



Universidade Federal  
do Rio de Janeiro  

---

Escola Politécnica

# **APLICAÇÃO WEB DE PLANEJAMENTO PESSOAL UTILIZANDO CLOUD COMPUTING**

Bianca Modesto Coelho

Projeto de Graduação apresentado ao Curso de Engenharia Eletrônica e de Computação da Escola Politécnica, Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Engenheiro.

Orientador: Flávio Luis de Mello

Rio de Janeiro

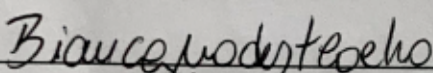
Março de 2021

# APLICAÇÃO WEB DE PLANEJAMENTO PESSOAL UTILIZANDO CLOUD COMPUTING


Bianca Modesto Coelho

PROJETO DE GRADUAÇÃO SUBMETIDO AO CORPO DOCENTE DO CURSO DE ENGENHARIA ELETRÔNICA E DE COMPUTAÇÃO DA ESCOLA POLITÉCNICA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE ENGENHEIRO ELETRÔNICO E DE COMPUTAÇÃO

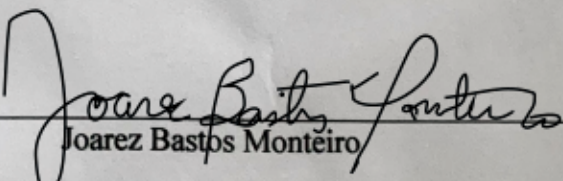
Autor:

  
Bianca Modesto Coelho

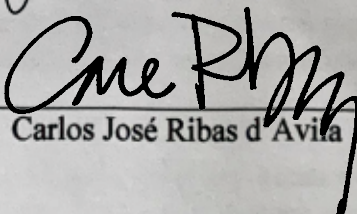
Orientador:

  
Flávio Luis de Mello

Examinador:

  
Joarez Bastos Monteiro

Examinador:

  
Carlos José Ribas d'Avila

Rio de Janeiro – RJ, Brasil

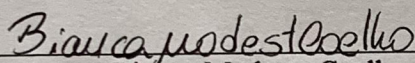
Março de 2021



### Declaração de Autoria e de Direitos

Eu, *Bianca Modesto Coelho* CPF 089.379.796-08, autor da monografia *Aplicação Web de Planejamento Pessoal utilizando Cloud Computing*, subscrevo para os devidos fins, as seguintes informações:

1. O autor declara que o trabalho apresentado na disciplina de Projeto de Graduação da Escola Politécnica da UFRJ é de sua autoria, sendo original em forma e conteúdo.
2. Excetuam-se do item 1. eventuais transcrições de texto, figuras, tabelas, conceitos e ideias, que identifiquem claramente a fonte original, explicitando as autorizações obtidas dos respectivos proprietários, quando necessárias.
3. O autor permite que a UFRJ, por um prazo indeterminado, efetue em qualquer mídia de divulgação, a publicação do trabalho acadêmico em sua totalidade, ou em parte. Essa autorização não envolve ônus de qualquer natureza à UFRJ, ou aos seus representantes.
4. O autor pode, excepcionalmente, encaminhar à Comissão de Projeto de Graduação, a não divulgação do material, por um prazo máximo de 01 (um) ano, improrrogável, a contar da data de defesa, desde que o pedido seja justificado, e solicitado antecipadamente, por escrito, à Congregação da Escola Politécnica.
5. O autor declara, ainda, ter a capacidade jurídica para a prática do presente ato, assim como ter conhecimento do teor da presente Declaração, estando ciente das sanções e punições legais, no que tange a cópia parcial, ou total, de obra intelectual, o que se configura como violação do direito autoral previsto no Código Penal Brasileiro no art.184 e art.299, bem como na Lei 9.610.
6. O autor é o único responsável pelo conteúdo apresentado nos trabalhos acadêmicos publicados, não cabendo à UFRJ, aos seus representantes, ou ao(s) orientador(es), qualquer responsabilização/ indenização nesse sentido.
7. Por ser verdade, firmo a presente declaração.

  
Bianca Modesto Coelho

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO

Escola Politécnica – Departamento de Eletrônica e de Computação

Centro de Tecnologia, bloco H, sala H-217, Cidade Universitária

Rio de Janeiro – RJ CEP 21949-900

Este exemplar é de propriedade da Universidade Federal do Rio de Janeiro, que poderá incluí-lo em base de dados, armazenar em computador, microfilmear ou adotar qualquer forma de arquivamento.

É permitida a menção, reprodução parcial ou integral e a transmissão entre bibliotecas deste trabalho, sem modificação de seu texto, em qualquer meio que esteja ou venha a ser fixado, para pesquisa acadêmica, comentários e citações, desde que sem finalidade comercial e que seja feita a referência bibliográfica completa.

Os conceitos expressos neste trabalho são de responsabilidade do(s) autor(es).

## **DEDICATÓRIA**

Ao meu pai e mãe, obrigada por todo o apoio nessa jornada.

## **AGRADECIMENTO**

Foram inúmeras as pessoas que contribuíram para a minha trajetória na UFRJ, e as levarei na memória com todo carinho. Gostaria de agradecer primeiramente a todos os membros da equipe MinervaBots, foi a primeira instituição dentro da UFRJ que me acolheu, foram 2 anos na equipe de robótica que trouxeram muito aprendizado em diversos setores da minha vida e foram essenciais pro meu crescimento profissional e pessoal. Também gostaria de agradecer a todos da equipe de Cheerleading nos dois anos em que fiz parte, com muito esforço, treino e dedicação, representava a nossa instituição e nossas cores com esmero. Um abraço em especial a Beatriz Guimarães que levarei para a vida.

O caminho até aqui foi árduo, de muito estudo e dificuldades, porém todos foram sendo superados pouco a pouco, gostaria de agradecer a todos os professores que fizeram parte dessa trajetória, foram muitos os ensinamentos e após esses anos de UFRJ me sinto preparada pra qualquer outro desafio que venha surgir. Também gostaria de agradecer a uma pessoa em especial, que estava sempre ao meu lado em todos os momentos, os difíceis e os alegres, me incentivando e vice versa, Artur Knupp, você foi essencial nesses anos pra mim e espero que essa amizade perdure por muitos anos.

Finalmente gostaria de agradecer aos meus pais, Tania e Renato, que sempre me incentivaram e apoiaram até aqui, sem vocês nada disso seria possível. Aos meus irmãos, Renata e Gustavo, vocês são um dos motivos de eu querer me superar sempre.

## **RESUMO**

Este trabalho consiste no desenvolvimento e implantação em produção de um sistema simples, prático e intuitivo de Planejamento Pessoal. O sistema foi desenvolvido utilizando as principais tecnologias utilizadas atualmente além da implantação em ambiente Cloud, proporcionando uma alta disponibilidade e escalabilidade. Visando a criação de um sistema baseado em microserviços, foram estudados os produtos oferecidos pela Amazon Web Services para o desenho do mesmo e suas funcionalidades. Portanto esse projeto mostra as decisões de arquitetura, tecnologias e modelo de dados para atingir o objetivo do sistema, assim como o resultado final do desenvolvimento, que pode ser visto no Capítulo 5.

Palavras-Chave: React, .NET Core, Docker, AWS

## **ABSTRACT**

This project consists in the development and implementation in production of a simple, practical and intuitive Personal Planning system. The system was developed using the main useful technologies in addition to deployment in a Cloud environment, providing high availability and scalability. Aiming at creating a system based on microservices, the products offered by Amazon Web Services for the design of the system and its characteristics were studied. Therefore, this project shows how architecture, technology and data model decisions to achieve the system objective, as well as the final result of the development, which can be seen in Chapter 5.

Keywords: React, .NET Core, Docker, AWS



## **SIGLAS**

API – Application Programming Interface

AWS – Amazon Web Services

CLI - Interface de Linha de Comando

DOM – Document Object Model

FedRAMP – Federal Risk And Authorization Management Program

FISMA – Federal Information Security Modernization

HIPAA – Health Insurance Portability and Accountability

HITECH – Health Technology for Economic and Clinical Health Act

IaaS – Infrastructure as a Service

NoSQL – Not Only Structured Query Language

PCI-DSS – Payment Card Industry – Data Security Standard

S3 – Simple Storage Service

SaaS – Software as a Service

TTL – Time to Live

UI – User Interface

# Sumário

<b>1</b>	<b>Introdução</b>	<b>1</b>
1.1	Tema .....	1
1.2	Delimitação .....	1
1.3	Justificativa .....	1
1.4	Objetivos .....	2
1.5	Metodologia .....	3
1.6	Descrição .....	3
<b>2</b>	<b>Fundamentação Teórica</b>	<b>4</b>
2.1	Kanban .....	4
2.2	Container .....	5
2.3	– .NET Core 3.0 .....	7
2.4	– React .....	8
2.5	– Amazon Web Services .....	11
2.5.1	– Introdução .....	11
2.5.2	– DynamoDB. ....	12
2.5.3	– Elastic Load Balancer .....	13
2.5.4	– Elastic Container Service .....	13
2.5.5	– Amazon S3. ....	14
2.5.6	– AWS Lambda. ....	15
2.5.7	– CloudWatch. ....	15
<b>3</b>	<b>Implementação</b>	<b>17</b>
3.1	Cadastro e Autenticacao. ....	17

3.2 - Kanban .....	19
3.2.1 – Modelo de Dados .....	19
3.2.2 – Busca e Armazenamento de Board. ....	21
3.2.3 – Redux .....	22
3.2.4 - Componentes .....	23
3.3 – Burndown .....	25
3.3.1 – Modelo de Dados. ....	25
3.3.2 – Implementação da atualização de busca .....	26
3.4 – Deploy em Homologação .....	28
3.4.1 – Backend. ....	28
3.4.2 – Frontend. ....	32
<b>4      Apresentação do Sistema</b>	<b>33</b>
4.1 – Diagrama de Telas .....	33
4.2 – Cadastro e Autenticação. ....	33
4.3 – Tela Principal .....	35
4.4 – Dashboard e Boards. ....	36
4.5 – Highlights .....	39
4.6 – Burndown .....	39
..	
<b>5      Conclusão</b>	<b>41</b>
<b>6      Bibliografia</b>	<b>42</b>

# Lista de Figuras

2.1 – Arquitetura Docker. . . . .	7
2.2 – Componentes na tela de Kanban do PlanejAqui. . . . .	9
2.3 – Flux. . . . .	10
2.4 – Arquitetura Website estático com S3 . . . . .	14
2.5 – Funcionamento do CloudWatch. . . . .	16
3.1 – Esquema da requisição de cadastro. . . . .	18
3.2 – Esquema da requisição de autenticação. . . . .	19
3.3 – Modelo de Dados Kanban. . . . .	20
3.4 – Esquema busca e armazenamento do Boar. . . . .	22
3.5 – Reducers e suas respectivas actionsos reducers. . . . .	22
3.6 - Diagrama de components. . . . .	24
3.7 – Modelo de Dados Burndown backend. . . . .	26
3.8 – Modelo de dados Burndown frontend. . . . .	26
3.9 – Esquema da atualização de dados do Burndown. . . . .	27
3.10 – Esquema de busca de dados do Burndown. . . . .	28
3.11 – Dockerfile. . . . .	28
3.12 – Publicando imagem docker no repositório ECR. . . . .	29
3.13 – Configurações do Cluster. . . . .	30
3.14 – Load Balancer da aplicação. . . . .	31
3.15 – Requisição HTTP no microserviço. . . . .	31
3.16 – Configurações do Bucket. . . . .	32
4.1 – Diagrama de telas. . . . .	33
4.2 – Tela de cadastro. . . . .	34
4.3 – Dados insuficientes. . . . .	34
4.4 – Usuário com mesmo e-mail. . . . .	34
4.5 – Cadastro com sucesso. . . . .	34
4.6 – Tela de Autenticação. . . . .	35
4.7 – Erro na autenticação. . . . .	35
4.8 – Tela Principal. . . . .	35
4.9 – Erro ao adicionar Board. . . . .	36
4.10 – Boards criados. . . . .	37

4.11 – Board selecionado. ....	37
4.12 – Adição de lista. ....	38
4.13 – Listas adicionadas. ....	38
4.14 – Adição e edição de Card. ....	38
4.15 – Página de Highlights. ....	39
4.16 – Detalhes da página de highlights. ....	39
4.17 – Burndown chart. ....	40
4.18 – Entradas de dados Burndown. ....	40



# Capítulo 1

## Introdução

### 1.1 – Tema

O tema do trabalho é o desenvolvimento de uma aplicação intuitiva e prática para Planejamento Pessoal utilizando as tecnologias mais utilizadas no mercado atualmente, além de disponibilizá-la em ambiente *Cloud*, que possui um baixo custo de manutenção, alta disponibilidade e escalabilidade, garantindo uma aplicação robusta.

### 1.2 – Delimitação

A aplicação terá como principal interface um *Kanban* para que o usuário possa organizar seus projetos de maneira individual. Também estará disponível nessa interface, funcionalidades como impedimentos para conclusão, *burndown* de tarefas, quantidade de horas estimadas e realizadas para cada tarefa, entre outros. A interface será disponibilizada em ambiente de Cloud, com o *backend* e *frontend* separados a nível de aplicação, de forma alinhada com o que o mercado tem difundido nos últimos anos, utilizando segregação de responsabilidade e o conceito de micro serviços.

### 1.3 – Justificativa

Realizar projetos novos sejam pessoais ou em conjunto, exige uma série de habilidades para que se atinja o objetivo com sucesso. A organização é um dos pilares indispensáveis para o sucesso de um projeto, por isso utilizar da tecnologia para auxiliar nessa etapa é uma boa opção, visto que na internet é mais fácil manter os dados e passos do projeto do que, por exemplo, em papéis físicos. Por isso alinhar o conceito de *Kanban* com uma plataforma online é interessante e tem sido muito utilizada por empresas que implementam a técnica Ágil para desenvolver projetos.

Com o avanço da tecnologia, a necessidade das plataformas serem escaláveis, terem alta disponibilidade, independente da demanda, e serem de baixo custo, tem feito empresas de diversos setores, nem sempre com foco em tecnologia, a migrar suas soluções para *Cloud Computing*. Assim são garantidos todos os benefícios que uma infraestrutura física em data centers não consegue proporcionar, evitando falhas por alto consumo em tempos sazonais.

Um grande exemplo das vantagens da utilização de arquiteturas em *Cloud* são as plataformas de *E-Commerce*, nos períodos de *BackFriday*, quando a demanda dos produtos e serviços é significativamente maior do que em outros períodos do ano. Se a infraestrutura fosse baseada nos períodos sazonais de alta demanda para suportar o volume de requisições, haveria um desperdício de processamento durante o resto do ano.

Com *Cloud Computing* isso é evitável, pois diversas plataformas que oferecem serviços na nuvem proveem o conceito *On-Demand*, ou seja, o custo é, grande parte das vezes, pela demanda, como os serviços do tipo SaaS. Para o caso de produtos IaaS, é possível configurar o processamento e escalabilidade dos serviços de acordo com a época do ano ou número de requisições, entre outras métricas flexíveis.

## 1.4 – Objetivos

O objetivo geral é buscar as melhores e mais modernas arquiteturas de *Cloud Computing* para implementar uma aplicação Web de planejamento pessoal de tarefas, obtendo baixo custo, escalabilidade e disponibilidade da aplicação.

Os objetivos específicos são:

- (1) Criação da aplicação de *Backend* com todos os métodos necessários para a implementação das funcionalidades da aplicação Web.
- (2) Criação da aplicação de *FrontEnd* com todas as telas e conexões com o *Backend*.
- (3) Teste unitário e integrado de toda a aplicação para garantir o funcionamento sem bugs.
- (4) Implementação de toda a estrutura da aplicação na AWS, tanto o *Backend* quanto o *Frontend*. Nesta etapa também será necessário realizar testes integrados para validar o funcionamento da aplicação.

## 1.5 – Metodologia

A primeira etapa consiste no desenvolvimento da interface de *backend* da aplicação, que terá a lógica e a conexão com o banco de dados. O banco de dados escolhido para a aplicação foi uma base *NoSQL* da própria AWS, o *DynamoDB*.

A primeira etapa segue concomitante à segunda etapa, pois à medida que as funcionalidades são aplicadas no *backend*, o *frontend* será desenvolvido para validá-las. Portanto a cronologia será a criação de *backend* e *frontend* de cada funcionalidade, e após finalizar esse desenvolvimento, passa-se para a próxima funcionalidade.

A segunda etapa, como dito anteriormente, segue concomitante à primeira etapa, o objetivo é utilizar a biblioteca *React* para criar uma interface amigável e intuitiva. Afim de criar uma boa UX, haverá uma pesquisa em sites semelhantes para avaliar a usabilidade e servir de inspiração no desenvolvimento.

Ao longo do desenvolvimento, haverá o teste das funcionalidades apresentadas, porém os testes de integração entre os micros serviços será realizado apenas na terceira etapa. Nessa etapa haverá uma pessoa de fora do projeto para utilizar a aplicação a fim de encontrar possíveis falhas e funcionalidades não intuitivas, para que o desenvolvedor consiga ter uma visão melhor sobre as melhorias, e assim, corrigi-las.

A quarta etapa consiste na implantação de toda aplicação no ambiente AWS. Para isso será necessário a criação dos componentes de IaaS (*Infrastructure as a Service*) e SaaS (*Software as a Service*) que serão utilizados pela aplicação, bem como criação das estruturas de rede e de segurança.

## 1.6 – Descrição

O capítulo 2 consiste na fundamentação teórica para a realização de todo o projeto, desde a linguagem/framework utilizado para desenvolver o *Backend* e *Frontend* até uma explicação sobre os serviços abordados na AWS que foram necessários para a implantação da aplicação.

# Capítulo 2

## Fundamentação Teórica

### 2.1 – Kanban

Metodologias Ágeis são um conjunto de metodologias cujo objetivo é acelerar os processos de desenvolvimento de software. A princípio as Metodologias Ágeis eram empregadas apenas em desenvolvimento de software, tendo um manifesto bastante conhecido [2], porém outras áreas como marketing e finanças passaram a empregar essas metodologias em seus projetos. E uma das formas mais utilizadas para organizar projetos *Agile* é o *Kanban*.

O *Kanban* nasceu, inicialmente, como um sistema de controle de estoques, criado pela Toyota, para garantir um método de produção eficiente, evitando o acúmulo de produtos em estoque e, por outro lado, evitando também a falta dos mesmos. A implementação do sistema se dá através de um quadro com colunas e cartões coloridos, em que as colunas representam os produtos ou os status de fabricação e as cores o nível de urgência para a produção de cada produto [1]. Atualmente o *Kanban* tem sido amplamente utilizado por diversas áreas para auxiliar a gestão de projeto por ser um sistema ágil e visual.

Não existe um formato correto para a construção de um quadro *Kanban*, a formatação das colunas depende das necessidades do projeto e pode ser adaptado para cada caso. Um exemplo encontrado comumente em projetos são quadros com as seguintes colunas:

1. Tarefas
2. Planejado
3. Desenvolvendo
4. Testando
5. Implantando
6. Feito

Mas também poderia é comum encontrar quadros mais simples com as colunas:

1. Tarefas
2. Em andamento
3. Feito

A ideia é que cada cartão do *Kanban* corresponda a uma tarefa, que passa de coluna em coluna durante o processo até ser concluído. Os cartões podem ser coloridos, para que sejam identificados por área, ou categoria, dependendo do que o time de projeto definir [3]. Também é comum encontrar no *Kanban* uma coluna de Impedimentos, em que são incluídas tarefas que possuem algum impedimento seja técnico, financeiro e etc.

Além de auxiliar projetos corporativos, o *Kanban* também tem sido amplamente utilizado para a organização de tarefas pessoais, como tarefas no trabalho, faculdade, escola, leitura, entre outros.

O *Kanban* pode ser construído em um quadro físico com cartões, mas também pode ser construído em aplicações web, utilizando o mesmo conceito do quadro físico, porém com a vantagem de poder ser acessado de qualquer lugar, apenas necessitando de uma conexão com a Internet, além de ter funcionalidades como notificação, gráficos em tempo real do andamento do projeto, entre outros.

## **2.2 – Container**

Com a evolução da tecnologia e a necessidade cada vez maior de criar softwares robustos e com características cada vez mais elaboradas, a solução de arquitetura monolítica, em que toda a lógica de uma solução existia em um único aplicativo, se tornou inviável. Além de trazer uma complexidade atribuir todas as funcionalidades a uma única aplicação, há uma dificuldade enorme de manutenção, visto que para incluir uma nova funcionalidade ou correção de bug, é necessário fazer a implantação de toda a aplicação, o que pode acarretar diversos problemas.

A arquitetura de micro serviços, em contraste com a monolítica, tem como objetivo dividir uma solução em pequenas aplicações, separadas por responsabilidade e que existem de forma independente. Isso permite incluir novas funcionalidades sem precisar implantar toda a solução, dessa forma há mais agilidade, inclusive em relação a equipes de desenvolvimento, que podem ser separadas por responsabilidades de forma



prática. Também tem a vantagem de separar o processamento da solução por aplicação, sem precisar sobre dimensionar toda a infraestrutura por causa de uma parte da solução.

O *Container* é um ambiente isolado contido em um servidor, em que os componentes de software necessários para que uma determinada aplicação funcione nesse ambiente são empacotados. O *Docker* é uma solução *Open Source* de gerenciamento de *Containers* muito utilizada no mercado de tecnologia atualmente. É através dele que os *Containers* são estruturados no servidor.

Diferentemente de uma Máquina Virtual, que depende da criação de um sistema operacional completo para cada módulo, o *Docker* empacota as partes de softwares de um sistema de arquivo completo e reúne os recursos essenciais para a sua boa execução [17]. Com todos os recursos isolados sob medida para a aplicação, há a garantia de que o sistema funcione independente de qual ambiente esteja rodando.

O *Docker* utiliza a infraestrutura do kernel do seu sistema operacional para não instalar outro SO completo, assim, utiliza bibliotecas e binários já existentes abstraindo as camadas de SO das suas aplicações utilizando a *Docker Engine*, que é uma camada de administração. Como não existe um sistema operacional em cada *Container*, é possível iniciá-los em menos de 1 segundo.

Para criar um container *Docker* é necessário criar uma imagem, que nada mais é que um modelo de um sistema de arquivos. As imagens são criadas através de um processo de *build* que é descrito pelo *DockerFile* (arquivo sem extensão) que contém o passo a passo para geração da imagem [18]. Após a geração da imagem é possível armazená-las utilizando um repositório a escolha. Existem diversas opções, a AWS, por exemplo, fornece o serviço ECR (*Elastic Container Registry*) em que é possível gerenciar e armazenar, de forma fácil, imagens *Docker*.

Já a interação com o *Docker* via linha de comando ou interface gráfica, é feita através do *Docker Client*.

Portanto o *Docker* é dividido em três partes: *Docker Engine*, *Docker Registry* e *Docker Client*. A Figura 2.1 ilustra a arquitetura do *Docker*.

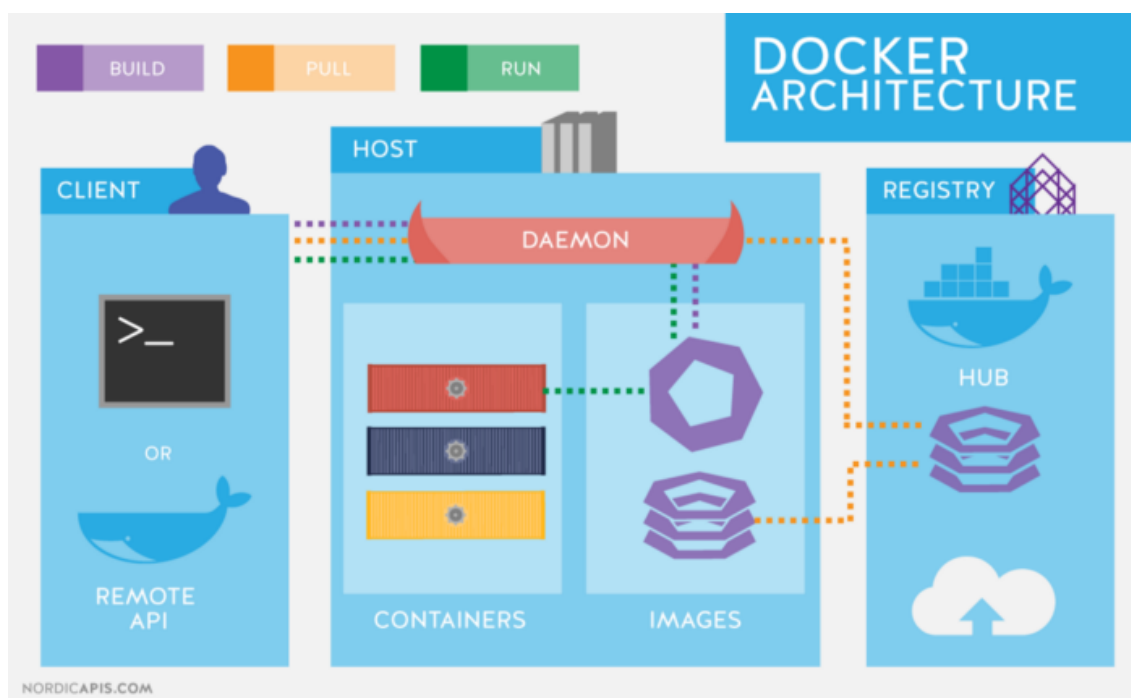


Figura 2.1 – Arquitetura Docker. Fonte: [18]

## 2.3 – .NET Core 3.0

Uma etapa importante ao iniciar o desenvolvimento de uma aplicação é a definição da linguagem a ser utilizada. Essa decisão pode ser tomada por familiaridade da equipe de desenvolvimento, plataformas e tecnologias disponíveis para a linguagem, entre outros. As linguagens mais utilizadas no mercado para desenvolvimento de software são *Java*, *C*, *Python*, *Java Script*, entre outras.

A linguagem C possui duas extensões conhecidas e muito utilizadas para desenvolvimento de software, o C++ e o C#, que são linguagens com orientação a objetos. O C# é multiparadigma, de tipagem forte e foi desenvolvida pela Microsoft como parte da plataforma .NET. A plataforma .NET é de software livre e gratuito, e com ela é possível criar diversos tipos de aplicações como: Web, APIs da Web e Micro serviço, aplicativos nativos em nuvem, aplicativos para dispositivos móveis. Jogos, Internet das Coisas, aprendizado de máquina, serviços do Windows, entre outros. O .NET também é multiplataforma, portanto é possível criar aplicações para *Windows*, *MacOS*, *Linux*, *Android*, *iOS*, *watchOS*, e suporta arquiteturas de processador *x64*, *x86*, *ARM32* e *ARM64* [19].

Para utilizar o .NET existem alguns ambientes disponíveis, o *Visual Studio* é um deles, que tem uma ampla funcionalidade e possui a versão *Community Edition* que é

gratuita para estudantes, colaboradores de software livre e pessoas físicas. O *Visual Studio* possui diversas extensões, uma delas que facilita o desenvolvimento e testes para desenvolvedores que utilizam a *Amazon Web Services* é o *AWS Toolkit*, que fornece conexão com a AWS e uma série de ferramentas de interação com os serviços fornecidos por ela. A desvantagem de se utilizar o *Visual Studio* é que a plataforma funciona apenas para *Windows*.

Existem diferentes implementações da plataforma *.NET* disponíveis para a criação de aplicativos, a primeira implementação da *Microsoft* foi o *.NET Framework*, que tem como desvantagem a compatibilidade apenas com o sistema operacional *Windows*. Em 2014 a *Microsoft* lançou uma evolução da plataforma, agora compatível também com *Linux* e *MacOS* denominada *.NET Core*, que atualmente está em sua versão 3.1.

Para desenvolver aplicações em *.NET Core* é necessário instalar a *SDK* do mesmo, que é um conjunto de bibliotecas e ferramentas que contém componentes necessários para criar e executar aplicativos: O CLI (Interface de Linha de Comando) do *.NET Core*, bibliotecas e *runtime* do *.NET Core* e o driver *dotnet*, que fornece comandos para trabalhar com projetos *.NET*.

Para o desenvolvimento específico de aplicações web, existe o *ASP.NET Core*, ela é uma extensão do *.NET Core* em que é possível criar serviços *Web* e *Web Apps*, *Apps IoT* e *Backends Mobile*. Uma vantagem de se utilizar as plataformas *.NET Core* é o fato de serem modulares, pois existe uma gama de pacotes disponíveis através do *NuGet*, que permite instalar, atualizar e remover bibliotecas. A plataforma *ASP.NET Core* facilita o desenvolvimento de aplicações, principalmente se, para o desenvolvimento, o *Visual Studio* for utilizado.

## 2.4 – React

O *React* é uma biblioteca do *JavaScript* de código aberto que se tornou popular por permitir a criação de forma intuitiva uma interface de usuário (UI). O *React* foi desenvolvido pelo *Facebook*, e desde então é usado em aplicativos difundidos e famosos por fornecerem uma UI intuitiva e amigável como *Netflix*, *Airbnb*, *American Express*, *Facebook*, *WhatsApp*, *Instagram* e *eBay* [4].

O *React* facilita trabalhar com componentes interativos e reutilizáveis para interfaces de usuário.

Ele trabalha com o conceito de Componentes. Um componente *React* tem como finalidade separar responsabilidade, acoplando funcionalidades específicas a este e que pode ser reutilizado em diversas páginas ao longo de toda a aplicação, reduzindo a complexidade do código, portanto facilitando o desenvolvimento e a manutenção. Um exemplo de página utilizando componentes pode ser vista na Figura 2.2.

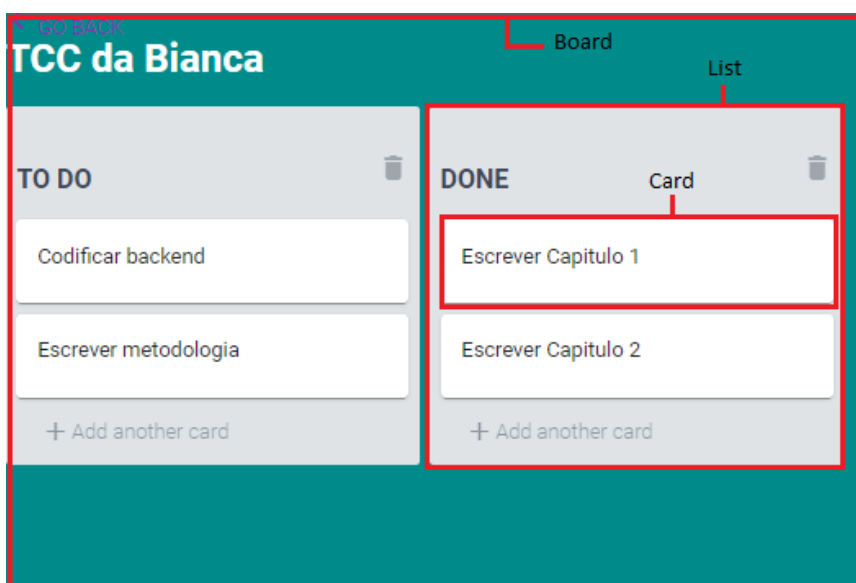


Figura 2.2 – Componentes na tela de Kanban do PlanejAqui

Uma vantagem relevante da utilização do *React* é a eficiência como ele trabalha com o DOM (*Document Object Model*). O DOM é uma convenção multiplataforma de linguagem de programação para representação e interação com objetos em documentos HTML. O DOM fornece uma representação do documento com um grupo estruturado de nós e objetos, possuindo várias propriedades e métodos. Os nós também podem ter manipuladores de eventos que lhe são inerentes, e uma vez que um evento é acionado, os manipuladores de eventos são executados. Essencialmente, ele conecta páginas web a scripts ou linguagens de programação. [5]

O que torna o *React* eficiente é o fato de utilizar um *Virtual DOM*, em que os componentes vivem. Toda vez que um componente é renderizado, o *React* atualiza o *Virtual DOM* de cada componente já renderizado e busca as mudanças, a partir disso ele compara o DOM com o *Virtual DOM* para verificar as mudanças e atualiza na DOM apenas os componentes que realmente mudaram. Dessa forma há um enorme ganho de performance. [6]

Nos componentes *React*, existe um atributo chamado *props*, em que dados podem ser incluídos para serem renderizados de um componente para outro. Porém se houver uma grande iteração desses dados por diversos componentes, fica inviável a manipulação do mesmo. Para resolver esse problema, existe uma biblioteca de armazenamento de estados de aplicações chamada *Redux*.

O *Redux* nasceu através da implementação do *Flux*. O *Flux* é uma arquitetura que visa justamente resolver o problema que temos ao criar uma cadeia complexa de componentes que trafegam dados entre si. Basicamente ele é dividido em 3 partes: *Dispatcher*, *View* e *Store* (ver Figura 2.2).



Figura 2.3 – Flux

Na parte da *View* estão os componentes, em que os dados passam de pai para filho, criando um fluxo unidirecional. Os dados e a lógica para modificar os dados ficam centralizados na *Store*, portanto é possível separar a lógica da aplicação de toda a parte visual. Já o *Dispatcher* funciona como uma central de comunicação.

Para que um dado seja modificado na aplicação, a ação (*Action*) é acionada pela *View* através de um botão ou algum outro evento, a *action* é direcionada ao *Dispatcher* que aciona a *Store* para executar a ação de modificação do dado, após a realização da tarefa em questão. A *View* recebe os dados alterados a partir da store e será propagado por todos os componentes. [7]

O *Redux* implementa toda essa arquitetura, e facilita todo o fluxo de dados entre os componentes. Dessa forma cada componente tem acesso a todos os dados da aplicação. No caso do *Redux*, o *Dispatcher* é nomeado de *Reducer*, cada dado da store deve ter seu próprio *reducer* e é encarregado de lidar com todas as ações, como algum componente pedindo para alterar algum dado da store. [8]



Uma biblioteca popular no desenvolvimento utilizando *React* é o *Material UI*, ela possui uma vasta quantidade de componentes *React* para um desenvolvimento ágil e fácil. Aplicações conhecidas como *Netflix*, *Amazon*, *Coursera*, entre outros, utilizam os componentes do *Material UI* para a implementação do design. A vantagem de sua utilização é a velocidade para o desenvolvimento da interface com o usuário, a qualidade visual e padronização dos componentes para uma UX eficiente. A sua utilização é fácil e possui uma documentação vasta com exemplos e *templates*. Toda a documentação com exemplos, *templates* e informações adicionais se encontram em [9].

## 2.5 – Amazon Web Services

### 2.5.1 – Introdução

O termo *Cloud Computing* surgiu em 1997 em uma palestra ministrada por Ramnath Chellapa. Já o conceito da tecnologia é associado a John McCarthy que, nos anos 60, já havia discutido sobre computação compartilhada, em que um computador poderia ser utilizado por dois ou mais usuários. Existem diversos rumores sobre quem foi o pioneiro a aplicar a computação em nuvem pela primeira vez, alguns afirmam ter sido a *Amazon*, outros o *Google*. Sabe-se que entre 2006 e 2008 essa tecnologia começou a ser oferecida comercialmente. [10]

A ideia em torno de *Cloud Computing*, e que é exatamente o que a *Amazon Web Services* (AWS) oferece, é o aluguel do espaço e poder computacional, não exigindo de seus clientes ter um datacenter potente para atender as necessidades das aplicações de suas empresas. Manter um Datacenter é custoso, visto que não existe a flexibilidade de processamento, vantagem que a computação na nuvem oferece. Um bom exemplo dessa vantagem são os eventos sazonais que podem ocorrer ao longo do ano e que subutilizam ou sobrecarregam os servidores em um datacenter. Na *Blackfriday* por exemplo, plataformas de *E-Commerces* necessitam de muito mais processamento do que se comparado com as outras épocas do ano. Utilizar um serviço de Cloud para essas plataformas é vantajoso visto que há a possibilidade de escalar as aplicações, seja utilizando métricas de sazonalidade seja utilizando métricas de utilização, a depender da arquitetura e definição de cada usuário.

A utilização de *Cloud Computing* é tão eficiente que grandes plataformas utilizam seus recursos para hospedar suas aplicações. A companhia de turismo *CVC*, o aplicativo de entrega de comida e mercado *iFood*, a plataforma de streaming *Netflix*, o banco *Nubank* entre outros, são exemplos de plataformas que utilizam os serviços da AWS para hospedar suas aplicações.

Existem três modelos conhecidos de nuvem que podem ser utilizados na AWS. O IaaS (*Infrastructure as a Service*), PaaS (*Platform as a Service*) e SaaS (*Software as a Service*).

O modelo de IaaS permite que a empresa contrate toda a infraestrutura em nível de hardware, podendo escolher processador, memória, armazenamento entre outros. Nessa modalidade, comumente o preço é contabilizado sob a quantidade de servidores, dados armazenados e etc. Portanto é como uma contratação de um data center em que o cliente paga apenas o que usar.

Já no modelo PaaS, toda a infraestrutura subjacente (hardware e sistemas operacionais) são fornecidos, fazendo com que as empresas possam concentrar apenas na implantação e gerenciamento das suas aplicações. Dessa forma a AWS fornece o planejamento de capacidade, manutenção do software, e outros serviços envolvendo a execução da aplicação. [11]

O modelo SaaS é o mais completo, que é executado e gerenciado pelo provedor. Portanto o cliente precisa se preocupar apenas em como usará o software. Nesse modelo há uma grande vantagem com relação a escalabilidade e praticidade.

### **2.5.2 – DynamoDB**

Bancos *NoSQL* são uma classe de banco de dados que fornecem um mecanismo diferente dos bancos relacionais para seu armazenamento e recuperação de dados. Enquanto banco de dados *SQL* possuem uma ideia tabular com relacionamento entre as tabelas, o banco *NoSQL* possui uma ideia de documento, em que os dados possuem uma chave e valor armazenados em uma tabela não necessariamente estruturada.

O *Amazon DynamoDB* é um banco de dados *NoSQL* que oferece um desempenho de milissegundos com um dígito em qualquer escala de demanda. O *DynamoDB* pode processar mais de 10 trilhões de solicitações por dia e comportar picos de mais de 20 milhões de solicitações por segundo [12]. Além de escalável, não é

necessário provisionar servidores nem manter ou operar, o que facilita a utilização com um alto desempenho e alta disponibilidade. Além disso, para utilização menos robusta, é possível provisionar sua capacidade e para utilizações em maior escala há a possibilidade de configurar por demanda, dessa forma há uma otimização dos custos.

### 2.5.3 – Elastic Load Balancer

*Load Balancer* é uma configuração utilizada para redistribuir requisições entre dois ou mais *web server*, repartindo recursos como banda e hardware para garantir convergência, alta disponibilidade do serviço e maior tolerância a falhas [15].

O *Elastic Load Balancer* é um serviço da AWS que dimensiona o *Load Balancer* à medida que o tráfego da aplicação muda com o passar do tempo [16]. Assim como a maioria dos serviços AWS, o pagamento é feito com base na utilização do serviço.

### 2.5.4 – Elastic Container Service

O *Amazon Elastic Container Service* (ECS) é um serviço de orquestração de *containers*. A orquestração de *containers* automatiza a implantação, gerenciamento, escala e rede de *containers*. Portanto ele gerencia todo o ciclo de vida, monitorando a integridade do container.

Uma das vantagens em se utilizar o ECS é o fato de poder utilizar o *AWS Fargate*, que é um mecanismo de computação sem servidor, eliminando, assim, a necessidade de provisionar e gerenciar servidores. O *Fargate* aloca a quantidade certa de computação, eliminando a necessidade de escolher instâncias e ajustar a escala da capacidade do cluster.

O cliente só paga pelos recursos exigidos para execução dos containers [14], o que otimiza o processamento e consequentemente os custos. Uma outra forma de utilizar o ECS seria criando uma instância EC2 (*Elastic Compute Cloud*), porém toda a infraestrutura incluindo provisionamento teria que ser dimensionada pelo cliente.

Portanto utilizar o ECS facilita a utilização de *containers*. Além disso, possui suporte ao *Docker*, mantém a disponibilidade e permite a escalabilidade, visando atender os requisitos de capacidade da aplicação.

O ECS é integrado a recursos como *Elastic Load Balancer*, que é bastante utilizado nessa arquitetura de *containers*. A cobrança para utilizar os recursos do ECS é o mesmo que se paga ao utilizar instâncias EC2, e o pagamento é feito conforme o uso. [14]

### 2.5.5 – Amazon S3

O *Amazon Simple Storage Service* (S3) é um serviço de armazenamento de objetos que oferece escalabilidade, disponibilidade de dados, segurança e performance [20]. O S3 pode ser utilizado para diversas finalidades como:

- Armazenamento de backups. Quando existe uma tabela de grande uso do *DynamoDB*, é possível acoplar um TTL ligado a uma coluna dos dados com formato *epoch* que indica a data de expiração do dado. Utilizando uma Lambda com o *Amazon Kinesis Firehose* é possível fazer o envio desses dados ao S3.
- Processamento de arquivos. É possível incluir uma trigger no S3 que é acionada toda vez que um evento ocorre no repositório, como inserção de arquivo, e, portanto, realizar uma ação a partir desse evento.
- Análises de *Data Lake* e *Big Data*, visto que suporta uma alta quantidade de dados e fornece diversos outros serviços compatíveis para a recuperação, tratamento e análise dos mesmos.
- Hospedar um *Website Estático*. Possibilidade de publicar aplicações em que não há processamento do lado do servidor, como por exemplo na arquitetura vista na figura 2.3. Portanto para uma arquitetura em que o frontend e o backend da aplicação são totalmente desacoplados, o S3 é uma solução prática, simples e barata para hospedar sites. Basta realizar algumas configurações simples seguindo o passo-a-passo que pode ser visto em [20].

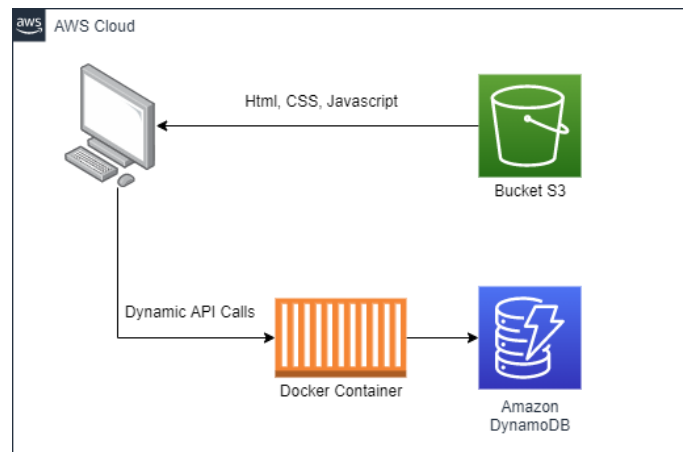


Figura 2.4 – Arquitetura Website estático com S3

O S3 é escalável, tem alta disponibilidade, tem recursos de segurança altamente confiáveis seguindo os programas de conformidade de segurança como PCI-DSS, HIPAA/HITECH, FedRAMP, EU Data Protection Directive e FISMA.

### 2.5.6 – AWS Lambda

O *AWS Lambda* é um dos serviços AWS que fazem parte do modelo SaaS, ele permite que um código seja executado sem a necessidade de provisão ou gerenciamento de infraestrutura, e o pagamento é sobre o tempo de computação utilizado. É possível integrá-lo com diversos outros serviços AWS, e também chamar um *Lambda* através de qualquer outra aplicação web, tornando-a versátil.

Uma grande vantagem do *AWS Lambda* é o fato de ser escalável sem que o cliente tenha que se preocupar com o gerenciamento de recursos, o próprio serviço se encarrega de alterar a escala baseado em métricas.

Existem diversos casos de uso para o *Lambda*, uma delas, mencionada na seção 2.5.5 é a utilização da mesma como *trigger* para o S3, permitindo o processamento de arquivos em tempo real. Também é possível criar uma *trigger* em um *DynamoDB* utilizando um *lambda*, portanto qualquer evento que aconteça na tabela, como modificação, inclusão ou deleção, a *lambda* é acionada para executar determinado processo.

Outro exemplo é utilizar o *CloudWatch*, que será abordado na seção 2.5.7, como uma *trigger* para a *Lambda*, podendo criar rotinas em horários específicos sem a utilização, por exemplo, de um serviço que funciona durante todo o tempo validando o



horário para decidir se executa a rotina, e com isso há uma economia de processamento e consequentemente é menos custoso.

### 2.5.7 - CloudWatch

O *CloudWatch* é um serviço de monitoramento que fornece dados e métricas em que é possível monitorar aplicações, realizar ações a partir de métricas de desempenho ou erros nas mesmas, como por exemplo envio de alertas. Ele coleta dados em forma de logs, métricas e eventos e exibe todos esses dados de forma unificada e simples. Também é possível criar regras para executar ações em determinado período, é possível configurar em quais horários do dia há a execução, os dias da semana e até do mês.

As vantagens de se usar o *CloudWatch* é poder obter e visualizar métricas de diversas aplicações com infraestruturas distintas em uma única plataforma, o que facilita a manutenção e a rapidez na ação para cenários de mau funcionamento. O esquema de funcionamento do *CloudWatch* pode ser visualizado na figura 2.4.

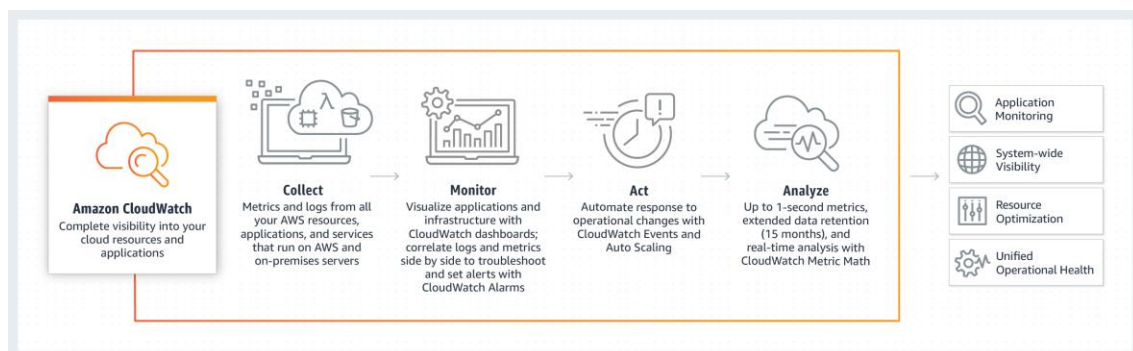


Figura 2.5 – Funcionamento do CloudWatch

# Capítulo 3

## Implementação

Para a implementação da aplicação com todas as suas funcionalidades, foram criadas três aplicações, sendo duas de *backend* e uma de *frontend*.

A aplicação principal de *backend* é um API em *.NET Core 3.1* que se comunica via HTTP usando o padrão REST. Ela faz toda a interface com o banco de dados e implementa a lógica e transformação de dados da aplicação para a comunicação com o *frontend*. As funcionalidades que a API implementa são a de cadastro e autenticação, e toda a comunicação direta entre as ações do usuário no *frontend*, como por exemplo criação de boards e atualização do mesmo.

A segunda aplicação de *backend* é uma *AWS Lambda* implementada em *.NET Core 3.1*. Ela é responsável por gerenciar e armazenar todos os estados das tarefas salvas no *Kanban*, e é através dela que conseguimos ter uma linha do tempo exata com o tempo estimado e realizado das tarefas no *Kanban*.

A aplicação de *frontend* que faz toda a interface gráfica com o usuário foi implementada em *React*, utilizando bibliotecas de interface gráfica intuitivas e usando *Redux* para implementar de forma eficiente todas as interações entre os componentes presentes na aplicação.

### 3.1 – Cadastro e Autenticação

Foi criada uma tela inicial com duas visualizações, a de cadastro e autenticação. Nesta tela é possível alternar entre as duas telas, quando a tela de cadastro é utilizada, existem três campos obrigatórios para cadastro, o e-mail, nome e senha, e possui um campo opcional para informar o número de telefone.

No momento do cadastro, a aplicação do *frontend* realiza uma chamada HTTP utilizando o método POST na rota `planejaaqui/login` ao *backend*, que por sua vez faz uma busca na base de dados verificando se já existe um usuário com o mesmo e-mail. Visto que o e-mail é um atributo chave, só é possível ter um usuário por e-mail. Caso não exista um usuário cadastrado com esse e-mail, os dados são salvos na tabela

PlanejaAqui-Users contida no *DynamoDB* e retorna um *Success* com código 200 para o *frontend*. Se o usuário estiver cadastrado, um *BadRequest* com código 400 é retornado e uma mensagem de erro é mostrada ao usuário, como visto no esquema da Figura 3.1.

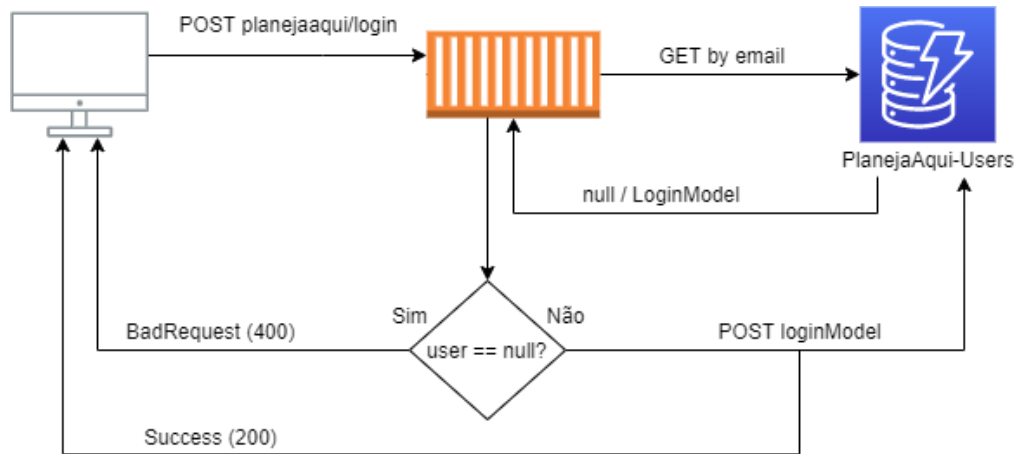


Figura 3.1 – Esquema da requisição de cadastro

Para a autenticação, o *frontend* realiza uma chamada HTTP utilizando o método GET na rota *planejaaqui/users/{email}* passando o e-mail como *Path Parameter* e a senha na *QueryString password*.

O *backend* por sua vez busca o dado na tabela utilizando o e-mail como chave e caso o usuário não exista, o *NotFound* com código 404 é retornado ao *frontend*. Caso exista, valida se o *password* informado pelo *frontend* corresponde ao que existe na tabela, em caso afirmativo o status *Success* com código 200 é retornado, caso não um *BadRequest* é retornado como pode ser visto na Figura 3.2.

O *frontend* redireciona o usuário para a tela de início dos boards de rota *admin/dashboard* caso receba um 200, caso contrário envia uma mensagem informando que o usuário não existe ou a senha está incorreta, por motivos de segurança os dois cenários possuem a mesma mensagem para evitar que um *hacker* consiga diferenciar se o usuário não existe ou se é a senha que está incorreta.

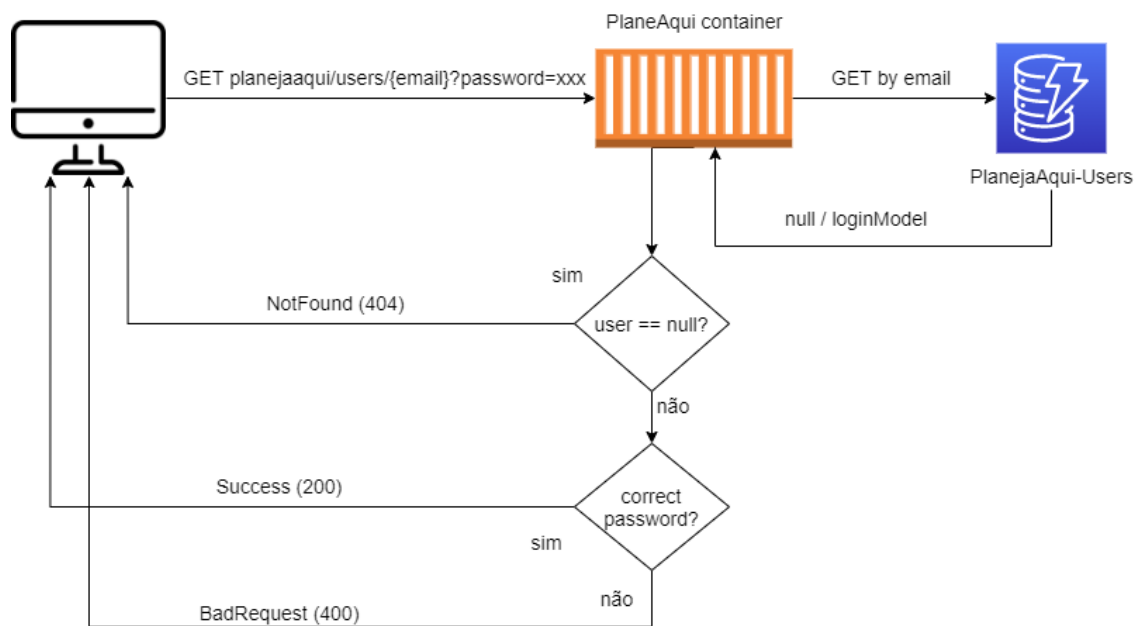


Figura 3.2 – Esquema da requisição de autenticação

Para permitir a navegação pelas páginas da aplicação, os dados do usuário são armazenados em um *session storage* utilizando um *key* do tipo JWT. Quando o usuário utiliza a funcionalidade de logout ou fecha a sessão, a *key* é deletada do *storage* e a autenticação passa a ser necessária novamente.

Para obter os dados do usuário através das páginas sem ter a necessidade de passá-las via *props* pelos componentes, foi utilizado um *redux* específico para o usuário, nesse sentido duas *actions* foram criadas, a *updateUser* para atualizar os dados do usuário e a *getUser* para busca-las através de qualquer componente caso necessário.

## 3.2 – Kanban

### 3.2.1 – Modelo de Dados

Primeiramente, antes da implementação, o modelo de dados foi definido. Visto que a quantidade de boards, listas e cards são variáveis, utilizar uma base *NoSQL* facilitou a montagem do modelo, visto que é necessária apenas uma tabela para armazenar todos os dados, dispensando a criação de uma tabela para cada item mencionado e seus respectivos relacionamentos. O modelo de dados pode ser visto na Figura 3.3.

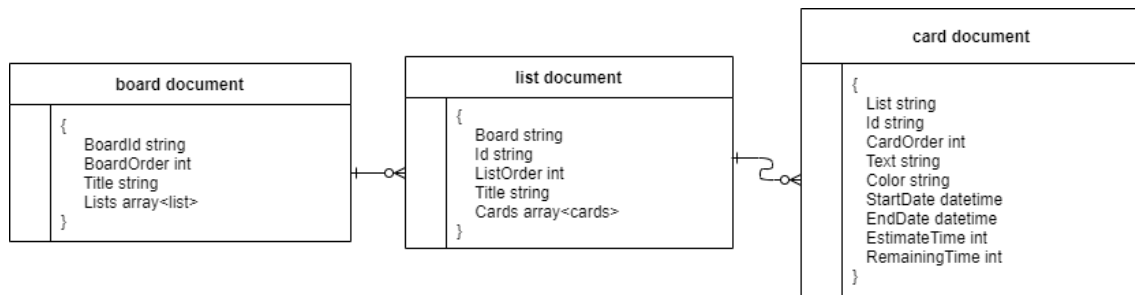


Figura 3.3 – Modelo de Dados Kanban

A tabela PlanejaAqui-Boards foi criada no *DynamoDB* com sua chave primária sendo o e-mail, o que é importante visto que no *DynamoDB* a busca de dados só é eficiente caso se disponha da chave primária ou criando-se índices para filtrar atributos específicos.

Utilizar o e-mail como chave garante que não será preciso a criação de índices na tabela, que é mais custoso, e apenas uma query é necessária para se obter os dados dos boards. Segue os atributos no modelo do Kanban no *backend*:

- *Boards*, que é uma lista do atributo *Board*. Contém todos os boards pertencentes ao e-mail.
  - *Board*, que é um objeto contendo os dados de um Board. Dentro desse objeto quatro atributos, o *BoardId*, *BoardOrder*, *Title* e *Lists*
  - *BoardId*, uma *string* que identifica unicamente o Board, com a base de dados *NoSQL* a alternativa para criar identificadores únicos é a utilização do *Guid*.
  - *BoardOrder*, um inteiro que identifica a ordem na qual o board está disposto na lista de boards.
  - *Title*, uma *string* com o texto imputado pelo usuário como sendo o nome do board.
- *Lists*, uma lista do atributo *List*, contém os dados das listas presentes naquele board.
  - *List*, objeto contendo os dados de uma lista. Possui cinco atributos, o *Board*, *Id*, *ListOrder*, *Title* e *Cards*.
  - *Board*, uma *string* que está no modelo de *List* e equivale ao *BoardId*.
  - *Id*, um identificador único do tipo *string* para identificar a lista.
  - *ListOrder*, inteiro para identificar a ordem da lista no *Board*.
  - *Title*, que é o nome da lista dada pelo usuário.

- *Cards*, uma lista de atributos do atributo *Card*, em que estão todos os cards contidos dentro da lista.
  - *Card*, objeto contendo os dados do Card, possui nove atributos, *CardOrder*, *Id*, *List*, *Text*, *Color*, *StartDate*, *EndDate*, *EstimateTime* e *RemainingTime*.
  - *CardOrder*, que indica a ordem em que o *Card* aparece na lista.
  - *List*, que é o Id da lista ao qual o card pertence.
  - *Text*, uma string com a descrição contida no card.
  - *Color*, que é opcional e identifica a cor do marcador de *Card*.
  - *StartDate*, que identifica a data inicial do desenvolvimento da tarefa.
  - *EndDate* que identifica a data final de desenvolvimento da tarefa.
  - *EstimateTime*, inteiro responsável por identificar as horas estimadas para finalização da tarefa.
  - *RemaningTime*, inteiro responsável por identificar a quantidade de horas remanescentes da tarefa

Esse é o modelo que a base de dados reconhece e mantém armazenada. O que o *frontend* enxerga é diferente e, portanto, para que os dados cheguem nele, corretamente, o *backend* faz uma transformação dos dados para se adequar ao modelo.

Como pode-se observar, apesar de as listas estarem dispostas na estrutura do seu board, a necessidade da inclusão do atributo de *BoardId* se fez necessária pois o *frontend* visualiza os dados de forma separada. O modelo de dados do *frontend* é composto por:

- *Boards*, que é um dicionário cuja chave é o *Id* do *board* e o valor é um objeto contendo os atributos *Id*, *Title* e uma lista de *string* chamada *Lists* contendo os *Ids* das listas em ordem.
- *Lists*, que é um dicionário cuja chave é o *Id* da lista e o valor um objeto contendo o *Id* da lista, o *Id* do board ao qual a lista pertence e uma lista de strings que contem os *Ids* dos cards pertencentes a lista de forma ordenada.
- *Cards*, que é um dicionário cuja chave é o *Id* do *card* e o valor um objeto contendo o *Id* do card, o *Id* da lista a qual pertence e o texto.
- *BoardOrder*, que é uma lista de *string* contendo os *Ids* dos *boards* de forma ordenada.

O modelo do *frontend* e do *backend* são diferentes por causa da forma como são estruturados, para construir um *redux* de boards seria mais complexo utilizar um modelo como o da base de dados, visto que um *reducer* é criado para cada componente. Portanto foi uma decisão de projeto fazer essa transformação de dados dentro do *backend* para não tornar o *frontend* complexo.

### 3.2.2 – Busca e Armazenamento do Board

O *backend* do *Kanban* possui dois métodos disponíveis, o método POST e GET na rota `planejaaqui/users/{email}/boards` em que foi utilizado um conceito de *API Rest* para sua construção, com o *path parameter* sendo o email, e os boards com o recurso a ser acessado que pertence ao e-mail especificado.

Para o POST os dados vindos do *frontend* sofrem uma transformação para o formato do modelo do *backend* e, posteriormente, são armazenados, o método de GET, o oposto, os dados são transformados para o modelo do *frontend* e são enviados ao *backend*, o status 200 é enviado para o *frontend* e caso os dados não existam, a lista e os dicionários serão enviados, porém com dados vazios.

O método GET é acionado pelo *frontend* no momento da autenticação do usuário e seus dados são armazenados na estrutura de *redux* do *Kanban* para que toda a aplicação possa ter acesso aos mesmos sem a necessidade de passagem de dados por parâmetros.

O método POST é acionado quando o usuário clica no botão em formato de *disket* presente na aba de Boards, em que todo o estado da estrutura de boards é enviada para o *backend*. A Figura 3.4 mostra o esquema mencionado para busca e inserção de dados dos boards.

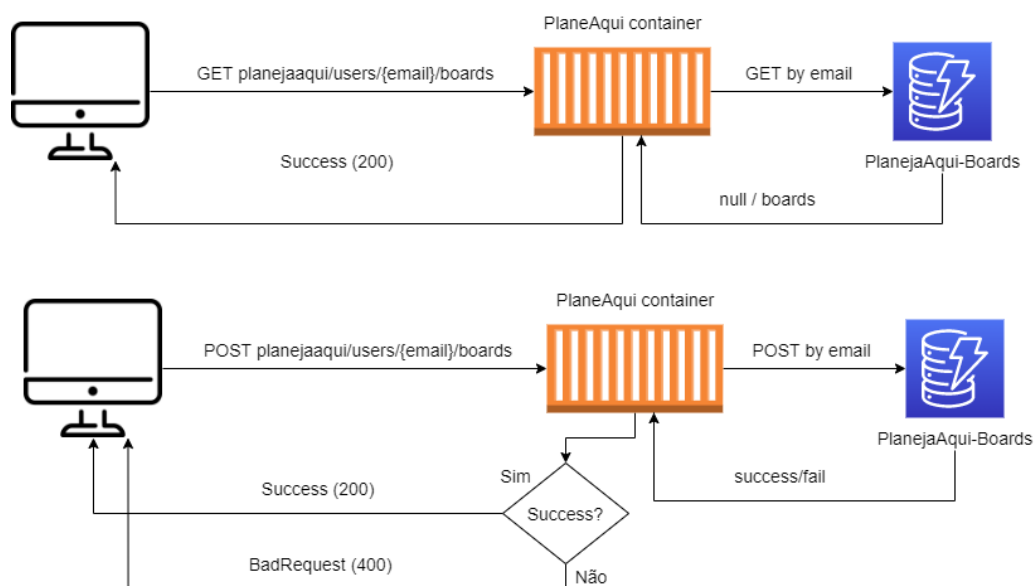


Figura 3.4 – Esquema busca e armazenamento do Board

### 3.2.3 – Redux

Como dito anteriormente, um *Redux* foi criado especificamente para o *Kanban*. Para isso foram criados os *reducers* *activeBoardReducer*, o *boardOrderReducer*, o *boardsReducer*, o *cardsReducer* e o *listsReducer*, como visto na Figura 3.5.

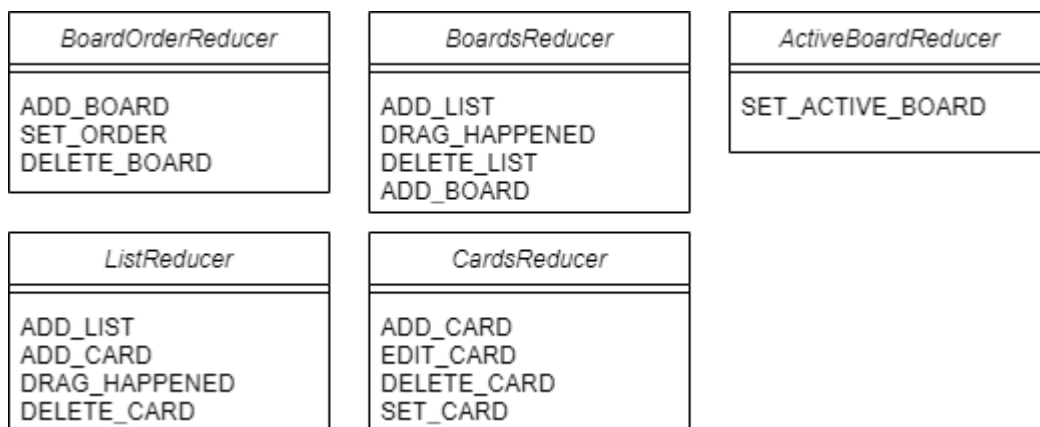


Figura 3.5 – Reducers e suas respectivas actionsos reducers

- **ActiveBoardReducer**  
Responsável pela *action* SET\_ACTIVE\_BOARD, que indica qual *board* está ativo no momento para que as listas desse *board* sejam apresentadas, por isso cada lista no *frontend* possui o *BoardId* para que seja possível a renderização das mesmas corretamente.
- **BoardOrderReducer**  
Responsável pelas ações correspondentes a ordem do *board*, portanto possui as actions ADD\_BOARD em que o *id* do novo *board* é adicionada ao final da lista de ordem. O SET\_ORDER que serve justamente para indicar a ordem dos *boards*, é utilizada no momento de busca na base de dados para que se mantenha a ordem salva pelo usuário e a *action* DELETE\_BOARD, em que o *id* de um *board* é deletado da lista de ordenação.
- **BoardsReducer**  
Responsável pelas ações dos *boards*, possui as *actions* ADD\_LIST em que uma nova lista é adicionada no *board*, o DRAG\_HAPPENED, responsável pelas ações de reordenação das listas feitas pelo usuário, o DELETE\_LIST que



remove pelo *id* a lista deletada, o ADD\_BOARD em que um novo *board* é adicionado no modelo de *boards* com seu respectivo objeto e o DELETE\_BOARD em que o dicionário cuja chave é o *id* do *board* selecionado é removido do objeto de *boards*.

- CardsReducer

Responsável pelas ações dos *cards*, possui as *actions* ADD\_CARD, EDIT\_CARD em que o texto do *card* pode ser alterado, DELETE\_CARD em que o dicionário cuja chave é o *id* do *card* selecionado é removido do objeto *cards*, e SET\_CARD que é utilizado para carregar os dados da base no estado correspondente.

- ListsReducer

Responsável pelas ações das listas, possui a ADD\_LIST, ADD\_CARD em que o *id* do *card* adicionado é incluído na lista de *cards*, o DRAG\_HAPPENED que contém a lógica para ordenação de listas e de *cards*, dentro dessa lógica há a verificação pelo tipo se é uma lista e caso seja o estado é retornado, caso não há a verificação se o *card* pertence a mesma lista e a ordem nova é retornada e caso sejam listas diferentes. É necessário alterar o *id* da lista dentro do *card*, adicionar o *id* do *card* na lista destino e remover da lista origem. Possui a *action* DELETE\_CARD, EDIT\_LIST\_TITLE e DELETE\_LIST.

A partir desses *reducers*, foi criada uma combinação dos mesmos para criar uma Store que contém os estados de todos esses componentes.

### 3.2.4 - Componentes

Foram criados três componentes principais para a visualização dos boards, o componente *Dashboard*, *Board*, *Card* e *List* que estão esquematizados na Figura 3.6. Como todos os componentes estão dentro da estrutura de *Provider* que possui a store criada para o *Kanban*, todos os componentes renderizados dentro dessa estrutura podem obter os estados utilizando o *connect*, tirando a necessidade de passar de um componente a outro essas informações.

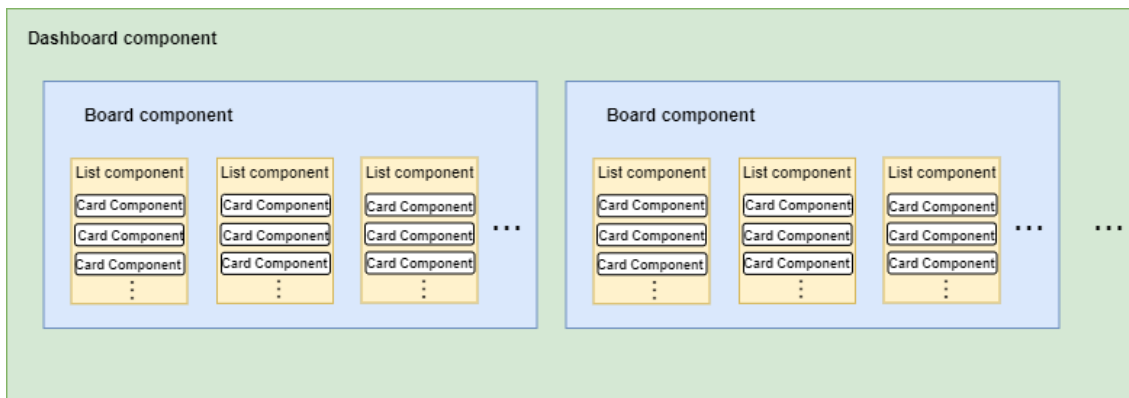


Figura 3.6 - Diagrama de componentes

- Dashboard

Dentro do componente *boards*, os *boards* são renderizados dentro do componente *GridItem* em um *GridContainer* existente na biblioteca Material-UI que permite uma melhor formatação do *layout*, com espaçamentos e tamanhos definidos para cada componente. Um outro *GridItem* foi adicionado com um *TextInput* para que o usuário possa digitar o nome e adicionar um novo *board* com esse nome. Ao clicar no botão de adicionar novo *board*, a *action* `ADD_BOARD` é acionada e os *reducers* atualizam o novo estado na *store*. O nome do *board* está contido dentro da estrutura de um componente *Link*, com esse componente quando há um *click* no nome do *Board*, há um roteamento para a rota `/boardId` em que o *board* será renderizado.

- Board

Assim que criado o componente de *Board*, a *action* `setActiveBoard` é acionada passando como parâmetro o *id* do *board* para que o estado seja atualizado. O *boardId* é obtido via parâmetro da rota e a partir dele o *board* é buscado no estado de *boards* e as listas são renderizadas na ordem que estão listadas na lista dentro do *board* utilizando a função *map*, dentro desse *map* o componente *List* é identificado, e os parâmetros de *id* da lista, título e lista de *cards* são passados via *props* para o componente. Ao final do mapeamento das listas existentes, caso haja alguma, é renderizado um formulário com *input* de nome da lista para que uma lista seja adicionada.

- List

O componente possui em sua raiz o componente *Draggable* da biblioteca *react-beautiful-dnd* e é o responsável pela função de arrastar e soltar que a lista possui,

também possui um botão de edição e remoção em seu cabeçalho para editar o nome ou remover a lista. No corpo da lista os *cards* são renderizados utilizando a função *map*. Na lista também é *renderizado* um formulário após os cards existentes para que um novo seja adicionado.

- Card

O *card* também possui em sua raiz o componente *Draggable*, visto que pode ser arrastado pelas listas dentro do *board*. Dentro dele o texto é mostrado e um botão de delete e edição aparecem no lado direito para que as informações do *card* sejam atualizadas ou o *card* deletado.

### 3.3 – Burndown

#### 3.3.1 – Modelo de Dados

Para o *Burndown* foi criada uma tabela no *DynamoDB* chamada *PlanejaAqui-Burndown*, o objetivo da tabela é armazenar um histórico com todos os dados de atualização das *tasks* de um board para que seja possível criar o *Burndown* de tarefas. A tabela, portanto, possui como *PartitionKey* que é utilizada para a busca dos dados, o *BoardId*, visto que o mesmo é único independente do usuário. O modelo definido contém o Id do *board* e uma lista de informações dos cards, nas informações do cards existem os atributos de *StartDate*, *EndDate*, *EstimateTime*, *RemaningTime* já mencionados na seção 3.2.1 e ilustrados na Figura 3.3, além desses atributos existe uma lista de atualizações que contém os atributos de *UpdatedDate* e *RemaingTime*, o objetivo é, com essas informações, plotar um gráfico *burndown*, mostrando o progresso das tarefas ao longo do período escolhido pelo usuário. A Figura 3.7 ilustra o modelo mencionado.

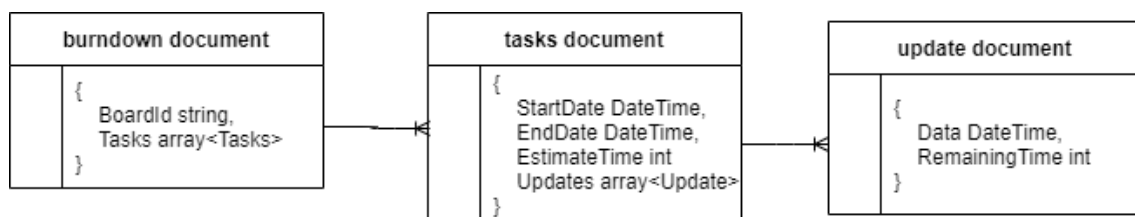


Figura 3.7 – Modelo de Dados Burndown backend

Já para o modelo recebido pelo *frontend*, há uma transformação dos dados da base para que ele tenha apenas a responsabilidade de mostrar os dados. Essa é uma decisão de projeto vista em todos os itens da aplicação, a responsabilidade de processamento e transformação dos dados sempre é feita pelo *backend*. Portanto o modelo visto pelo *frontend* contém os atributos *BoardId*, *StartDate* e *EndDate* e uma lista de objetos do tipo *WorkProgress* que possui os atributos *Date* e *RemainingTime*, que são os únicos atributos necessários para a construção do gráfico. O esquema do modelo pode ser visto na Figura 3.8.

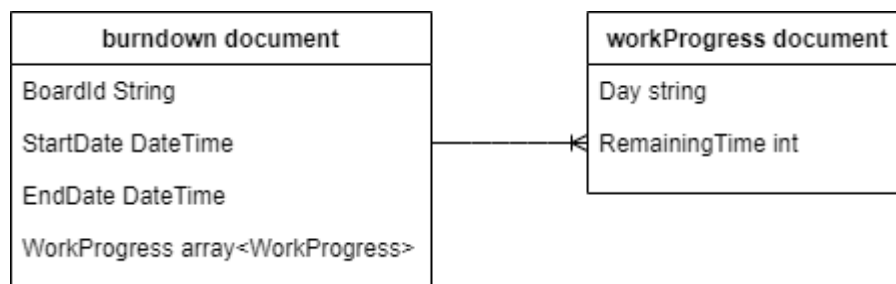


Figura 3.8 – Modelo de dados Burndown frontend

### 3.3.2 – Implementação da atualização e busca

Para a atualização dos dados na tabela do *DynamoDB*, uma *lambda trigger* foi criada na tabela PlanejaAqui-Boards em que cada inserção, modificação e deleção de dados é tratada pela *trigger* e enviada para a tabela PlanejaAqui-Burndown, para que os dados históricos de atualizações sejam salvos em tempo real, fazendo com que o *burndown* esteja sempre atualizado com as informações incluídas pelo usuário, melhorando sua experiência. A *lambda* foi codificada em *.NET Core 3.1* e possui uma lógica simples, a cada evento acionado pela tabela, faz uma transformação de dados coerente com o modelo apresentado na seção 3.3.1 e salva os dados na tabela de *Burndown*, o esquema do processo pode ser visto na Figura 3.9.

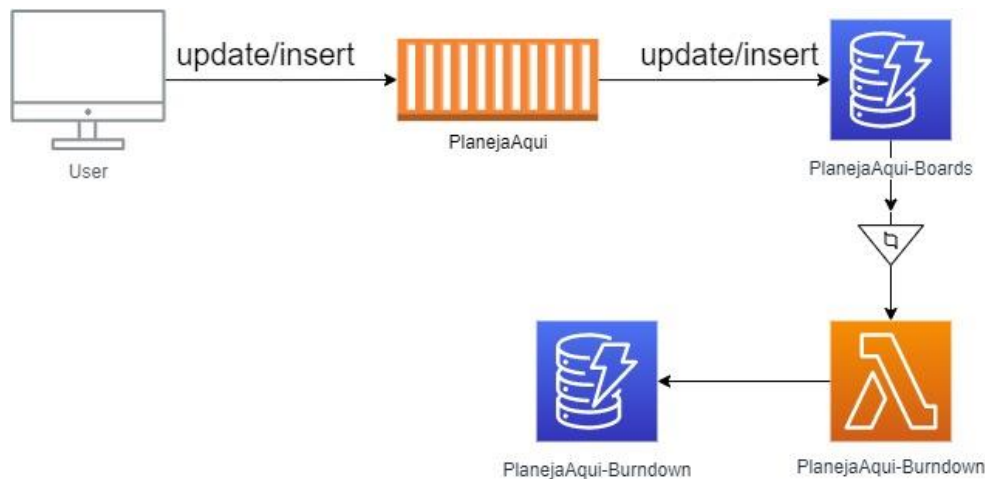


Figura 3.9 – Esquema da atualização de dados do Burndown

Já a busca dos dados é feita em tempo de execução, há um botão na tela de estatísticas que será apresentada no Capítulo 5 e uma entrada de data com data início e fim de amostragem dos dados, a partir de um click no botão “Buscar” uma requisição HTTP é feita ao *backend* para a busca desses dados.

A requisição é um GET no *path* `planejaaqui/boards/{boardId}/burndown` contendo duas *querystring*, uma de *startDate* e *endDate*. A partir desses dados o *Backend* busca o dado do board a partir do *boardId* na tabela de *PlanejaAqui-Burndown*. Com os dados em memória, a aplicação filtra na lista de *Tasks* as tarefas com *StartDate* maior que o *startDate* da *querystring* informada e menores que o *endDate* da *querystring*. Após o filtro há uma transformação de dados para que o *frontend* apenas receba os dados importantes, como mencionado na seção 3.3.1.

É feito um somatório dos *RemainingTimes* de todas as tarefas contidas no board por data e retornadas essas informações por uma lista, a partir disso os dados são plotados no gráfico, em que no eixo x encontram-se as datas do período informado por dia e no eixo y as horas restantes para a conclusão de todas as tarefas do período. O esquema de busca pode ser visto na Figura 3.10.

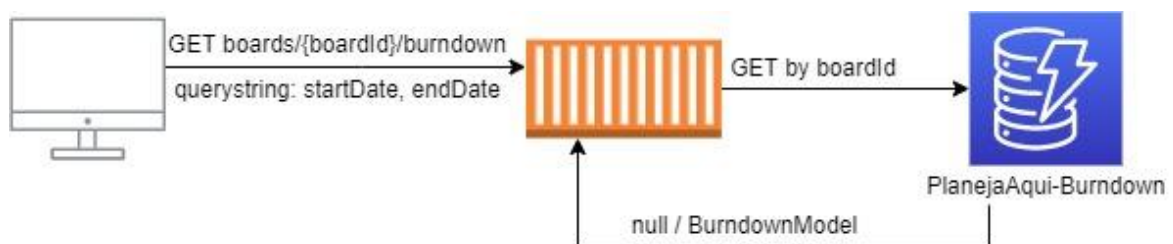


Figura 3.10 – Esquema de busca de dados do Burndown

### 3.4 – Deploy em Homologação

#### 3.4.1 – Backend

- Rest API

Esta aplicação foi para produção como um container *Docker*, para isso o arquivo *Dockerfile* contendo as instruções para criação da imagem foi incluída no projeto.

```
FROM mcr.microsoft.com/dotnet/core/runtime:3.1
WORKDIR /src
COPY . ./
RUN chmod +x #{PlanejaAqui.Worker}#

ENV DOTNET_SYSTEM_GLOBALIZATION_INVARIANT 1
ENTRYPOINT ["./#{PlanejaAqui.Worker}#"]
```

Figura 3.11 – Dockerfile

Para criar a imagem basta rodar o comando ‘docker build -t nomeDaImagem’ que a mesma será criada. Visando o armazenamento das imagens da aplicação foi utilizado o *Amazon Elastic Container Registry*.

Foi criado um repositório para publicar a imagem da aplicação e o *upload* foi feito através de linha de comando. O processo para fazer o *deploy* pode ser realizado através do console da AWS que é um ambiente gráfico, porém o objetivo de se utilizar a linha de comando para esse procedimento é a possibilidade de automatizar esse processo futuramente e essas ações acionadas a partir de um *push* de código por exemplo.

Seguem as instruções feitas para realizar o *upload*:

- Criação do repositório:

```
$ aws ecr create-repository --repository-name
planejaaquiworker
```

- Inclusão de política para remoção de imagens antigas com o objetivo de diminuir os custos por armazenamento:

```
$ aws ecr put-lifecycle-policy --registry-id 725353802875--
repository-name planejaaquiworker --lifecycle-policy-text
'{"rules":[{"rulePriority":10,"description":"Expire old
images","selection":{"tagStatus":"any","countType":"imageCoun
tMoreThan","countNumber":10},"action":{"type":"expire"}}}]'
```

- Realizando o login da configuração *Docker* para que possa executar os comandos de *pull* e *push*:

```
$ (aws ecr get-login --registry-ids 725353802875--no-include-email)
```

- Comando para fazer o *upload* da imagem:

```
$ docker push 725353802875.dkr.us-west-2.amazonaws.com/planejaaquiworker
```

O resultado da criação da imagem e publicação no ECR pode ser visto na Figura 3.12.

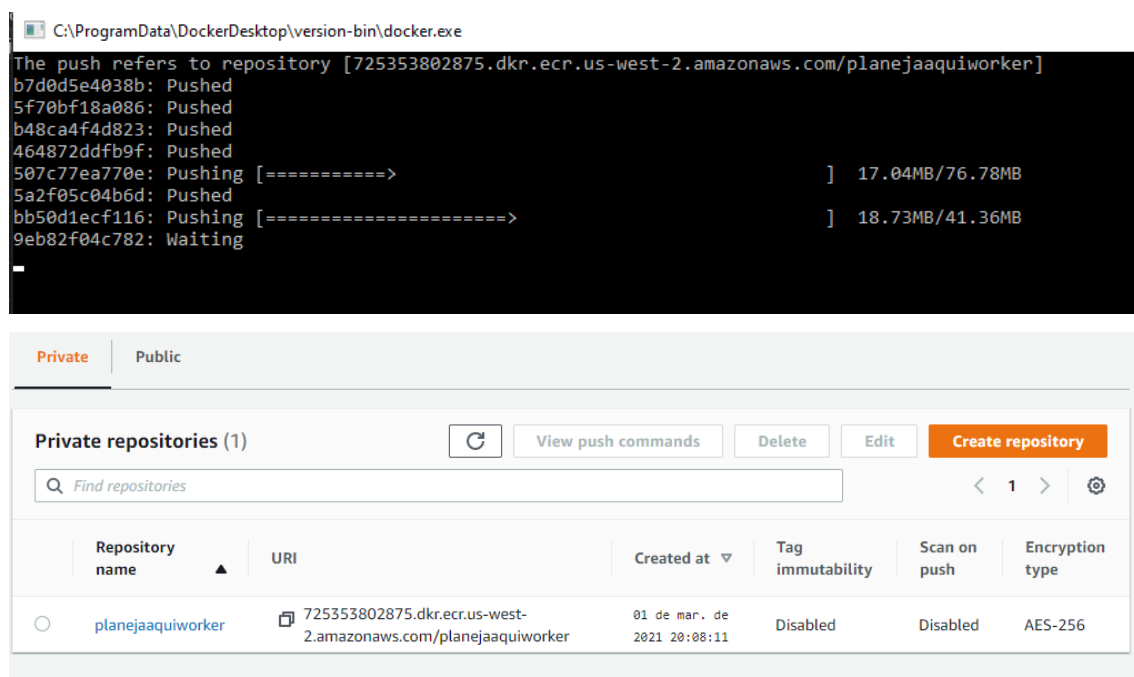


Figura 3.12 – Publicando imagem docker no repositório ECR

Após esse processo, a imagem da aplicação está armazenada no ECR e pode ser utilizada no orquestrador escolhido para gerenciar o *container*. No caso da aplicação PlanejaAqui, o orquestrador escolhido foi o *AWS Elastic Container Service*.

Para configurar o ECS basta entrar no console da AWS e buscar o serviço ECS e iniciar a configuração. O primeiro passo sinalizado é a escolha da aplicação, a AWS fornece uma aplicação de exemplo, porém como já existe uma imagem publicada no ECR, não é necessário usar uma aplicação de exemplo. O segundo passo consiste em definir a tarefa, em que é possível especificar qual imagem *Docker* será usada para os *containers*, quantos *containers* serão utilizados e a alocação de recursos para cada um,

no caso deste trabalho, foram utilizadas as configurações mínimas de recurso por questões de custo.

O próximo passo é definir um serviço, ele mantém cópias da definição de tarefa no cluster, portanto quando o aplicativo é executado como um serviço, qualquer tarefa interrompida é recuperada. No caso deste trabalho, foi configurado para o serviço o número de tarefas desejado igual a 1 para diminuir custos.

O próximo passo é a definição do *load balancing*, é um passo opcional, porém como nossa aplicação será consumida por um *frontend* apartado em questões de infraestrutura, criar um *Load Balancing* é fundamental para o apontamento do *endpoint* do *frontend* para o *backend*.

O último passo é a configuração do *cluster* em que são definidos o nome do *cluster*, o tipo de instância, número de instâncias, para o caso desta aplicação, as configurações mínimas foram escolhidas como pode ser visto na Figura 3.13.

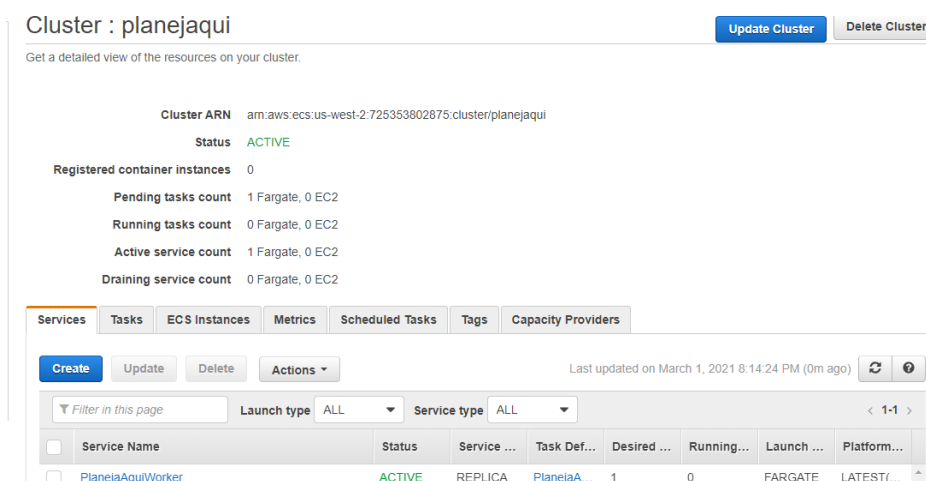


Figura 3.13 – Configurações do Cluster

Após essas etapas o ECS está pronto e a aplicação já pode ser iniciada. O DNS visto na Figura 3.14 do ELB é utilizado como *endpoint* para que o *frontend* possa se comunicar sem precisar estar na mesma camada de infraestrutura, apenas utilizando comunicação HTTP.



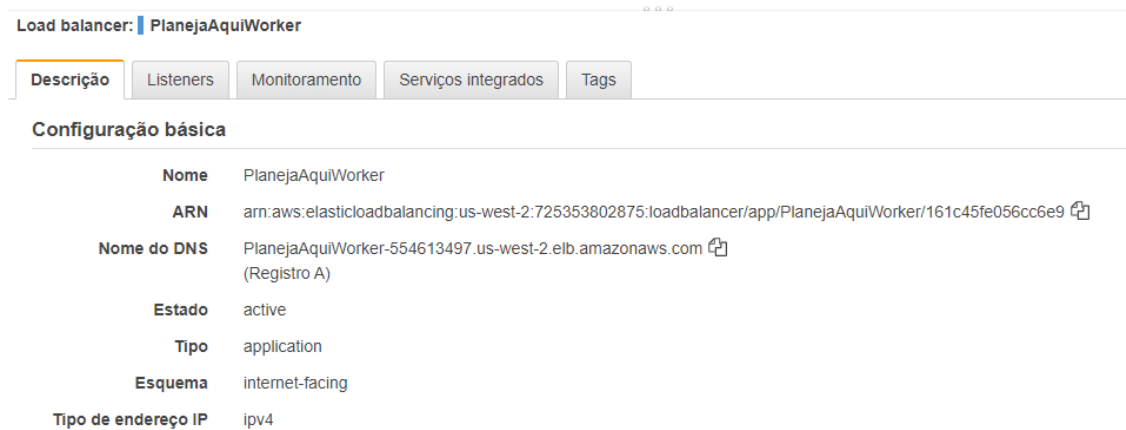


Figura 3.14 – Load Balancer da aplicação

Para verificar se a aplicação está funcionando corretamente em ambiente produtivo, bastou realizar uma requisição HTTP através do software gratuito *Postman* utilizando o LB criado que pode ser visto na Figura 3.15.

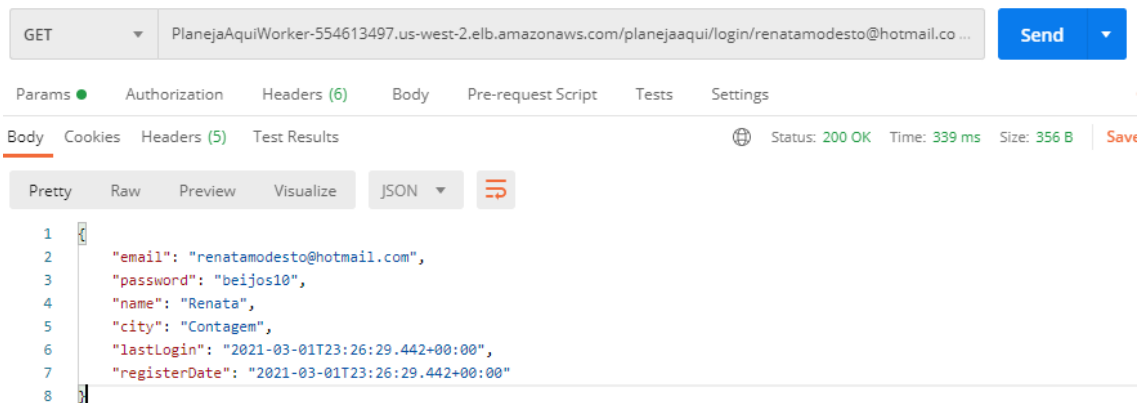


Figura 3.15 – Requisição HTTP no micro serviço

- Lambda

Para fazer a implantação da *Lambda*, foi utilizada a ferramenta *AWS Toolkit for Visual Studio*, é uma ferramenta fácil e bastante útil para conectar e realizar ações com serviços sem precisar abrir a *Console* da AWS. Basta executar a linha de comando na pasta em que está o projeto da Lambda em *.NET Core* e executar o seguinte comando:

```
dotnet lambda deploy-function
```

Após executar o comando, digita-se o nome da *lambda* “PlanejaAqui-Kanban” e a região da AWS, “us-east-1”, na primeira execução, a *lambda* é criada na AWS e execuções subsequentes, é feita a atualização do código.

Após a criação da *lambda*, foi criada a *trigger* na tabela PlanejaAqui-Boards, associando a mesma a *lambda* PlanejaAqui-Kanban. Após esse processo, todo dado salvo na tabela aciona a *lambda* e a mesma realiza a lógica implementada.

### 3.4.2 – Frontend

Como o *frontend* é uma página web estática que realiza requisições HTTP ao *backend*, sua infraestrutura é simples. Através da console da AWS foi criado um *Bucket* S3 “PlanejaAqui-Web”, nas configurações, o bloqueio de acesso público foi desativado para que a aplicação possa ser acessada de qualquer lugar.

Após a criação do mesmo, em propriedades há a opção de criar um website estático na opção “*Static Website Hosting*” como pode ser visto na Figura 3.14. Após as configurações feitas, o comando “*npm run build*” foi executado na pasta raiz da aplicação para que os arquivos estáticos fossem gerados. No *bucket*, foi feito o upload dos arquivos gerados no passo anterior, e assim o site estático está disponível para acesso.



Figura 3.16 – Configurações do Bucket

# Capítulo 4

## Apresentação do Sistema

Nesta seção serão apresentadas em forma de imagens todas as funcionalidades apresentadas no capítulo anterior, abrangendo as funcionalidades de cadastro e autenticação, criação de *boards*, listas e tarefas com suas respectivas configurações, atualização e remoção de dados do *Kanban* e gráfico de *Burndown*.

### 4.1 – Diagrama de Telas

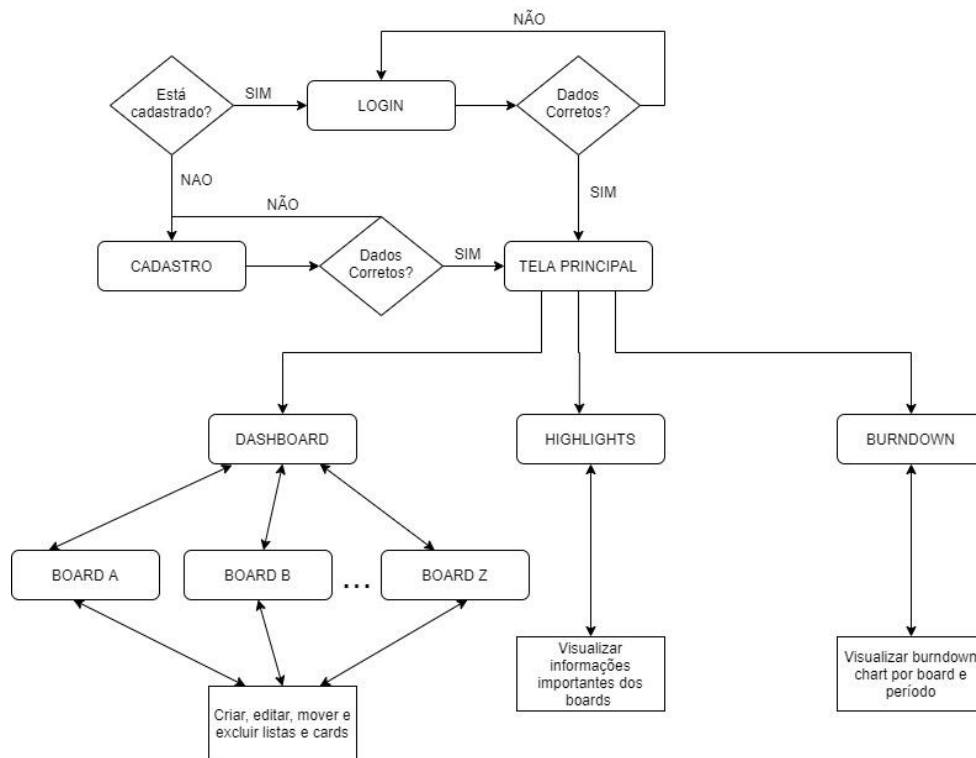


Figura 4.1 – Diagrama de telas

### 4.2 – Cadastro e Autenticação

- Cadastro

Na Figura 4.2 é possível visualizar a tela de cadastro com seus possíveis campos e o botão de “Get Started” responsável pela confirmação dos dados e efetivação do cadastro e o botão de “Login” para navegar a tela de autenticação. Os possíveis erros de cadastro podem ser vistos nas Figuras 4.3 e 4.4. Já a Figura 4.5 mostra a mensagem exibida no cadastro feito com sucesso.

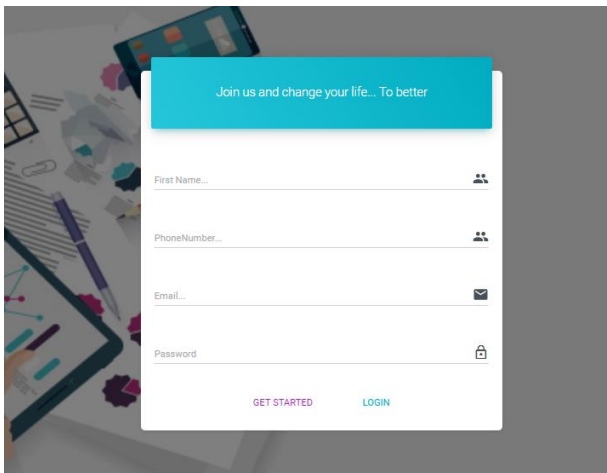


Figura 4.2 – Tela de cadastro

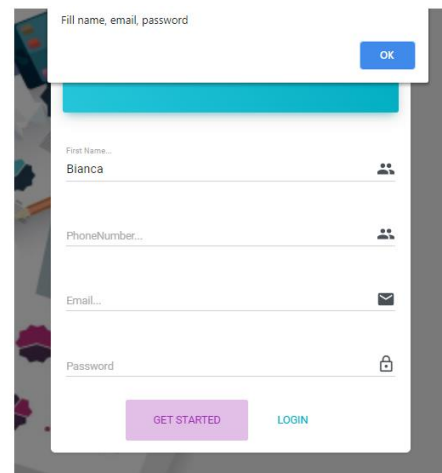


Figura 4.3 – Dados insuficientes

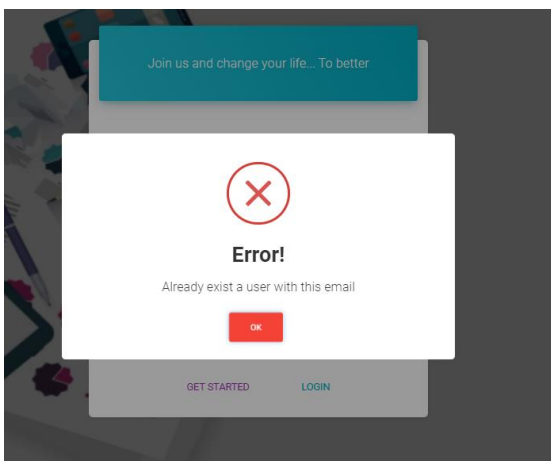


Figura 4.4 – Usuário com mesmo e-mail

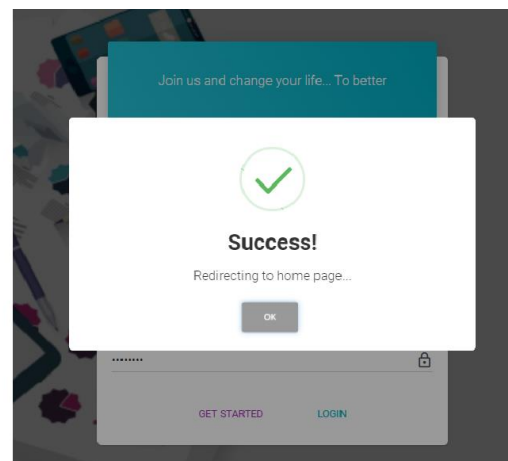


Figura 4.5 – Cadastro com sucesso

- **Autenticação**

A tela de autenticação pode ser vista na Figura 4.6 em que é possível entrar com as credenciais previamente cadastradas na tela vista na Figura 4.1. É possível alternar entre as duas telas utilizando os botões “Login” e “First Time?”. A Figura 4.7 mostra o erro retornado ao usuário caso a autenticação falhe. Apesar do *backend* conseguir distinguir entre um usuário inexistente e senha incorreta, a mesma mensagem de erro é exibida para prevenir possíveis tentativas de fraude.

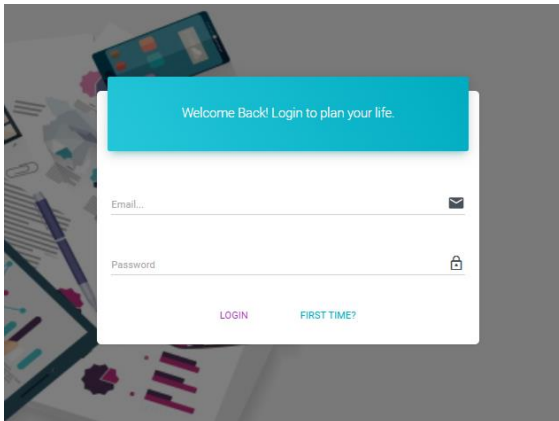


Figura 4.6 – Tela de Autenticação

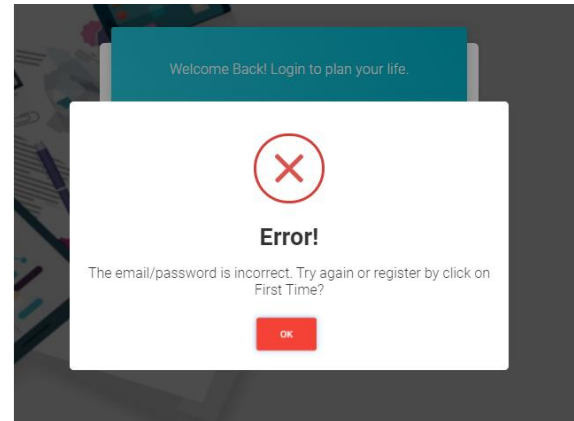


Figura 4.7 – Erro na autenticação

### 4.3– Tela Principal

A Figura 4.8 mostra a tela principal, que é apresentada após o sucesso no cadastro (Figura 4.5) e autenticação com sucesso. As descrições de cada funcionalidade na tela são realizadas a seguir, e referenciadas através de numeração arábica:

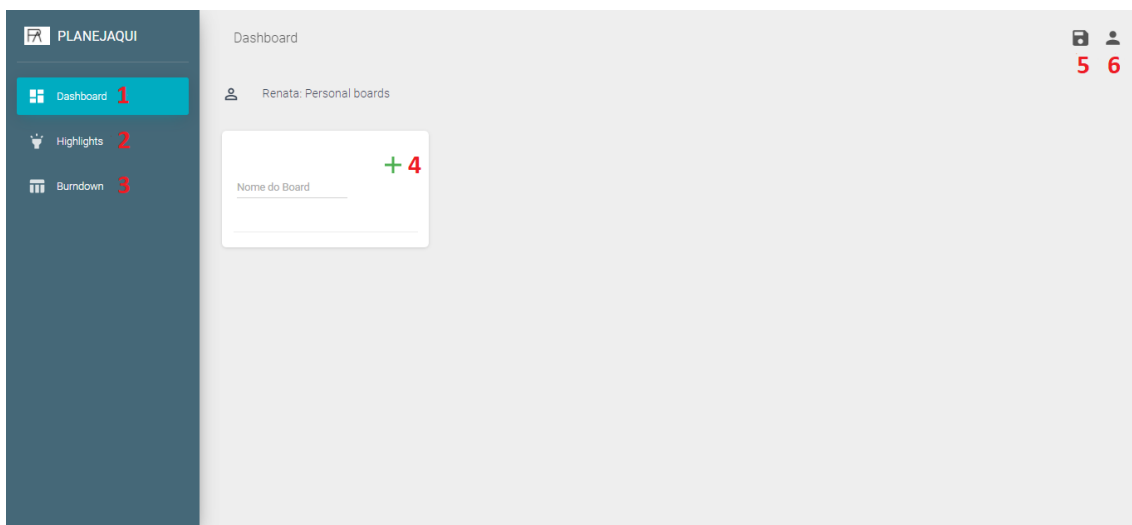


Figura 4.8 – Tela Principal

- 1- *Dashboard*: Contém todos os boards do usuário, inicialmente o mesmo encontra-se com apenas com o componente necessário para adição de um novo Board que será explicado mais detalhadamente na seção 4.3.
- 2- *Highlight*: Contém um resumo dos dados dos boards, como tarefas em aberto, horas que faltam a completar, será melhor detalhado na seção 4.4.
- 3- *Kanban*: Possui o gráfico *Kanban* dos *boards*, é possível filtrar por *Board* e faixa de data, será melhor detalhado na seção 4.5.
- 4- Botão para adicionar um novo *board*.

- 5- Botão para salvar as modificações realizadas pelo usuário.
- 6- Botão para realizar funcionalidades do usuário, no caso foi implementada a função de *logout*.

#### 4.4– Dashboard e Boards

Ao realizar um cadastro não existem *boards* a serem exibidos, apenas uma caixa com entrada para o Nome do *Board* e um botão para adição do mesmo. A Figura 4.9 mostra o possível erro no momento de adição do *board*, que acontece caso o usuário não escolha um nome para o *board*.

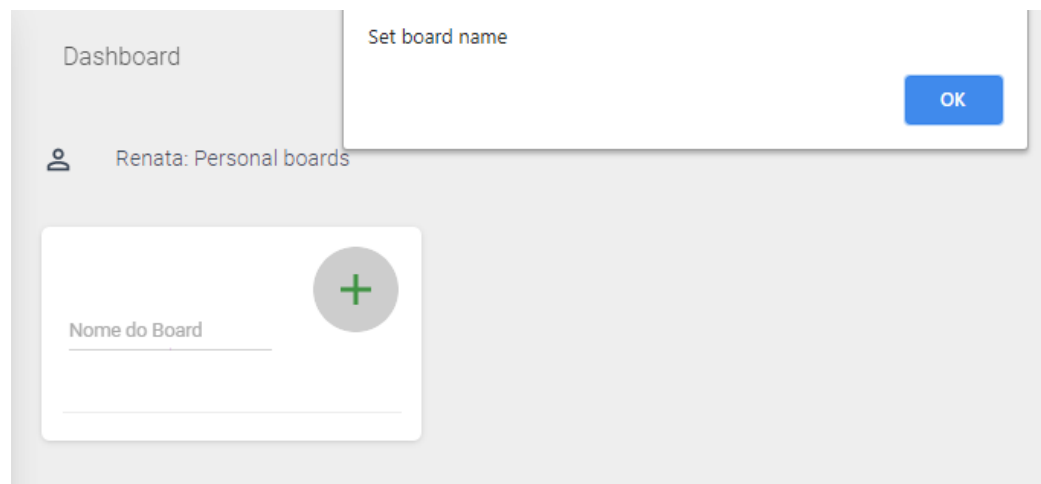


Figura 4.9 – Erro ao adicionar Board

Na Figura 4.10 é possível observar um *Dashboard* mais completo com os *Boards* criados e divididos por projetos, o botão em formato de lixeira serve para deletar o *board* em questão e ao final da listagem de *boards*, a caixa para adicionar um novo *board* é exibida.

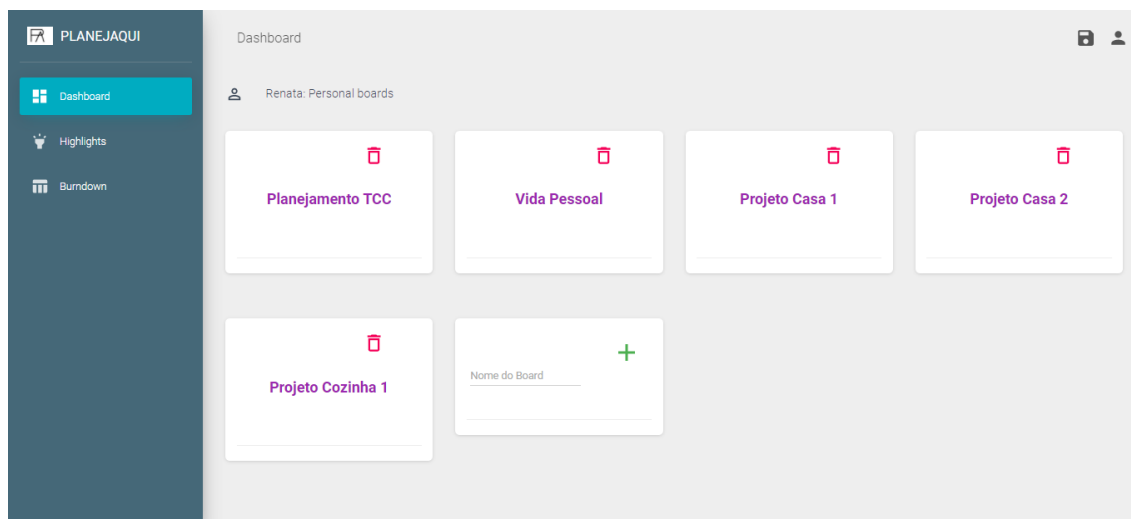


Figura 4.10 – Boards criados

Ao clicar sobre um *board* o usuário é redirecionado para a tela do *Board*, caso o mesmo ainda não esteja preenchido, a tela mostrada na Figura 4.11 é exibida, em que o primeiro passo para iniciar o preenchimento do board é adicionar uma nova lista no botão “Add another list”, quando o usuário clica sob o botão uma entrada de texto é exibida e um botão para adicionar a lista, como pode ser visto na Figura 4.12.

Após a adição, uma entrada de texto para adicionar um *card* é exibida no início da lista assim como um botão em formato de lixeira para remover a lista. A caixa de texto para adicionar uma nova lista aparece ao final das listas, como pode ser visto na Figura 4.13.



Figura 4.11 – Board selecionado

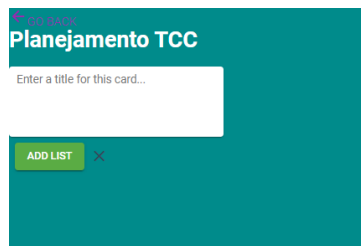


Figura 4.12 – Adição de lista

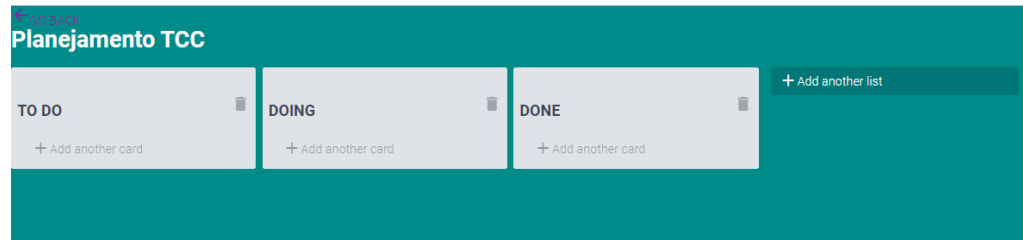


Figura 4.13 – Listas adicionadas

Para criar um *card* basta clicar em “Add another card” e digitar o texto do *card*. Ao ser criado, dois botões surgem ao lado direito do *card*, um para edição e outro para deleção do mesmo. Ao clicar duas vezes sob o texto do *card* é possível editar o mesmo, e clicando sob o botão de edição é possível incluir as informações de início e término da tarefa, horas estimadas e horas faltantes. A funcionalidade de adição e edição de *card* pode ser vista na Figura 4.14.

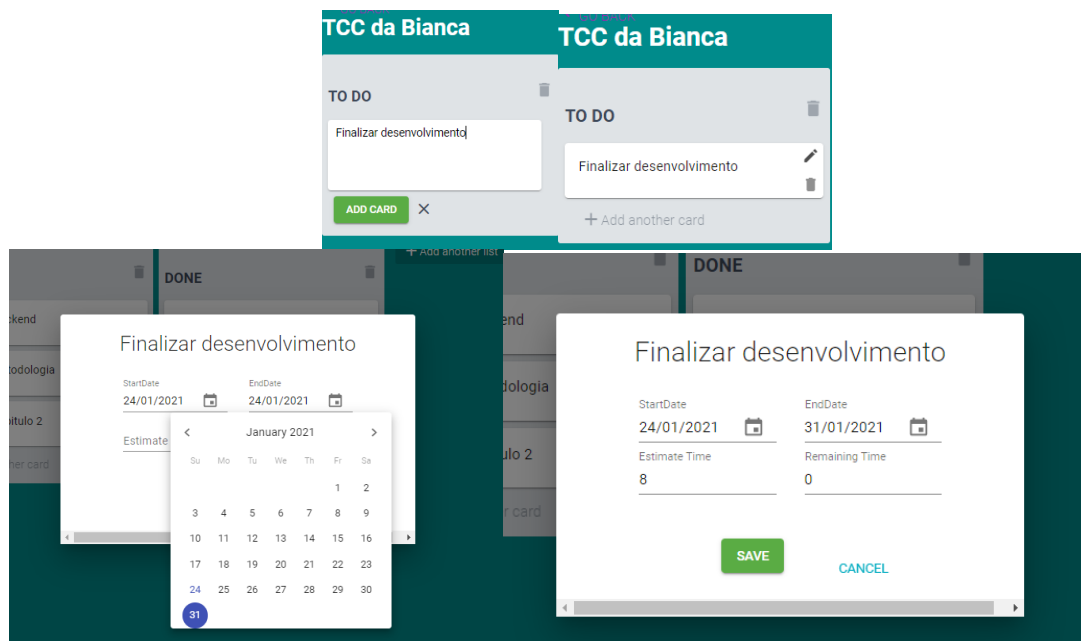


Figura 4.14 – Adição e edição de Card



## 4.5– Highlights

A tela de *Highlights* possui três entradas de usuário, o nome do *Board* ao qual quer visualizar os dados e a data de início e de fim. A partir disso é exibida na tela as tarefas em aberto do *board* em formato de lista, indicação de número de tarefas atrasadas para o *board* selecionado no período selecionado e as horas que já foram concluídas. A Figura 4.15 exibe a tela completa e a Figura 4.16 os detalhes da tela de forma ampliada.

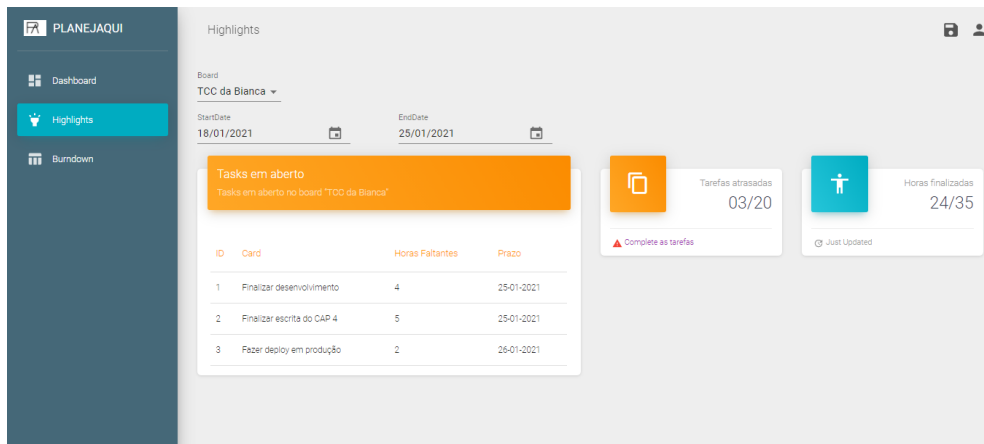


Figura 4.15 – Página de Highlights

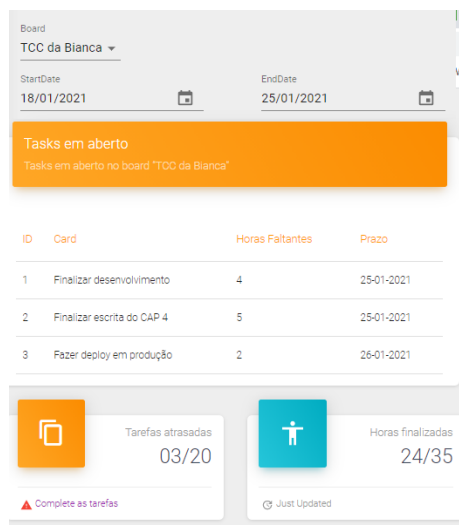


Figura 4.16 – Detalhes da página de highlights

## 4.6- Burndown

Na tela de *Burndown*, as mesmos três entradas da seção 4.5 estão disponíveis para o usuário, a partir dessas entradas o gráfico de *Burndown* é desenhado para se adequar aos filtros selecionados. Portanto é possível obter o mesmo indicando o *Board*

e a faixa de datas a ser visualizado, um exemplo de gráfico pode ser visto na Figura 4.17 e os detalhes das entradas na Figura 4.18.

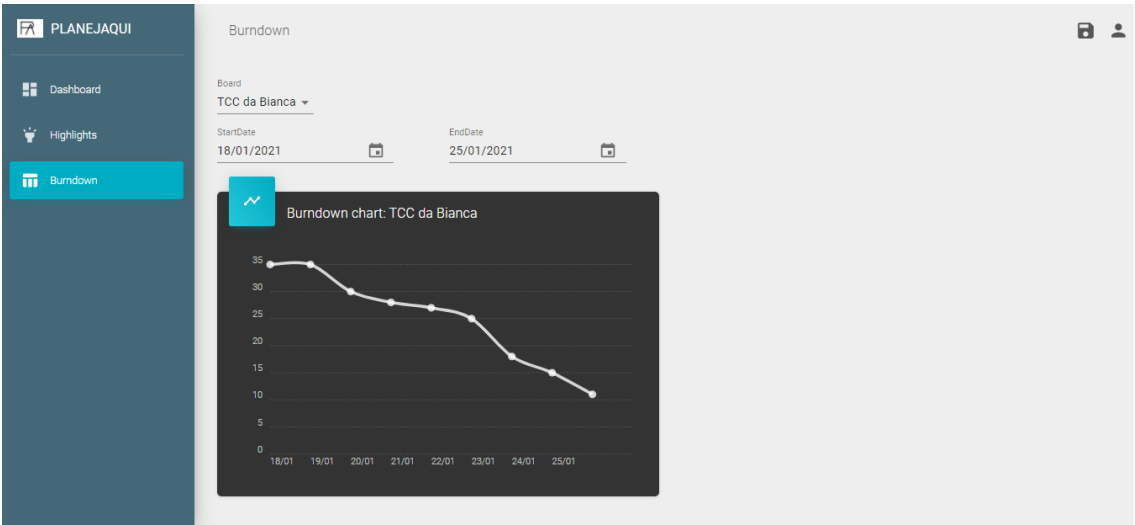


Figura 4.17 – Burndown chart

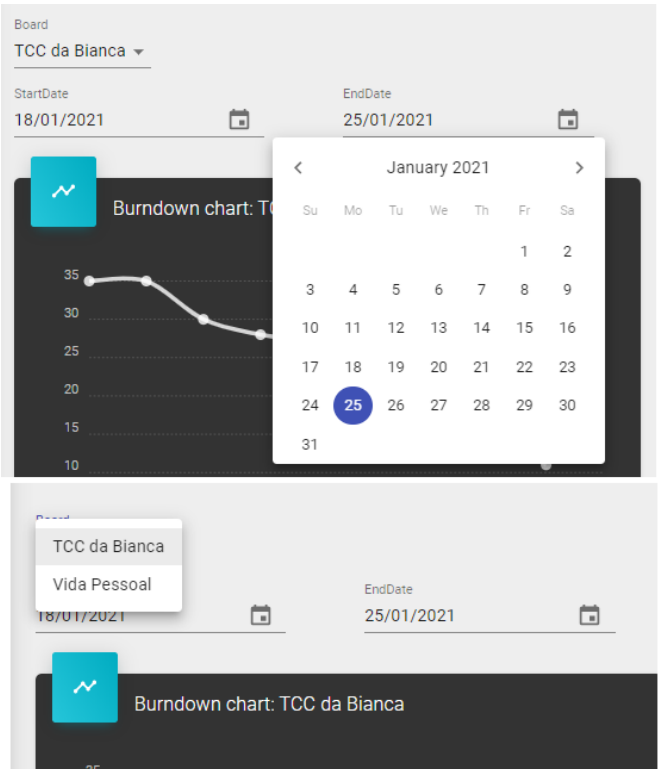


Figura 4.18 – Entradas de dados Burndown

# Capítulo 5

## Conclusão

A presente monografia buscou implementar um sistema intuitivo, prático para organização de tarefas diárias e/ou projetos com único usuário, além da disponibilização do mesmo na nuvem da AWS.

Os objetivos foram atingidos com sucesso e a utilização da biblioteca *ReactJS* e *MaterialUI* para desenvolvimento da interface possibilitaram uma UI amigável e simples para o usuário. A separação das aplicações por funcionalidade também teve êxito, tornando o ecossistema da aplicação limpo e fácil de realizar manutenções.

A decisão de utilizar a AWS facilitou a implantação dos sistemas e sua disponibilização na internet, visto que possui uma gama de serviços disponíveis para a implementação das funcionalidades do sistema.

Alguns pontos podem ser aprimorados para o futuro, como validação do e-mail no cadastro, para que uma mensagem de confirmação seja enviada ao e-mail informado e possibilite a confirmação de cadastro e também a recuperação de senha. Outro ponto seria a flexibilização da tela de *Highlights* para que o usuário possa editar, mover ou personalizar os avisos mostrados na tela. Algo interessante também seria a criação de times para que os usuários possam interagir nos *Boards*.

# Bibliografia

- [1] BECK, Kent. BEEDLE, Mike. BENNEKUM, Arie van. COCKBURN, Alistair. CUNNINGHAM, Ward. FOWLER, Martin. GRENNING, James. HIGHSMITH, Jim. HUNT, Andrew. JEFFRIES, Ron. KERN, Jon. MARICK, Brian. MARTIN, Robert C. MELLOR, Steve. SCHWABER, Ken. SUTHERLAND, Jeff. THOMAS, Dave. Manifesto para Desenvolvimento Ágil de Software. Disponível em: <https://agilemanifesto.org/iso/ptbr/principles.html>. Acesso em: 7 out. 2020.
- [2] ESPINHA, Roberto Gil. Kanban: O que é e TUDO sobre como gerenciar fluxos de trabalho. [S. l.], p. 1-1, 23 jan. 2019. Disponível em: <https://artia.com/kanban/>. Acesso em: 7 out. 2020.
- [3] VEYRAT, Pierre. Não confunda mais: Agile, Scrum e Kanban. [s. l.], 15 fev. 2017. Disponível em: <https://www.heflo.com/pt-br/agil/agile-scrum-e-kanban/>. Acesso em: 7 out. 2020.
- [4] ENTENDA o que são containers e quais são suas vantagens. [S. l.], p. 1-1, 19 fev. 2019. Disponível em: <https://www.valuehost.com.br/blog/o-que-sao-containers/>. Acesso em: 19 out. 2020.
- [5] HIRANO, Marcio. Docker e Containers. [S. l.], p. 1-1, 7 jul. 2019. Disponível em: <https://medium.com/tecnologia-e-afins/o-que-%C3%A9-docker-188e283088dd>. Acesso em: 19 out. 2020.
- [6] Introdução ao .NET. Disponível em: <https://docs.microsoft.com/pt-br/dotnet/core/introduction>. Acesso em: 19 out. 2020.
- [7] LONGEN, Andrei. O Que é React e Como Funciona?. [s. l.], 21 maio 2019. Disponível em: <https://www.hostinger.com.br/tutoriais/o-que-e-react-javascript>. Acesso em: 11 out. 2020.
- [8] \_\_\_\_\_”Modelo de objeto de Documento”. Disponível em: [https://developer.mozilla.org/pt-BR/docs/DOM/Referencia\\_do\\_DOM](https://developer.mozilla.org/pt-BR/docs/DOM/Referencia_do_DOM). Acesso em 11/10
- [9] REACT: o que é e como funciona essa ferramenta?. [s. l.], 13 set. 2018. Disponível em: <https://tableless.com.br/react-o-que-e-e-como-funciona-essa-ferramenta/-essa-ferramenta/>. Acesso em: 11 out. 2020.

- [10] HANASHIRO, Akira. Flux – descubra o motivo do sucesso dessa arquitetura em grandes empresas [S. l.], p. 1-1, 21 maio 2019. Disponível em: <https://www.treinaweb.com.br/blog/flux-descubra-o-motivo-do-sucesso-dessa-arquitetura-em-grandes-empresas/>. Acesso em: 13 out. 2020.
- [11] KRÖGER, Hélio. Entendendo React e Redux de uma vez por todas. [S. l.], p. 1-1, 16 set. 2017. Disponível em: [https://medium.com/@hliojnior\\_34681/entenda-react-e-redux-de-uma-vez-por-todas-c761bc3194ca](https://medium.com/@hliojnior_34681/entenda-react-e-redux-de-uma-vez-por-todas-c761bc3194ca). Acesso em: 13 out. 2020.
- [12] Material UI – Biblioteca de components React para um desenvolvimento ágil e fácil. Construa seu próprio design, ou comece com Material Design. Disponível em: <https://material-ui.com/pt/>. Acesso em 13 out. 2020.
- [13] QUEM inventou a computação em nuvem [S. l.], p. 1-1, 14 set. 2020. Disponível em: <https://skyone.solutions/pb/conheca-a-computacao-em-nuvem/>. Acesso em: 13 out. 2020.
- [14] Tipos de Computação em Nuvem. Disponível em: <https://aws.amazon.com/pt/types-of-cloud-computing/>. Acesso em: 13 out. 2020.
- [15] Amazon DynamoDB – Serviço de banco de dados NoSQL rápido e flexível para qualquer escala. Disponível em: <https://aws.amazon.com/pt/dynamodb/> . Acesso em: 13 out. 2020.
- [16] MELLO, Vanessa de Oliveira. Load Balance: o que é e como funciona. [s. l.], p. 1-1, 3 jul. 2019. Disponível em: <https://king.host/blog/2018/07/load-balance/>. Acesso em: 19 out. 2020.
- [17] What is Elastic Load Balancing. Disponível em: <https://docs.aws.amazon.com/elasticloadbalancing/latest/userguide/what-is-load-balancing.html>. Acesso em: 19 out. 2020.
- [18] Perguntas Frequentes sobre o Amazon Elastic Container Service. Disponível em: <https://aws.amazon.com/pt/ecs/faqs/> . Acesso em: 19 out. 2020.
- [19] Amazon S3 – Armazenamento de objetos construído para armazenar e recuperar qualquer volume de dados de qualquer local. Disponível em: <https://aws.amazon.com/pt/s3/> . Acesso em: 23 out. 2020.
- [20] Como configurar um bucket do S3 para hospedagem em site estático? Disponível em: [https://docs.aws.amazon.com/pt\\_br/AmazonS3/latest/user-guide/static-website-hosting.html](https://docs.aws.amazon.com/pt_br/AmazonS3/latest/user-guide/static-website-hosting.html) . Acesso em: 23 out. 2020.