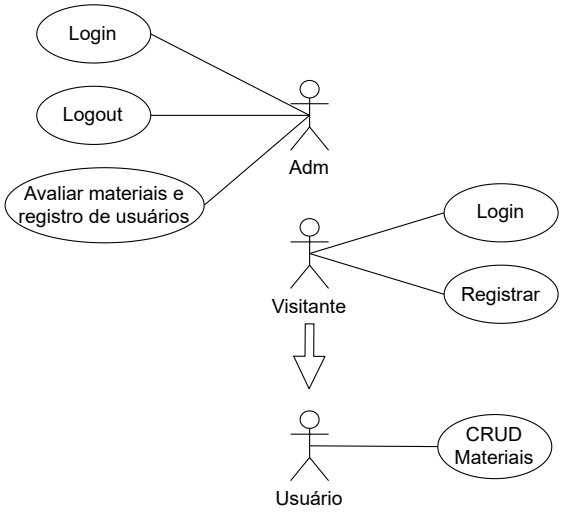
O diagrama de casos de uso ilustrado na Figura 7 apresenta as funcionalidades do sistema final, após todas as incrementações e modificações terem sido realizadas. Na imagem, CRUD (*Create, Read, Update, Delete*), indicando que é possível criar uma conta, postar, ler, atualizar e deletar um registro, por registro lê-se material.

Nota-se pelo diagrama que, assim como em outros sites nos quais é possível ter uma conta para o usuário, esta plataforma dispõe para todos a opção de fazer um registro e *login*, assim como ações relacionadas ao registro, por exemplo, recuperação de senha. Para cada tipo de usuário são solicitados alguns dados específicos no registro, como o *e-mail* com o domínio *USP*.

Os usuários podem visualizar, na tela inicial, os materiais que postaram, bem como acessar a tela de busca e *download* de materiais.

Há também o usuário do tipo *administrador* que fica responsável por analisar as solicitações de registro no site e aceitá-las ou não, dependendo da validade das suas informações, bem como servir de moderador de conteúdos.

Figura 7 – Casos de uso



Fonte: Elaborado pelo autor

3.2 Arquitetura

Com as funcionalidades listadas é preciso definir o fluxo das visões, ou seja, os procedimentos a serem realizados pelo usuário para permitir ações como registro, postagem de novos materiais, entre outros. Nas Figuras 8 a 10 são ilustrados os diagramas do mapa do site resultante no final do projeto, divididos em visões diferenciadas para visitantes sem uma conta, usuários e administrador, respectivamente. Cada bloco dentro da visão representa uma seção do site e, após o *login*, podem ser acessados a partir de qualquer página, já que no site há um cabeçalho com os *links* para o direcionamento.

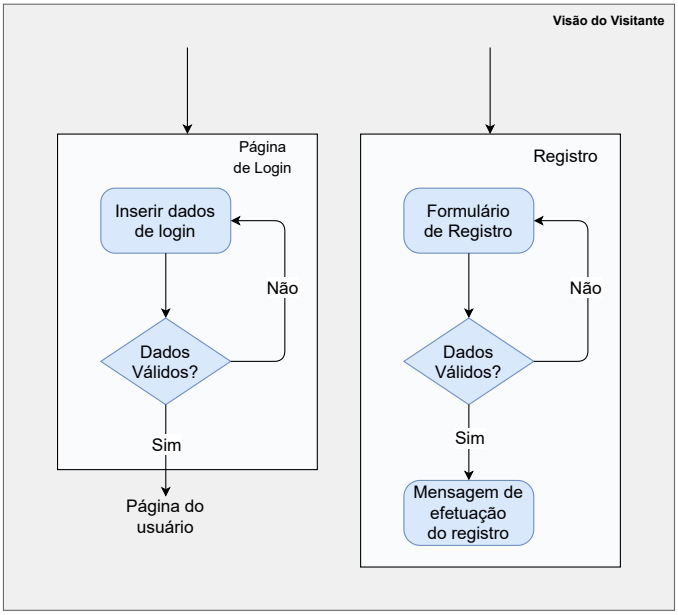
3.2.1 Visão Visitante

Na Figura 8 é ilustrado o percorrer no site como visitante permite ao usuário realizar os seguintes procedimentos: *Login* e Registro.

No processo de ***Login*** o usuário começa em uma página para inserir os dados e, se estes forem válidos, será redirecionado à sua página inicial. Caso contrário é exibida a seguinte mensagem de erro para o usuário: *Dados Incorretos*.

Para o **Registro** o usuário deve primeiro informar seus dados sendo estes *Nome Completo*, *E-mail*, *Senha*, e *Repetir a Senha*. Se a senha e a confirmação dela forem iguais e o *e-mail* com domínio *USP* for válido, o usuário recebe uma mensagem de confirmação de cadastramento e retorna para a página de *Login*. Caso contrário, o usuário recebe uma mensagem de erro.

Figura 8 – Visão Visitante

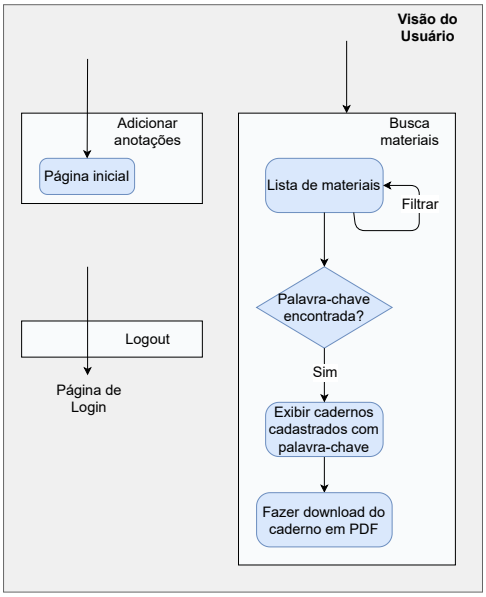


Fonte: Elaborado pelo autor

3.2.2 Visão Usuário

A **Página Inicial** do usuário contém um formulário para envio de novos materiais, sendo eles novos cadernos ou atualização de cadernos posteriormente criados e a lista dos cadernos já criados pelo usuário.

Figura 9 – Visão Usuário



Fonte: Elaborado pelo autor

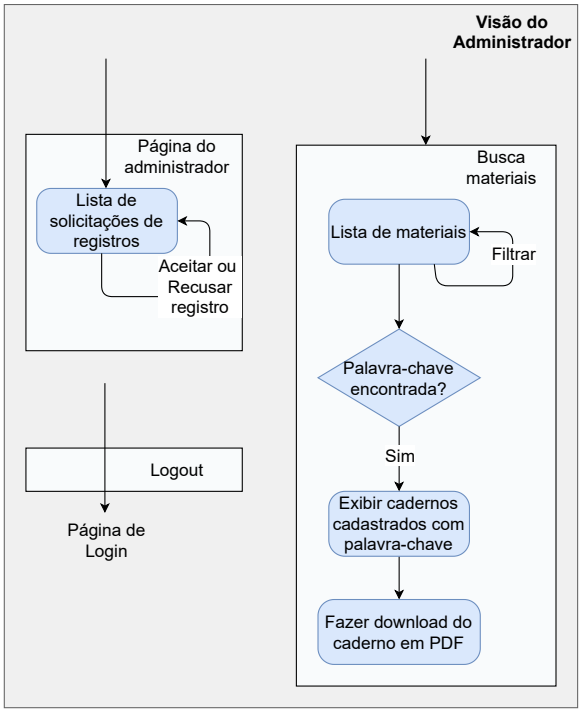
As seções **Página Inicial**, **Buscar**, **Perfil**, **Trocar Senha** e **Logout** estão pre-

sentes para todos os usuários na *Top Bar*

3.2.3 Visão Administrador

Conforme ilustrado na Figura 10, o administrador do site pode realizar **Login**, **Logout** e ver sua **Página do Administrador**.

Figura 10 – Visão Administrador



Fonte: Elaborado pelo autor

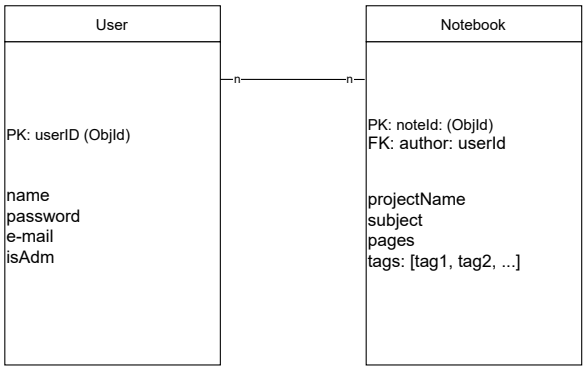
Login e *Logout* funcionam da mesma forma que nos outros modelos. Na **Página Inicial** do **Administrador** há uma lista de solicitações de registros com as informações dos usuários. Com isso o administrador pode avaliar o cadastro e recusar ou aceitar a solicitação de cadastro. Após a ação ele continua na sua página de administrador.

3.3 Modelo Banco de Dados

Assim como no projeto das funcionalidades para o banco de dados também foi feito um modelo inicial que, com o decorrer do desenvolvimento, foi sofrendo mudanças devido às novas funcionalidades adicionadas ou para facilitar a implementação e busca ou inserção de dados. O modelo final do banco de dados é ilustrado na Figura 11.

Como já apresentado anteriormente, o modelo ilustra que o banco de dados é composto por duas coleções: *Users*, *Notebooks*

Figura 11 – Modelo UML de classes do banco de dados



Fonte: Elaborado pelo autor

A Coleção **Users** contém parâmetros comuns a qualquer tipo de usuário(*id*, *name*, *password*, *e-mail* e tipo de conta, definindo se é Administrador, pelo valor *isAdmin*). O parâmetro id é sua *Primary Key*, ou seja, na coluna *id* não há valores repetidos e seu valor é utilizado como referência para se relacionar com outras coleções.

A tabela **Notebooks** contém os parâmetros (id, *projectName*, *author*, *subject*, *pages*, e *tags*. O parâmetro id é sua *Primary Key* e *author* utiliza a *Primary Key* do usuário que criou o caderno.

3.4 Considerações finais

Neste capítulo foi apresentado o projeto para o andamento deste trabalho, necessário para dar início ao desenvolvimento. Foram abordadas as funcionalidades, mapas e banco de dados da aplicação.

No próximo capítulo são apresentados os conceitos relacionados ao desenvolvimento que permitiram a criação da plataforma.

4 DESENVOLVIMENTO

Neste capítulo são mencionadas as etapas de aprendizado para o desenvolvimento das funcionalidades da aplicação e tomada de decisão de fluxos de processos, envolvendo aspectos criação de servidor, de roteamento, conexão e manipulação do banco de dados, comunicação com o sistema de nuvem, envio de dados do *back-end* para o *front-end* e vice-versa e persistência de sessão. São mencionados também os resultados obtidos no trabalho.

4.1 Etapas de aprendizado

4.1.1 *Node.js*

Primeiramente, foi necessário estudar o funcionamento do Node, seu comportamento não bloqueante e sua característica assíncrona. Assim, foram desenvolvidas diversas pequenas funções, como teste de aprendizado, utilizando o conceito de *callbacks*, *promises* e, principalmente, *promises* com o *syntactic sugar "async/await"*. Esta última foi a opção preferida para situações em que o comportamento assíncrono era exigido e havia um fluxo estritamente bloqueante, para se evitar o *callback hell* e se acostumar com a sintaxe mais moderna fornecida pelo *Javascript*.

4.1.2 *Express*

Para criar uma aplicação no Node[1] e facilitar o recebimento de requisições HTTP e envio de respostas ao cliente, utilizou-se o Express.js. No Código 1 é apresentado um exemplo básico de sua aplicação.

Código 1 - Exemplo de requisição do Express.js.

```
const express = require('express');
const app = express();

app.get('/', functionGet);
app.post('/', functionPost);

app.listen(8080);
```

Fonte: O autor

[1] Curso Node <https://www.udemy.com/course/the-complete-nodejs-developer-course-2>

No exemplo, Código 1, ocorre primeiramente a importação do módulo *express*, seguida pela criação de uma instância da aplicação. Há então o estabelecimento das rotas para que o servidor saiba como lidar com cada requisição feita pelo cliente. Neste exemplo, ao receber uma requisição *GET* no endereço inicial do site, é realizada a função *functionGet*. No caso de uma requisição *POST* no mesmo endereço, será realizada a *functionPost*. Por fim a aplicação é vinculada à porta 8080.

Fazendo uso do *Express* é possível então criar uma aplicação, definir os endereços acessíveis pelo usuário e as respostas que serão enviadas em cada endereço.

4.1.3 Entendendo o *Servidor*

Foi necessário estudar o processo por trás de utilizar ou não um servidor *back-end* para a aplicação, seus prós e contras. Então, a principal necessidade encontrada para a utilização do servidor foi a possibilidade de criar uma camada de segurança extra à aplicação, considerando que o servidor faz as funções de maior complexidade, funcionando como o "cérebro"da aplicação, bem como servir de intermediário em qualquer tipo de comunicação entre o banco de dados, o *front-end* e o sistema de nuvem, o que se mostra de extrema importância, considerando que a aplicação possui dados sensíveis como nome, endereço de e-mail e senha do usuário. Além disso, oferece a possibilidade de separação entre o *back-end* e o *front-end* em infraestruturas diferentes, adicionando assim, uma camada extra de segurança aos dados sensíveis.

4.1.4 *Angular, React ou Vue*

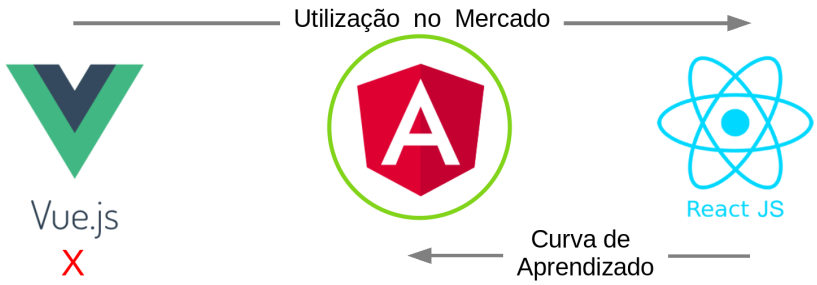
Para se escolher o *framework* ou biblioteca *front-end* a ser utilizada no projeto, como o aluno não possuía qualquer experiência prévia além de HTML, CSS e Javascript, foi feita uma pesagem entre curva de aprendizado e utilização da tecnologia no mercado. Como há uma diferença considerável de utilização entre *Vue* e as outras tecnologias, ele foi excluído. Restaram, assim, *Angular* e *React*, sendo esta a mais utilizada na indústria. Porém, *Angular* pareceu oferecer uma curva de aprendizado mais acelerada para noções básicas, o que seria suficiente para o desenvolvimento do projeto. Assim, pensando no prazo para aprendizado e desenvolvimento do projeto, *Angular* foi a escolha mais sensata como *framework front-end*, como é ilustrado na Figura 12.

4.1.5 *Angular*

O estudo e a utilização do Angular[2] se iniciou com o aprendizado dos comandos básicos, como criação de um projeto, inicialização de um servidor *front-end* local, criação de componentes, a compreensão da estrutura básica do *framework* que é, principalmente,

[2] Curso Angular <https://www.udemy.com/course/the-complete-guide-to-angular-2/>

Figura 12 – Escolha do ambiente Front-end



Fonte: Elaborado pelo autor

baseada em componentes e mudança de estado dos mesmos, com estes funcionando de forma modular além de a utilização do arquivo *app-module.ts*.

Cada componente funciona de forma independente e pode ser formado por outros componentes. Cada um deles possui um HTML, um CSS e um Typescript associados onde se podem fazer as alterações necessárias.

Código 2 - Exemplo de estrutura de arquivos do Angular

```
@Component({
  selector: 'app-login-screen',
  templateUrl: './login-screen.component.html',
  styleUrls: ['./login-screen.component.css'],
})
```

Fonte: O autor

Todos os arquivos Typescript dos componentes têm um trecho de código como o ilustrado no Código 2, onde se tem o nome do componente, e o direcionamento para os seus respectivos arquivos HTML e CSS.

Código 3 - HTML do componente Topnav

```
<div class="topnav" id="myTopnav">
  <a routerLink="/home" routerLinkActive="active">
  Pagina Inicial</a>
  <a routerLink="/search" routerLinkActive="active">Buscar</a>
  <a routerLink="/user" routerLinkActive="active">Perfil</a>
  <a class="icon" id="hamburger">
    <i class="fa fa-bars"></i>
  </a>
</div>
```

Fonte: O autor

No Código 3 tem-se o HTML referente à *Topbar* que possui o *Menu* oferecido ao usuário depois de logado.

Código 4 - Chamada do componente Topnav

```
<div class='topbarContainer'>
  <app-topbar></app-topbar>
</div>
<h2 class='header'>Envie imagens para seu caderno</h2>
<app-upload></app-upload>
```

Fonte: O autor

Já no Código 4 tem-se o HTML referente à Página Inicial, *Home*, a qual possui o componente *app-topbar* como um elemento.

4.1.6 *Routing*

Depois de entender a estrutura básica de funcionamento do *Angular*, foi necessário aprender a fazer o *Routing* dentro dele. Para isto, foi utilizado o módulo *Router* do próprio *framework*, o que facilitou muito a troca de páginas e retirou a necessidade de novo recarregamento a cada nova renderização. O roteamento acontece utilizando o arquivo *app-module.ts*

Código 5 - Declaração de rotas no Angular

```
const appRoutes: Routes = [
  { path: '', component: LoginScreenComponent },
  { path: 'home', component: HomeComponent },
  { path: 'user', component: UserComponent },
  { path: 'search', component: SearchComponent },
  { path: 'notifications', component: NotificationsComponent },
  { path: 'registration', component: RegisterScreenComponent },
];
```

Fonte: O autor

No Código 5 tem-se as rotas que foram criadas para a aplicação, no parâmetro *path* além de qual componente deve ser renderizado ao acessar essa rota no parâmetro *component*. Tudo isso é gerenciado pelo módulo *Router*.

Código 6 - Redirecionamento para página de registro

```
import { Component, OnInit } from '@angular/core';
import { Router } from '@angular/router';

import { environment } from 'src/environments/environment';

@Component({
  selector: 'app-login-screen',
  templateUrl: './login-screen.component.html',
  styleUrls: ['./login-screen.component.css'],
})
export class LoginScreenComponent implements OnInit {
  constructor(
    private router: Router
  ) {}

  register = () => {
    this.router.navigate(['/registration']);
  };
}
```

Fonte: O autor

No exemplo ilustrado no Código 6 tem-se uma parte do código da página de *login*, a função do botão para se redirecionar à página de registro, é possível observar como fazer a importação do módulo *Router* e como fazer a utilização dele. Primeiramente, faz-se necessária a importação do módulo, após isso, é feita a declaração do módulo no *constructor* e após isso a chamada da função *navigate* usando como argumento */registration* que é a rota referente à página de registro no *app.module.ts*.

4.1.7 *HTTP Requests*

Ao receber informações no *front-end*, foi necessário passá-las para o *back-end* e vice-versa, para que fossem devidamente processadas, porém, foi fundamental entender melhor quais as possibilidades de comunicação entre cliente e servidor. No caso, a mais utilizada é por comunicação HTTP[3]. Dentro dela, foi necessário compreender as formas de envio de informação e quais argumentos é possível passar dentro delas. No caso, os métodos mais utilizados no projeto foram *GET* e *POST*, onde a primeira se utiliza para

[3] Curso HTTP <https://www.youtube.com/watch?v=iYM2zFP3Zn0&ab_channel=TraversyMedia>

receber informações no *front-end* vindas do servidor e *POST* o contrário. Porém, algo importante a se ressaltar é a possibilidade de envio de parâmetros do *front* para o *back* a partir de requisições *GET* também, por utilização de *query*. Depois dessas considerações, optou-se por utilizar sempre *POST* para envio de informações do usuário para o *back-end* e *GET* apenas para o caminho contrário, com a intenção de criar uma maior padronização do código.

4.1.8 Enviando dados do *back-end* para o *front-end*

Com a conexão ao banco de dados estabelecida é possível pegar valores que estão armazenados em seu interior, sendo preciso depois disso passar estes dados para o *front-end* o qual foi construído com a utilização do *Angular* como *framework*.

No *Angular* a programação é feita em cima de componentes, os quais possuem um arquivo HTML, um CSS e um Typescript associados. Assim, cada página pode receber diversos componentes, que funcionam de forma modular, podendo formar diversas páginas diferentes com os mesmos componentes.

No Código 7 tem-se um exemplo básico da utilização do Typescript de um componente do *Angular*, o componente *login-screen* para estabelecer a comunicação cliente servidor.

Código 7 - Exemplo da utilização do Typescript na função *login*

```
import { Component, OnInit } from '@angular/core';
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { Router } from '@angular/router';
import { ToastrService } from 'ngx-toastr';
import { environment } from 'src/environments/environment';

@Component({
  selector: 'app-login-screen',
  templateUrl: './login-screen.component.html',
  styleUrls: ['./login-screen.component.css'],
})
export class LoginScreenComponent implements OnInit {
  constructor(
    private toastr: ToastrService,
    public http: HttpClient,
    private router: Router
  ) {}
```

```
  readonly apiUrl = environment.apiURL;
  user = {
    email: '',
    password: '',
  };
  authorizedID = 'null';

  login = () => {
    const userUrl = this.apiUrl + 'login';
    this.http.post(userUrl, this.user)
    .subscribe((responseData) => {});
    this.http.get<any>(userUrl).subscribe((data) => {
        this.authorizedID = data.msg;
        console.log('Session Hash ID', this.authorizedID);
        if (
          this.authorizedID.toString() === 'null' ||
          this.authorizedID.toString() === 'Dados Incorretos'
        ) {
          this.toastr.error(this.authorizedID);
          this.user.password = '';
        } else {
          this.user = {
            email: '',
            password: '',
          };
          this.router.navigate(['/home']);
        }
      });
    } else {
      this.toastr.warning('Preencha os Campos');
    }
  };
    ngOnInit(): void {}
}
```

Fonte: O autor

Neste Código 7 tem-se a função de *login* da aplicação, onde o botão de *login* serve como *trigger* para a execução da função, a qual envia uma requisição **HTTP** do tipo

POST com o endereço final *userUrl* e o objeto *user* que possui as informações do *login*: *email* e *password*. Para isso, foi utilizado o módulo *HttpClient* no *Angular*. Posteriormente, é feita a requisição **HTTP** do tipo **GET**, a qual recebe a mensagem vinda da comunicação entre o banco de dados e o *back-end*, autorizando ou não o *login* e a *hash* do ID do usuário e, posteriormente, redireciona à página inicial do usuário. Além disso, foram utilizados os módulos *ToasterService* e *Router*, os quais são, respectivamente, utilizados para exibir mensagens de confirmação ou erro ao usuário e para fazer a troca de *URLs* no *Angular*, para que não haja necessidade de recarregamentos de páginas.

4.1.9 *File Upload*

Para executar o *upload* dos arquivos de imagem do *front-end* para o *back-end*, foi utilizado o módulo *FileUploader* do *Angular*. Este trabalha criando um *stream* de dados com a requisição HTTP do tipo *POST*, recebendo como argumentos a URL e o arquivo.

Código 8 - Typescript do componente upload no Angular

```
import { Component, OnInit } from '@angular/core';
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { FileUploader } from 'ng2-file-upload';
import { ToastrService } from 'ngx-toastr';
import { environment } from 'src/environments/environment';
@Component({
  selector: 'app-upload',
  templateUrl: './upload.component.html',
  styleUrls: ['./upload.component.css'],
})
export class UploadComponent implements OnInit {
  constructor(private toastr: ToastrService,
              public http: HttpClient) {}
  readonly apiUrl = environment.apiURL;
  public uploader: FileUploader = new FileUploader({
    url: this.apiUrl + 'imgUpload',
    itemAlias: 'image',
  });
  uploadFiles = () => {
    this.uploader.uploadAll();
  }
}
```

Fonte: O autor

No Código 8 tem-se a importação dos módulos necessários, no *Angular*, e a declaração de *uploader* como um *FileUploader*, que recebe a URL para comunicação com o *back-end* e a imagem.

4.1.10 Salvando Arquivos Localmente

Em um primeiro momento, os arquivos de imagem foram salvos localmente.

Código 9 - Função de upload local de imagens no back-end

```
const multer = require("multer"),
  fs = require("fs")

let uploadLocal = multer({
  storage: multer.diskStorage({
    destination: (req, file, cb) => {
      cb(null, path);
    },
    filename: (req, file, cb) => {
      setTimeout(() => {
        cb(null, fileNameFunc(file));
      }, 500);
    },
  }),
});
```

Fonte: O autor

No Código 9 tem-se a importação do módulo *multer* e do módulo *fs* nativo do *Node* onde, respectivamente, permitem a manipulação de arquivos e a alteração de arquivos no sistema local. Após isso, utiliza-se o *multer* dando-lhe como argumentos o arquivo e o *path* com o qual deve ser salvo.

4.1.11 *Heroku*

A ideia do projeto é uma proposta de prova de conceito, sendo de grande interesse fazer a implementação mais próxima da real. Desse modo, com uma aplicação sendo executada independentemente do ambiente de desenvolvimento, com um servidor próprio, funcionando sem interrupções. Assim, iniciou-se o estudo de execução de *deploy* e seleção de tecnologias para este fim. Com isso em mente, a plataforma encontrada que oferece uma curva de aprendizado mais baixa para provas de conceito é o *Heroku*, já que ele exonera o desenvolvedor da criação e manutenção de uma infraestrutura própria, como

seria o necessário para o caso de utilização da AWS com máquina virtual, aliado a Docker, por exemplo, e ainda oferece o bônus de efetuar *deploy* automático ao se atualizar um repositório associado no *GitHub*.

4.1.12 *Git*

Ao se criar maior robustez e complexidade no projeto, foi necessário utilizar uma plataforma de gerenciamento de versão. Para isso, o estudo do funcionamento básico de comandos *Git* se mostrou de extrema necessidade, bem como a escolha de uma plataforma, sendo a escolhida o *GitHub* por ser a preferida por cerca de 90% da comunidade de desenvolvedores do mercado, além de oferecer todas as funcionalidades necessárias para o projeto de forma gratuita (CHACON, 2014).

O processo de estudo e aplicação dos *Git*, *GitHub* e *Heroku* são ilustrados na Figura 13.

Figura 13 – Fluxo de estudo e utilização do versionamento e do deploy

Fonte: Elaborado pelo autor

4.1.13 Integração MongoDB-Servidor

Para que a plataforma funcione é necessário armazenar e realizar transferências de dados do cliente para o servidor e vice-versa. É preciso então conectar a aplicação ao banco de dados MongoDB utilizando o Node.js [4]. Para isto foi utilizado o módulo *mongoose* e sua implementação é feita da forma apresentada nos códigos que seguem.

Código 10 - Conexão do back-end ao banco de dados

```
const mongoose = require('mongoose');

mongoose.set("useFindAndModify", false);

mongoose.connect(MONGO_URL, {
  useNewUrlParser: true,
```

[4] Curso-MEAN-Stack <https://www.udemy.com/course/angular-2-and-nodejs-the-practical-guide/>

```
  useCreateIndex: true,
  useUnifiedTopology: true,
});
```

Fonte: O autor

No trecho de Código 10 é demonstrado o estabelecimento de uma conexão com o banco de dados localmente. É feita, primeiramente, a importação do módulo *mongoose*, seguida pela definição dos valores para a conexão com o banco de dados.

Com a conexão estabelecida é possível então realizar facilmente os comandos de busca, inserção ou atualização de dados. No exemplo do Código 11 é feita uma busca de todos os cadernos listados na coleção *Notebooks*. O resultado da busca seria dado pelo parâmetro *note*.

Código 11 - Função para busca de um caderno no banco de dados

```
const Notebook = mongoose.model("notebook", {
  projectName: {
    type: String,
  },
  author: {
    type: String,
  },
  subject: {
    type: String,
  },
  pages: {
    type: Number,
  },
  tags: [],
});

async function findNotebook(info) {
  const note = await Notebook.findOne({
    projectName: info.projectName;
  }).then();
  return note;
}
```

Fonte: O autor

Com a aplicação respondendo às requisições e tendo estabelecido a conexão com o banco de dados fica possível desenvolver boa parte das funcionalidades, pois o usuário já pode navegar pelas páginas e receber ou enviar dados para o servidor. No trecho do Código 11 tem-se o exemplo da função de busca por um nome específico de caderno, que se encaixa no fluxo de conferência por cadernos já criados pelo usuário. Assim, se o resultado é encontrado, há uma atualização do caderno, inserindo mais imagens neste, caso contrário, cria-se um novo caderno com o nome buscado pela função.

Código 12 - Função para busca de palavra-chave no banco de dados

```
async function findKeyword(info) {
  const arrProjectName = await Notebook.find({
    projectName: { $regex: info.keywordSearch },
  }).then();
  const arrSubject = await Notebook.find({
    subject: { $regex: info.keywordSearch },
  }).then();
  const arrTags = await Notebook.find({
    tags: { $regex: info.keywordSearch },
  }).then();
  let arrSearchs = [arrProjectName, arrSubject, arrTags];
  let arrNotebooks = [];
  (() => {
    for (i = 0; i < arrSearchs.length; i++) {
      arrSearchs[i].map((note) => {
        arrNotebooks.push(note);
      });
    }
  })();
  if (arrNotebooks.length >= 1) {
    console.log("Keyword found in the DB:");
    console.log(arrNotebooks);
    return arrNotebooks;
  } else {
    console.log("No Notebook with this Keyword");
    return undefined;
  }
}
```

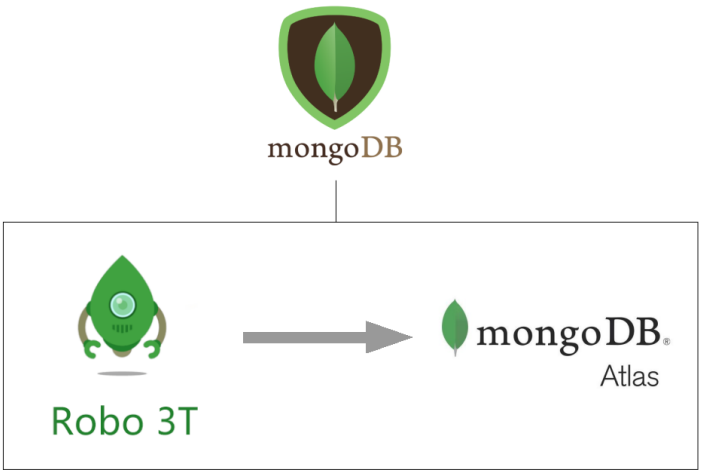
Fonte: O autor

Já no exemplo ilustrado no Código 12 há a função de busca por palavra-chave, função utilizada na busca por cadernos dentro de toda a plataforma e não só dentre os cadernos do próprio usuário. A busca se faz por conferência de *string* total ou parcial de nomes de cadernos, matérias ou *tags*.

4.1.14 *Atlas - MongoDB*

Após a implementação da maior parte das funções de manipulação do banco de dados, este foi movido para um sistema de nuvem, para que não houvesse qualquer dependência com a máquina local. No caso, foi-se escolhida a plataforma *Atlas*, que é um sistema de *clusters* dos mesmos criadores do MongoDB[5] e possui uma categoria de uso gratuito, como é ilustrado na Figura 14.

Figura 14 – Migração do banco de dados local para plataforma remota



Fonte: Elaborado pelo autor

4.1.15 Escolha do armazenamento remoto das imagens

Para o armazenamento das imagens em sistema de nuvem, foram tomadas decisão levando-se em consideração complexidade, utilização da ferramenta no mercado de trabalho, capacidade de processamento e disponibilidade da ferramenta no sistema de graduação. Sendo assim, optou-se pela utilização de: Google Drive, Google Cloud, Azure e AWS.

O Google Drive foi a primeira opção pensada pelo fato de o sistema de graduação da USP, atualmente, disponibilizar acesso ilimitado à ferramenta. Porém, após algum estudo, se mostrou inviável para o processo diante da complexidade, já que precisaria ser executado em uma máquina virtual após o *deploy* da aplicação e seria um potencial gargalo na importação das imagens.

Sendo assim, restaram Google Cloud, Azure e AWS, sendo os sistemas de nuvem mais utilizados no mercado de trabalho, respectivamente, das empresas Google, Microsoft e Amazon. Neste caso, a AWS é o sistema com mais tempo de mercado e, consequentemente, mais consolidação e mais materiais de apoio disponíveis na web para pesquisa. Além disso, foi o sistema com maior prazo para teste gratuito das funcionalidades, com prazo de 1 ano

[5] Curso MongoDB <https://www.udemy.com/course/mongodb-the-complete-developers-guide/>

para a maioria dos principais serviços, bem como foi o serviço que ofereceu maior crédito estudantil utilização em seus cursos.

A tomada de decisão está ilustrada na Figura 15.

Figura 15 – Escolha do armazenamento remoto das imagens

Fonte: Elaborado pelo autor

4.1.16 *AWS - S3 ou EC2*

Após a tomada de decisão de qual sistema de nuvem utilizar foi feita a seleção do serviço a ser utilizado, no caso, podendo ser o de criação de uma máquina virtual no repositório do tipo EC2 na AWS[6], bem como o repositório do tipo S3.

No longo prazo, a opção mais interessante seria a utilização do EC2 (*Elastic Compute Cloud*), já que oferece ao aluno oportunidade de aprender sobre outras áreas como *DevOps* para infraestrutura, bem como uma maior manipulação do sistema como um todo. Porém, pensando em prazo e curva de aprendizado, o S3 (*Simple Storage Service* ou Serviço de Armazenamento Simples) foi o escolhido, já que ofereceria uma solução mais simples que seria o suficiente para o projeto proposto, como ilustrado na Figura 16.

4.1.17 Integração AWS-Servidor

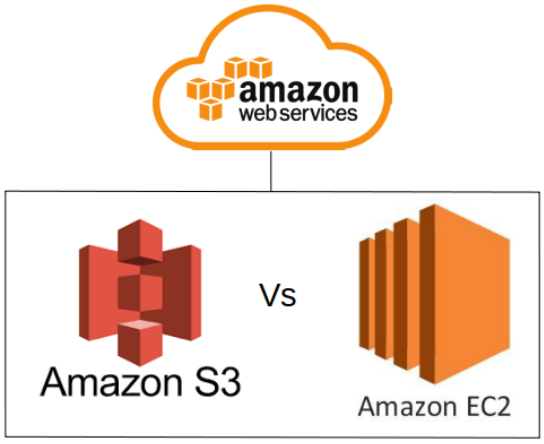
Tendo o sistema e a ferramenta selecionados, passou-se para a etapa de implementação no sistema *back-end*, substituindo o armazenamento local das imagens pelo armazenamento remoto.

Código 13 - Função para salvamento das imagens em nuvem

```
const multer = multerS3 = require("multer-s3"),
  AWS = require("aws-sdk");
const mongoose = require("../mongoDB/mongoose");
const configAWS = require("./config");
// AWS Info
```

6 Curso AWS <https://www.udemy.com/course/aws-certified-cloud-practitioner-new/>

Figura 16 – Escolha entre S3 ou EC2 na AWS



Fonte: Elaborado pelo autor

```
AWS.config.update({
  secretAccessKey: configAWS.AWS_SECRET_ACCESS_KEY,
  accessKeyId: configAWS.AWS_ACCESS_KEY,
  region: configAWS.AWS_DEFAULT_REGION,
});
S3_BUCKET = configAWS.BUCKET_NAME;
const s3 = new AWS.S3();
// File upload settings
const path = "/Programming/TCC/Actual/backend/uploads";
const fileInfo = {
  notebook: "NoFetch",
  newPages: "",
  i: 1,
};
let uploadAWS = multer({
  storage: multerS3({
    s3: s3,
    bucket: S3_BUCKET,
    key: function (req, file, cb) {
      setTimeout(() => {
        cb(null, file);
      }, 700);
    },
  }),
});
```

Fonte: O autor

No trecho do Código 13 é possível ver como foi feita a interação do sistema de nuvem da AWS, S3, com o *back-end*. No caso, foi feita a importação do módulo *multer-s3*, que tem a mesma intenção de uso do módulo *multer* anteriormente comentado, porém com o posterior armazenamento na AWS em vez de local. Tem-se a *AWS Info* que recebe os parâmetros de segurança da para o acesso à AWS, como *secretAccessKey, accessKeyId, region* e mais abaixo, tem-se o *S3_BUCKET* que recebe o nome do *bucket* criado no S3. Além disso, todo esses dados, que são sensíveis, foram armazenados no arquivo *.env*, que fica fora do fluxo de *push* para o *GitHub*.

4.1.18 Nomeando Imagens

A nomeação dos arquivos se mostrou de extrema importância, principalmente, porque no sistema de armazenamento S3 da AWS não há disponibilidade de funcionalidades de busca e filtro de nomes no serviço S3. Assim, optou-se por nomear os arquivos com o ID do *notebook* gerado pelo MongoDB no ato da criação de um caderno novo adicionado no número da página, na ordem de inclusão (ex: 60dbb3480c035019f8ebab26-1.jpg)

Código 14 - Função de Nomeação das imagens

```
const fileNameFunc = (file) => {
  let extension = file.originalname.split(".").pop();
  if (extension === "jpeg") {
    extension = "jpg";
  }
  let page = fileInfo.notebook.pages - fileInfo.newPages
  + fileInfo.i;
  const fileName = fileInfo.notebook._id + "-" + page
  + "." + extension;
  console.log(page);
  fileInfo.i++;
  if (fileInfo.i > fileInfo.newPages) {
    fileInfo.i = 1;
  }
  return fileName;
};
```

Fonte: O autor

No Código 14 tem-se a função responsável pela nomeação das imagens antes do envio das mesmas ao S3.

Primeiramente, a função separa a extensão do arquivo enviado e, depois, adiciona o ID e o número de páginas já existente no caderno, informações estas recebidas anteriormente pelo banco de dados.

4.1.19 Renderizando Informação do Banco de Dados

Ao se fazer uma busca por cadernos é necessário enviar a palavra-chave buscada do *front-end* para o *back-end*, posteriormente, para o banco de dados e, assim, retornar todo o fluxo até a interface com o usuário. Para tanto, como a comunicação *back-end* com o banco de dados já foi demonstrada, será dado foco à implementação no *fron-end* com o *back-end*.

Código 15 - Função no front-end para download dos arquivos PDF

```
download = (i) => {
  this.toastr.success('Seu caderno esta ficando pronto :)');
  this.http
    .post(this.downloadUrl, this.notes[i])
    .subscribe((responseData) => {});
```

Fonte: O autor

A função ilustrada no Código 15 é o *trigger* para iniciar o processo de *download* de um caderno, no lado do cliente.

No Código 16 é apresentado o trecho HTML referente ao componente de buscas, o *search.component*. Nele é possível observar duas das principais funcionalidades para renderização do *Angular*. A primeira é *[(ngModel)]="info.keywordSearch* que funciona como uma atualização em tempo real, um *event handler* entre variável e *input*, no caso, toda alteração feita no campo de *input* será atualizada na variável *info.keyworSearch* que se encontra no arquivo *Typescript* referente deste componente. A segunda é a **ngIf='filtersLoaded | async'*, em que há outro *event handler*, porém, com conferência de lógica *if*, a qual aguarda de forma assíncrona a variável *filtersLoaded*, que guarda uma *promise*, que, quando resolvida, executa o código HTML dentro dela. A terceira e última é a **ngFor='let note of notes; let i=index'*, a qual faz a leitura e renderização dos itens do *array notes*, também presente no *search.component.ts*.

Código 16 - HTML referente a página de buscas no Angular

```
<div class="wrapper">
  <img class="search-icon " id='search' (click)='search()'/>
  <input class="search" placeholder="Buscar Material"
  type="text" [(ngModel)]="info.keywordSearch"
    (keydown.enter)="search()" autofocus>
  <img class="clear-icon" id="clear" (click)='clear()' />
  <div class='notebooks'></div>
  <div *ngIf='filtersLoaded | async' class='items-box'>
    <div *ngFor='let note of notes; let i=index'
    class='item-box'>
      <div class='notebookBox'>
        <ul class='notesList'>
          <li class='projectName'> {{ note.projectName }}</li>
          <li class='subject'> {{ note.subject }}</li>
        </ul>
      </div>
      <img class='downloadIcon' id='{{ i }}'
      (click)='download(i)' alt='Download Notebook'
        src='assets/images/downloadIcon.png'>
    </div>
  </div>
</div>
```

Fonte: O autor

Já no Código 17 no lado do cliente, tem-se a função *search*, que se encontra no *search.component.ts*, a qual ao ser chamada executa uma requisição *http.post* enviando a palavra chave para o *back-end* que fará todo o tratamento de dados e retornará no mesmo endereço por meio de uma requisição *http.get* um vetor de objetos do tipo *notebooks* se existir algum caderno com uma palavra-chave totalmente ou parcialmente igual no banco de dados ou *null* se não existir. Após isso, a promessa *filtersLoaded* é resolvida, o que serve como *trigger* para as funções lógicas no HTML.

Código 17 - Função no cliente para envio de palavra-chave a ser buscada no banco de dados

```
readonly apiUrl = environment.apiURL;
readonly searchUrl = this.apiUrl + 'search';
readonly downloadUrl = this.apiUrl + 'download';

info = {
  keywordSearch: '',
};

notes = [];
filtersLoaded: Promise<boolean>;

search = () => {
  this.notes = [];
  this.http.post(this.searchUrl, this.info).
      subscribe((responseData) => {});
  this.http.get(this.searchUrl).subscribe((data) => {
    if (data !== null) {
      Object.values(data).map((note) => {
        console.log(note);
        this.notes.push(note);
      });
      this.filtersLoaded = Promise.resolve(true);
    } else {
      this.toastr.warning('Nenhum resultado encontrado :(');
    }
  });
  this.info.keywordSearch = '';
};
```

Fonte: O autor

No Código 18 tem-se as requisições HTTP no lado do servidor. No caso, ao se fazer a requisição do tipo *POST*, a função *search.searchkeyword()* recebe os parâmetros *req, res* referentes à requisição e a promessa que retorna da função é guardada em uma variável *promiseFoundNotesArr*. Após a finalização da função e consolidação do seu retorno, a resolução da promessa é enviada ao cliente por meio do *app.get()*.

Código 18 - Função no back-end para passada e recebimento de dados de busca de palavra-chave no banco de dados

```
let promiseFoundNotesArr = "null";

//Search keyword in DB
app.post("/api/search", (req, res) => {
  promiseFoundNotesArr = search.searchkeyword(req, res);
});

// Receive results from keyword Search
app.get("/api/search", (req, res) => {
  promiseFoundNotesArr.then((arrNotes) => {
    res.send(arrNotes);
  });
});
```

Fonte: O autor

4.1.20 Buscando Arquivos no S3-Bucket e baixando as imagens

Como anteriormente mencionado, o S3 não possui recursos de busca. Sendo assim, todos os nomes dos arquivos enviados à AWS precisariam ser salvos em algum lugar para que a busca fosse exata. Desta forma, chegou-se a duas opções: salvar o padrão de nome como *"ID + timestamp + extensão"* e salvar cada um dos nomes gerados no banco de dados ou *ID + número da página + extensão* e, assim, não salvar o nome de cada arquivo, mas sim o número total de páginas e utilizar um *loop* para iterar a busca.

Pensando em otimização de processamento e tamanho de banco de dados, escolheu-se a segunda opção, pois poderia reduzir o banco de dados e diminuir o número de requisições feitas a ele no *back-end*, adquirindo apenas um valor (páginas), em vez de diversas *strings*, uma para cada arquivo.

No Código 19 tem-se as funções responsáveis pela busca e pelo *download* das imagens na *AWS* referentes a um caderno. O processos foi iniciado pela importação dos módulos, *fs*, *AWS* e a importação do arquivo *config.js*, que faz a leitura das variáveis de ambiente, do *.env*, como vistos anteriormente. Após isso, foi configurada a conexão com a *AWS*, também como já mencionado em outra seção.

Código 19 - Função no back-end responsável por efetuar o download das imagens da AWS

```
const fs = require("fs"),
  AWS = require("aws-sdk")
const configAWS = require("./config");

// AWS Info
AWS.config.update({
  secretAccessKey: configAWS.AWS_SECRET_ACCESS_KEY,
  accessKeyId: configAWS.AWS_ACCESS_KEY,
  region: configAWS.AWS_DEFAULT_REGION,
});

S3_BUCKET = configAWS.BUCKET_NAME;
const s3 = new AWS.S3();

const homePath =
  process.env.HOME + "/Programming/TCC/Actual/backend/
  tempDownload/";

let arrFileNames = [];

async function downloadPdf(req, res) {
  let pages = req.body.pages;
  let id = req.body._id;
  let fileName = "";

  for (let i = 1; i <= pages; i++) {
    fileName = `${id}-${i}.jpg`;
    await getFilesFromAWS(fileName).then();
  }
  let pdfPath = await generatePDF(req.body.projectName).then();
  for (let i = 0; i < arrFileNames.length; i++) {
    const path = arrFileNames[i];
    console.log("path", path);
    await deleteFile(path).then();
  }
  arrFileNames = [];
  return pdfPath;
}
```

```
async function getFilesFromAWS(fileName) {
  try {
    console.log(fileName);
    const data = await s3
      .getObject({ Bucket: S3_BUCKET, Key: fileName })
      .promise();
    await saveTempFiles(`${homePath}${fileName}`, data.Body)
    .then(() => {
      arrFileNames.push(`${homePath}${fileName}`);
    });
  } catch (err) {
    console.log("File missing", fileName);
  }
}

async function saveTempFiles(name, buffer) {
  try {
    fs.writeFileSync(name, buffer);
  } catch (err) {
    console.log("Unable to save file:", name);
  }
}
```

Fonte: O autor

Após isso, tem-se a função *downloadPdf* que é chamada pela requisição HTTP feita a partir do *front-end*, enviando *req*. Este *req* já possui o objeto *notebook* vindo da busca por palavra-chave, contendo número de páginas do caderno e o ID, efetuada no *front-end*. Desta maneira, faz-se a iteração em *loop* já explicada anteriormente e, em cada *loop*, é feita a chamada da função *getFilesFromAWS*, que recebe o nome do arquivo buscado. A partir disso, é criado um *stream* de dados, *buffer* de binários (que é o único formato salvo pelo S3 na AWS) vindo da AWS, e é chamada a função *saveTempFiles*, que salva todas as imagens em uma pasta temporária no servidor, para que sejam, posteriormente, convertidas em formato PDF e, depois, baixadas pelo navegador do usuário.

4.1.21 Convertendo Imagens em PDF

Após as imagens serem baixadas temporariamente para o servidor, chama-se a função *generatePDF*, que recebe o nome do caderno especificado no comando vindo do

cliente.

Código 20 - Conversão das imagens temporárias em arquivo PDF

```
const imagesToPdf = require("images-to-pdf");
async function generatePDF(projectName) {
  try {
    let pdfPath = `${homePath}${projectName}.pdf`;
    imagesToPdf(arrFileNames, pdfPath);
    return pdfPath;
  } catch (err) {
    console.log("Unable to save file:", `${projectName}.pdf`);
  }
}
```

Fonte: O autor

Para efetuar a conversão, utilizou-se o módulo como ilustrado no Código 20, o *images-to-pdf*, que funciona recebendo um *array* com o *path* de todas as imagens que devem ser utilizadas no PDF e o *path* onde deve ser salvo o novo PDF. Assim, esse PDF é salvo na mesma pasta de arquivos temporários do servidor, como "nome do caderno + .pdf", e a função retorna seu *path*.

4.1.22 Baixando o Arquivo PDF

Após a criação do arquivo PDF, é preciso enviá-lo para *download* via *browser*.

Código 21 - Função no back-end responsável por gerenciar requisições HTTP para o download de PDFs

```
  let pdfPath = "";
  app.post("/api/download", (req, res) => {
    pdfPath = pdfDownloader.downloadPdf(req);
  });
  // Receive PDF
  app.get("/api/download", (req, res) => {
    pdfPath.then((path) => {
      res.download(path);
    });
  });
```

Fonte: O autor

Como ilustrado no Código 21, na requisição *POST*, no *back-end*, a promessa de *path* para o PDF é guardada em *pdfPath* e é resolvida na requisição *GET* do mesmo endereço e foi utilizado *res.download(path)* para efetuar o *download* do arquivo pelo navegador.

Código 22 - Função no cliente responsável pelo download do PDF na interface do navegador

```
download = (i) => {
  this.toastr.success('Seu caderno esta ficando pronto :)');
  this.http
    .post(this.downloadUrl, this.notes[i])
    .subscribe((responseData) => {});
  this.http
    .get(this.downloadUrl,{ responseType: 'blob' as 'json' })
    .subscribe((res) => {
      console.log(res);
      const blob = window.URL.createObjectURL(res);
      const link = document.createElement('a');
      link.href = blob;
      link.download = this.notes[i].projectName + '.pdf';
      link.click();
      window.URL.revokeObjectURL(blob);
      link.remove();
      this.http
        .post(this.downloadUrl + '/done', { info: true })
        .subscribe((data) => {});
    });
};
```

Fonte: O autor

Como ilustrado no Código 22, tem-se a função no *front-end* responsável pelo *trigger* e posterior *download* no *browser* recebendo os dados do *back-end*. No recebimento dos dados em *http.get* tem-se o recebimento dos dados com resposta no tipo *blob* em *json*. Assim, foi criada uma variável temporária *blob* para guardar uma *URL* criada pelo navegador utilizando a resposta do *back-end*, e criado um *link* clicável temporário, uma vez clicado no *link* o *download* se inicia. Após isso, é removida a *URL* e o *link* temporários.

Finalmente, é feita uma requisição *http.post* informando a finalização do processo de *download* para que os arquivos temporários no servidor, imagens e PDF, possam ser deletados.

4.1.23 *Hash* nas senhas

Ao realizar o registro, os dados do usuário são armazenados no banco de dados no mesmo formato em que foram inseridos caso não haja nenhum tratamento. Armazenar as senhas dos usuários de forma legível traria riscos à segurança e é, portanto, necessário que seja feito um *hash* nas mesmas. *Hash* pode ser entendido como uma forma de encriptação irreversível.

Para isso foi feito uso do módulo *bcrypt*.

Código 23 - Exemplo de criação de Hash com o módulo bcrypt

```
const bcrypt = require('bcrypt');

const password = 'senha123';
const saltRounds = 10;

bcrypt.hash(password, saltRounds, function(err, hash) {
  //fazer algo.
});
```

Fonte: O autor

O trecho do Código 23 ilustra a importação do módulo *bcrypt* seguida pela definição dos valores da *string* na qual vai ser feita um *hash* e do *saltRounds* que define o tempo necessário para realizar o processo, sendo o resultado mais complexo a medida que seu valor é aumentado. Por fim é feito o *hash* na *string*, gerando a *string hash* como resultado.

Utilizando o *hash*, pode-se armazenar as senhas dos usuários de forma segura.

4.1.24 Utilizando *session* para permanecer logado

Se faz necessário que, após realizar *login* no sistema, o usuário possua uma sessão para persistir os dados que indicam quem está logado na página. Sem isso o usuário ao fazer *login* teria apenas os dados de *login* verificados no momento e depois continuaria como se não estivesse logado e, não conseguindo realizar ações como a candidatura, por exemplo, onde é preciso saber quem está se candidatando.

Para resolver este problema fez-se uso do *express-session*. Ele cria as sessões, configura e as armazena no objeto que é recebido nas requisições. Com isso, sempre que o mesmo cliente fizer novas requisições o servidor terá as informações da sessão armazenadas e poderá acessá-las.

Código 24 - Exemplo de utilização do módulo express-session

```
let session = require('express-session');

app.use(session(
    {secret:'hms082jfowq89coen',
    resave:false,
    saveUninitialized:true}
));


function tryLogin(req, res) {
    sess = req.session;

        //..verifica os dados de login
        //busca o id do usuario caso login tenha sucesso..
        sess.user_id = user_id;
}
```

Fonte: O autor

No trecho ilustrado no Código 24 a função *tryLogin* é iniciada com *sess* recebendo o valor da sessão do usuário e, após feita a autenticação de *login* e buscado o *id* do usuário correspondente, essa *sess* recebe o parâmetro do *id*, e para acessar esse *id* a partir de então basta utilizar sess.user_id.

Com isso o usuário já pode permanecer logado durante o uso da aplicação.

4.1.25 Aplicação

Com todos esses módulos é possível criar uma aplicação, definir os endereços acessíveis pelo usuário e suas respectivas respostas, utilizar o banco de dados, salvar as senhas de modo seguro, fazer o *upload* das imagens para a nuvem, enviar dados do *back* para o *front-end*, manter sessões do usuário e efetuar a conversão e o *download* dos materiais. Isso basta para o desenvolvimento do projeto, sendo apenas uma questão de codificar o projeto em si.

4.2 Resultados

Com as tecnologias apresentadas foi possível a implementação das funcionalidades propostas no modelo UML de casos de uso definido para a plataforma, criando um sistema que permite o *upload* e compartilhamento de anotações. O presente trabalho tem

como foco principal o desenvolvimento das funcionalidades em si, não foram trabalhados aspectos estéticos da aplicação, cujas páginas ficaram com a aparência genérica e sem muita personalização.

As Figuras 17 a 20 ilustram algumas páginas do site, referentes a página de *login*, página de registro, página inicial do usuário e página de busca e *download* de materiais.

Figura 17 – Página de *login*

Fonte: Elaborado pelo autor

Figura 18 – Página de registro

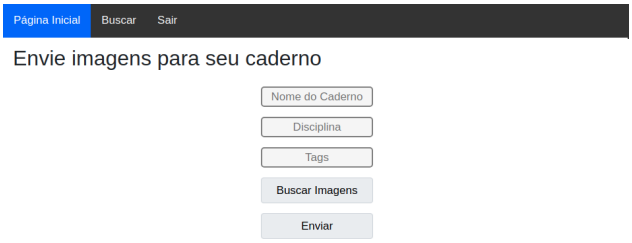
Fonte: Elaborado pelo autor

O código fonte do projeto pode ser encontrado em https://github.com/andre94cavalcante/TCC-BACKUP. O projeto está hospedado no Heroku e pode ser visto por meio do *link* https://tcc-andre.herokuapp.com/.

4.3 Considerações Finais

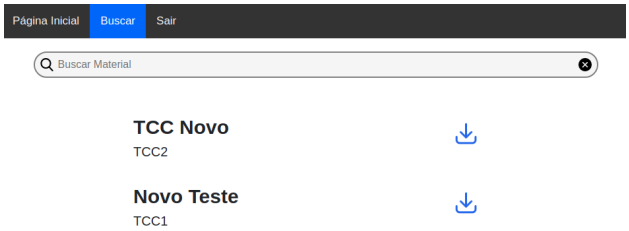
Este capítulo apresentou a plataforma desenvolvida, bem como os elementos necessários para o seu desenvolvimento. Além disso, foi possível realizar a criação não só das interfaces mas também da integração da interface web com o banco de dados, a comunicação do *back-end* com o sistema de nuvem e com a parte de *login* desenvolvidos.

Figura 19 – Página inicial do usuário



Fonte: Elaborado pelo autor

Figura 20 – Busca e *download* de materiais



Fonte: Elaborado pelo autor

O próximo capítulo trata das conclusões e dos trabalhos futuros.

5 CONCLUSÃO

Por meio do desenvolvimento da plataforma para compartilhamento de materiais de estudo, foi possível verificar as dificuldades relacionadas a um projeto de um software, desde seu planejamento até sua implementação. O trabalho proporcionou uma boa experiência a partir do uso de diferentes tecnologias para desenvolvimento web, tanto em partes relacionadas ao servidor como o Node.js, o banco de dados MongoDB e o repositório S3 do sistema de nuvem AWS, quanto na parte visual com HTML, CSS, Javascript e *Angular*.

Espera-se que o presente projeto sirva como base para a implantação de uma plataforma que ofereça ao corpo docente e discente da USP um local mais assertivo e dinâmico para busca de materiais de estudo.

Por fim, o curso de Engenharia Mecatrônica proporcionou-me o desenvolvimento das habilidades de aprendizado e de busca por soluções e tecnologias que foram de absoluta importância neste trabalho. O trabalho aqui apresentado extrapolou as áreas estudadas no curso, o que permitiu uma gama de conhecimentos adquiridos e crescimento deste aluno.

5.1 Trabalhos futuros

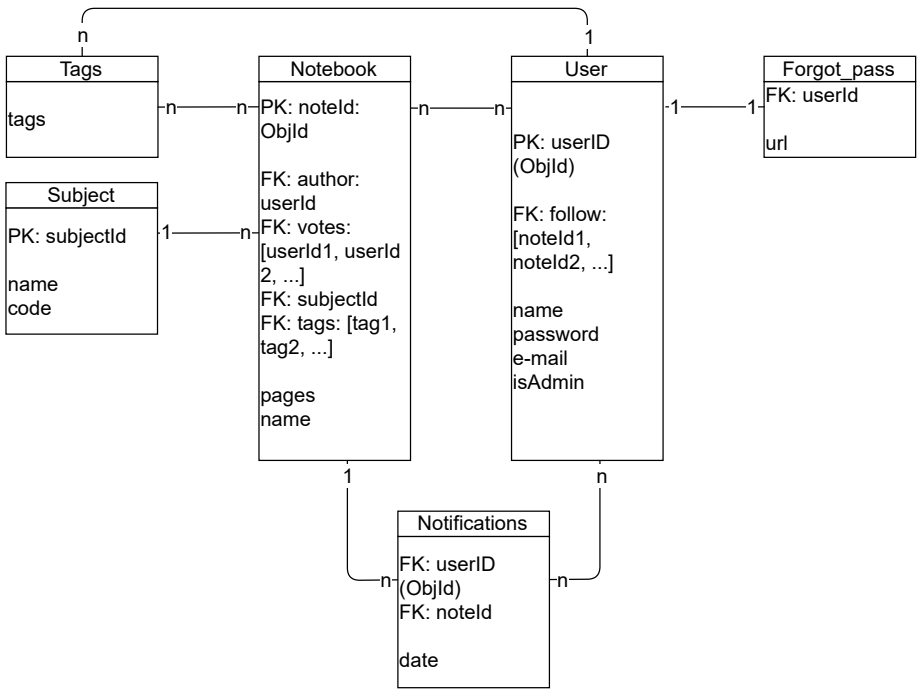
Neste trabalho foi desenvolvido um site responsivo para compartilhamento de materiais de estudo dentro da comunidade USP. Como sugestões para projetos futuros pode-se elaborar trabalhos que se aprofundem mais no desenvolvimento *front-end*, tornando as páginas mais personalizadas e envolvendo aspectos de *User Experience* e *User Interface*. Pode-se também criar aplicativos nativos como uma outra opção para o usuário, expandindo o alcance da aplicação na comunidade da USP.

Além disso, é possível implementar um sistema de notificação de atualização de cadernos de interesse, bem como um sistema de envio de *e-mails* automáticos para essas notificações. Também, é possível construir um sistema semelhante ao de uma rede social para comunicação entre usuários, possibilitando comentários nos cadernos, e, até mesmo, complementar materiais, bem como fornecer ao usuário a possibilidade de 'seguir' um usuário e seus materiais ou a possibilidade avaliar a qualidade de um material.

Ademais, é ilustrado na Figura 21 o primeiro modelo UML proposto para o trabalho, que sofreu alterações com o decorrer do projeto devido ao prazo de entrega.

No mais, a plataforma desenvolvida pode ser ainda extrapolada para uso em qualquer outro tipo de necessidade de compartilhamento de materiais para envio e recebimento entre usuários. Sendo o ponto mais interessante, o fato de o produtor e o consumidor serem a mesma persona, divergindo somente em qual disciplina cada um consegue performar melhor ou necessita mais de ajuda.

Figura 21 – Modelo UML Inicial



Fonte: Elaborado pelo autor

5.2 Problemas encontrados

Nesta seção são apresentados os problemas encontrados durante a execução deste trabalho.

- Configuração Servidor *Node*
  - Tentativa de exportar o *app = express()* com *module.exports*. Esse problema foi resolvido com a separação das requisições *HTTP* no servidor em um arquivo que exporta todo o *app*
- Configuração *Angular*
  - Problemas na montagem da aplicação. Esse problema foi resolvido a partir do aprendizado sobre o funcionamento do arquivo *angular.json*
- Comunicação *front-end back-end*
  - Tentativa de comunicação por passagem de valores localmente; Falta de conhecimento sobre integração do *Angular* com o servidor *Node*; Inicializar o servidor em ambiente de produção e inicializar o *front-end* em ambiente de desenvolvimento; e Problemas de *Headers* e *CORS* no navegador. Esses problemas foram

solucionados a partir do: Estudo de requisições HTTP; Aprender sobre processo de construção da aplicação - *ng build*; Inicializar o *front-end* em ambiente de produção ao inicializar o servidor; e Utilização do *middleware CORS* na API.

- Mongoose
  - Falta de conhecimento sobre comportamento assíncrono no *Javascript*; Problemas de passagem de resultados do banco de dados para o *back-end*; Problemas de concorrência; *Callback Hell*
    ; e *Promises* com algoritmo longo e confuso. Esses problemas foram solucionados a partir de: Estudo de *Promises* com *Async/Await*; e Utilização de *return* para funções assíncronas após utilização do banco de dados.
- Deploy - Heroku
  - Achar que precisaria da porta do endereço, que é dinâmica no *Heroku* e não pode ser acessada pelo *front-end*, para fazer as requisições *HTTP* assim como é feito no ambiente local. Como solução tem-se a utilização do próprio endereço da *URL* para fazer as requisições em ambiente remoto de produção.

REFERÊNCIAS

CHACON, S. **Pro Git**. [S.l.]: Apress, 2014.

HOWS, D. **Introdução ao MongoDB**. [S.l.]: Editora Novatec; Apress, 2015.

NOGUEIRA, T. d. C. et al. Estudo comparativo da experiência de usuários cegos e videntes no design web responsivo e não responsivo. Universidade Federal de Goiás, 2015.

ZEMEL, T. **Web Design Responsivo: páginas adaptáveis para todos os dispositivos**. [S.l.]: Editora Casa do Código, 2015.