JEVERSON GONÇALVES

DESENVOLVIMENTO DE MICROSERVICES PARA RESOLUÇÃO DE *PUZZLES*: UM ESTUDO SOBRE A ARQUITETURA ORIENTADA A SERVIÇOS

LAVRAS – MG

2023

JEVERSON GONÇALVES

DESENVOLVIMENTO DE MICROSERVICES PARA RESOLUÇÃO DE *PUZZLES*: UM ESTUDO SOBRE A ARQUITETURA ORIENTADA A SERVIÇOS

Monografia apresentada à Universidade Federal de Lavras, como parte das exigências do Curso de Sistemas de Informação, para a obtenção do título de Bacharel.

Prof. Dr. Rafael Serapilha Durelli
Orientador

LAVRAS – MG
2023

JEVERSON GONÇALVES

DESENVOLVIMENTO DE MICROSERVICES PARA RESOLUÇÃO DE *PUZZLES*: UM ESTUDO SOBRE A ARQUITETURA ORIENTADA A SERVIÇOS

Monografia apresentada à Universidade Federal de Lavras, como parte das exigências do Curso de Sistemas de Informação, para a obtenção do título de Bacharel.

APROVADA em 11 de Julho de 2023.

Prof. Dr. Rafael Serapilha Durelli UFLA
Prof. Dr. Paulo Afonso Parreira Junior UFLA
Prof. Dr. Mauricio Ronny de Almeida Souza UFLA

Prof. Dr. Rafael Serapilha Durelli
Orientador

LAVRAS – MG
2023

Dedico esse trabalho à toda minha família, em especial a minha mãe Maria Rosenei Ferreira Gonçalves, meu pai João Batista Gonçalves, meu irmão Janderson Gonçalves, minha namorada Julia Natalia Souza e todos meus amigos.

AGRADECIMENTOS

Agradeço meus pais por todo apoio e qualidade de vida que proporcionou para mim e meu irmão.
Agradeço à minha namorada pelos ensinamentos, apoio e compreensão.
Agradeço aos meus familiares e amigos, por palavras de afeto e apoio.
Agradeço ao Prof. Dr. Rafael Serapilha Durelli pelos ensinamentos durante o curso, paciência e conhecimento no desenvolvimento desse projeto.
E agradeço à Universidade Federal de Lavras e todos os professores que tive aula durante minha trajetória acadêmica.

RESUMO

O objetivo deste projeto consiste em criar microsserviços, envolvendo as áreas de *Frontend*, *Backend* e *DevOps*, visando resolver desafios de natureza lógica (*puzzles*), tais como *nQueen*, *Sudoku* e *Maze Solver*. O projeto investiga e emprega ferramentas modernas de desenvolvimento, incluindo linguagens de programação, *frameworks*, sistemas de mensageria e técnicas de orquestração de contêineres. A adoção dessas tecnologias oferece vantagens competitivas tanto para as organizações quanto para os profissionais de desenvolvimento, permitindo uma abordagem de desenvolvimento mais ágil, aprimoramento da qualidade do produto final, redução de custos e aperfeiçoamento da experiência do usuário.

Palavras-chave: microsserviços - backend - frontend - contêineres - docker - kubernetes

ABSTRACT

The objective of this project is to create microservices, involving the areas of Frontend, Backend, and DevOps, aiming to solve logical challenges (*puzzles*) such as nQueen, Sudoku, and Maze Solver. The project investigates and employs modern development tools, including programming languages, frameworks, messaging systems, and container orchestration techniques. The adoption of these technologies offers competitive advantages for both organizations and development professionals, enabling a more agile development approach, improved quality of the final product, cost reduction, and enhanced user experience.

Keywords: microservices - backend - frontend - containers - docker - kubernetes

LISTA DE FIGURAS

Figura 2.1 – Comparação entre *C4 Model* e UML 13
Figura 3.1 – Passo a passo seguido 24
Figura 3.2 – Modelagem de contexto do *C4 model* 25
Figura 3.3 – Modelagem do contêiner Puzzle Application 26
Figura 3.4 – Modelagem do contêiner Puzzle Processor 27
Figura 3.5 – Estrutura dos projetos de *backend* 28
Figura 3.6 – Estrutura da pasta src do *backend* ***Processor*** 29
Figura 3.7 – Estrutura do projeto *frontend* 30
Figura 3.8 – Modelagem do RabbitMQ 31
Figura 3.9 – Modelagem do banco de dados 32
Figura 3.10 – Solução válida para um tabuleiro 4x4 33
Figura 3.11 – Solução inválida para um tabuleiro 4x4 34
Figura 3.12 – Exemplo de grade pré-preenchida 34
Figura 3.13 – Exemplo de grade preenchida 35
Figura 3.14 – Exemplo de caminho encontrado no labirinto 36
Figura 4.1 – Tela inicial 39
Figura 4.2 – Adicionando uma *nQueen* 39
Figura 4.3 – Adicionando um *Sudoku* 40
Figura 4.4 – Adicionando um *Maze* 40
Figura 4.5 – Detalhes de um *nQueen* 41
Figura 4.6 – Detalhes de um *Sudoku* 41
Figura 4.7 – Detalhes de um *Maze Solver* 42

SUMÁRIO

1 INTRODUÇÃO 9

1.1 Contextualização e Motivação 9

1.2 Objetivo 9

1.3 Organização 10

2 REFERENCIAL TEÓRICO 11

2.1 Tecnologias utilizadas e técnicas 11

2.1.1 Arquitetura de Microsserviços x Monolítica 11

2.1.2 Por que utilizar microsserviços? 12

2.1.3 *C4 Model* 12

2.1.4 Comparativo entre *C4 Model* e UML 13

2.1.5 Javascript 13

2.1.6 Typescript 14

2.1.7 Node.js 15

2.1.8 Nest.js 16

2.1.9 Docker 17

2.1.10 Kubernetes 18

2.1.11 PostgreSQL 18

2.1.12 RabbitMQ 19

2.1.13 React 20

2.1.14 Next.js 20

2.2 Algoritmos Implementados no projeto 21

2.2.1 Dijkstra 21

2.2.2 Backtracking 22

3 METODOLOGIA DE DESENVOLVIMENTO 23

3.1 Procedimentos 23

3.2 Definição da arquitetura do software 23

3.3 Linguagem de programação e *frameworks* 24

3.4 Organização dos códigos fonte 27

3.5 Fila de processamento 30

3.6 Definição e modelagem do banco de dados 31

3.7 Algoritimos para resolução de *puzzles* 33

3.7.1 ***NQueens*** 33
3.7.2 ***Sudoku*** 34
3.7.3 ***Maze solver*** 35
3.8 **Conteinerização da aplicação** 35
4 **RESULTADOS** 37
4.1 ***Backend*** 37
4.1.1 ***API Application*** 37
4.1.2 **Processor** 38
4.2 **Frontend** 38
4.3 **Infraestrutura** 42
4.3.1 **Replicabilidade** 43
4.4 **Principais desafios enfrentados** 44
5 **CONSIDERAÇÕES FINAIS** 45
5.1 **Disciplinas que contribuíram para o desenvolvimento do trabalho** 45
5.2 **Trabalhos futuros** 46
REFERÊNCIAS 47

1 INTRODUÇÃO

1.1 Contextualização e Motivação

Atualmente, vivemos em um mundo altamente conectado à tecnologia, onde a presença de softwares é essencial em quase todas as áreas da nossa vida. Desde a forma como nos comunicamos e consumimos informações até como realizamos transações comerciais e gerenciamos nossas tarefas diárias, os softwares desempenham um papel fundamental. Segundo a *pesquisa*[1] Uso da TI nas empresas de 2023 da FGVcia, gastos e investimentos em tecnologia continuam crescendo e atingiram cerca de 9% da receita das empresas. Portanto, essa dependência crescente da tecnologia cria uma demanda cada vez maior por sistemas de software eficientes, escaláveis e inovadores.

Nesse contexto, a demanda por softwares de alta qualidade e desempenho é cada vez maior. Empresas buscam se destacar no mercado oferecendo soluções inovadoras, eficientes e escaláveis, capazes de atender às necessidades dos clientes de forma ágil. Nesse sentido, a utilização de ferramentas e técnicas modernas no desenvolvimento de software se torna essencial para acompanhar as demandas do mercado e garantir a competitividade. Profissionais da área de desenvolvimento também se beneficiam ao dominar essas ferramentas, tornando-se mais atrativos para o mercado de trabalho e abrindo portas para oportunidades de crescimento e reconhecimento. De acordo com (PRESSMAN, 2011), hoje a distribuição dos custos para o desenvolvimento de sistemas mudou drasticamente, o software ao contrário do hardware é o item de maior custo. Portanto, a adoção de abordagens modernas no desenvolvimento de software se apresenta como um fator chave para entregar soluções de qualidade, impulsionar o crescimento das empresas e se destacar em um mercado cada vez mais exigente e dinâmico.

1.2 Objetivo

Este trabalho tem como objetivo criar um sistema completo, com *Frontend*, *Backend* e *DevOps*, para a resolução de *puzzles*, tais como *nQueen*, *Sudoku* e *Maze Solver*. O projeto explora e utiliza ferramentas como linguagens de programação e *frameworks*, sistemas de mensageria e técnicas de orquestração de contêineres. A adoção dessas tecnologias representa um diferencial competitivo tanto para as empresas quanto para os desenvolvedores, proporcionando

[1] Disponível em: <https://eaesp.fgv.br/sites/eaesp.fgv.br/files/u68/pesti-fgvcia-2023_0.pdf>. Acesso em 26.Jun 2023

maior agilidade no desenvolvimento, maior qualidade do produto final, redução de custos e uma experiência aprimorada para os usuários.

1.3 Organização

Este trabalho está organizado da seguinte forma. No Capítulo 2 são apresentados todos os conceitos e fundamentações teóricas desse trabalho. No Capítulo 3 é apresentado a metodologia conduzida. No Capítulo 4 é apresentado os resultados obtidos. E por fim, no Capítulo 5 é explicitado as considerações finais com sugestões de trabalhos futuros.

2 REFERENCIAL TEÓRICO

Este capítulo apresenta as tecnologias e conceitos utilizados no desenvolvimento do projeto em questão. Ele está dividido em duas seções principais. A Seção 2.1 descreve as técnicas e tecnologias empregadas na criação do projeto, enquanto a Seção 2.2 aborda os conceitos e algoritmos utilizados.

2.1 Tecnologias utilizadas e técnicas

Nesta seção, são apresentadas todas as tecnologias utilizadas na implementação do projeto. Iniciando na seção 2.1.1 com uma discussão sobre a arquitetura utilizada no projeto e também o modelo utilizado para representá-la, na seção 2.1.3. Em sequência uma descrição sobre JavaScript na Seção 2.1.5. Em seguida, destaca-se a linguagem TypeScript na Seção 2.1.6. A plataforma Node.js é abordada na Seção 2.1.7, enquanto o Nest.js é explicado na Seção 2.1.8. As tecnologias Docker e Kubernetes são apresentadas nas Seções 2.1.9 e 2.1.10, respectivamente. Além disso, na Seção 2.1.11, é introduzido o Kubernetes, um gerenciador de banco de dados. O RabbitMQ é destacado como o *message broker* utilizado neste projeto, mencionado na Seção 2.1.12. Por fim, as bibliotecas/*frameworks* React e Next.js, utilizadas no projeto, são apresentadas nas Seções 2.1.13 e 2.1.14, respectivamente.

2.1.1 Arquitetura de Microsserviços x Monolítica

Um monólito é uma aplicação em que todas as suas funcionalidades estão agrupadas e interligadas em um único sistema. Todas as partes do monólito são desenvolvidas, implantadas e escaladas juntas, compartilhando a mesma base de código e banco de dados Newman (2020). Geralmente, um monólito é mais simples de ser desenvolvido e implantado, pois não exige a complexidade de comunicação entre diferentes componentes. No entanto, à medida que a aplicação cresce, o monólito pode se tornar difícil de manter e escalar, pois qualquer mudança em uma parte do sistema pode afetar outras partes.

De acordo com Newman (2020), os microsserviços são uma abordagem arquitetural em que uma aplicação é dividida em serviços independentes e autônomos, cada um responsável por uma única função ou conjunto de funcionalidades. Cada microsserviço possui sua própria base de código e pode ser desenvolvido, implantado e escalado separadamente. Essa abordagem

permite uma maior flexibilidade e escalabilidade, pois os microsserviços podem ser atualizados e dimensionados independentemente uns dos outros.

2.1.2 Por que utilizar microsserviços?

Segundo Fowler (2019), os microsserviços oferecem benefícios significativos, como escalabilidade, flexibilidade e desenvolvimento ágil. Com a capacidade de escalar e dimensionar partes específicas do sistema de forma independente, os microsserviços permitem uma alocação eficiente de recursos. Além disso, a abordagem distribuída dos microsserviços possibilita um desenvolvimento ágil e distribuído, com equipes independentes trabalhando em serviços separados, isso acelera o tempo de desenvolvimento e permite uma resposta rápida às mudanças de requisitos. Outro benefício é a capacidade de adotar diferentes tecnologias para cada serviço, permitindo escolher a tecnologia mais adequada para cada funcionalidade. No entanto, é importante destacar que a arquitetura de microsserviços também traz complexidade adicional, especialmente na comunicação entre serviços e no gerenciamento de dados distribuídos.

2.1.3 *C4 Model*

O *C4 Model* é uma abordagem de modelagem arquitetural que permite representar a arquitetura de um sistema de software em diferentes níveis de abstração. Ele é composto por quatro níveis principais: Contexto, Contêiner, Componente e Código.

No nível de Contexto, o objetivo é fornecer uma visão geral do sistema e seu ambiente externo. É representado por um diagrama de contexto que mostra as interações entre o sistema em questão e outras entidades externas, como usuários, sistemas legados ou serviços de terceiros.

No nível de Contêiner, a atenção é voltada para a estrutura interna do sistema. Nesse nível, são identificados os principais contêineres que compõem o sistema, como sistemas web, serviços ou bancos de dados. É representado por um diagrama de contêineres, que mostra as relações e as dependências entre os contêineres.

No nível de Componente, o foco está nas partes internas de cada contêiner. Nesse nível, são identificados os principais componentes do sistema e como eles se relacionam entre si. Pode ser representado por diagramas de componentes que descrevem a estrutura interna de cada contêiner e como eles se comunicam.

No nível de Código, a atenção é direcionada para a implementação detalhada dos componentes. Nesse nível, são mostradas as classes, interfaces e outros elementos de código que compõem os componentes identificados anteriormente.

2.1.4 Comparativo entre *C4 Model* e UML

O *C4 Model* e a UML (*Unified Modeling Language*) são duas abordagens distintas para a modelagem de arquitetura de software, cada uma com suas características e propósitos específicos. Embora ambas sejam utilizadas para representar aspectos da arquitetura de um sistema, elas diferem em suas abordagens e escopos, como indica a Figura 2.1.

O *C4 Model* é uma abordagem leve e focada na comunicação da arquitetura de um sistema de software. Ele se concentra em fornecer representações visuais de alto nível, como diagramas de contexto, diagramas de contêineres e diagramas de componentes, para facilitar a compreensão da estrutura e das interações entre os componentes do sistema. O *C4 Model* visa proporcionar uma visão clara e concisa da arquitetura, facilitando a comunicação entre os envolvidos e a tomada de decisões relacionadas ao sistema.

Por outro lado, a UML é uma linguagem de modelagem mais abrangente, que oferece uma variedade de diagramas e notações para descrever diferentes aspectos de um sistema, como a estrutura, o comportamento e a interação entre os componentes (PRESSMAN, 2011). A UML é uma linguagem padrão da indústria e possui uma ampla gama de elementos e diagramas, como diagramas de classes, diagramas de sequência e diagramas de atividades, que permitem uma representação detalhada e precisa do sistema.

Figura 2.1 – Comparação entre *C4 Model* e UML

	C4 Model	UML (Unified Modeling Language)
Foco	Comunicação da arquitetura	Modelagem abrangente de sistemas de software
Níveis de Abstração	Contexto, Contêiner, Componente, Código	Classes, Objetos, Estrutura, Comportamento, Interação, etc.
Representações	Diagramas de contexto, contêineres e componentes	Diagramas de classes, sequência, atividades, etc.
Propósito	Compreensão da estrutura e interações do sistema	Descrição detalhada de diferentes aspectos do sistema
Escopo	Visão de alto nível da arquitetura	Representação detalhada e precisa do sistema
Notação	Simplificada e de fácil compreensão	Extensa, com uma ampla variedade de elementos e diagramas
Aplicação	Arquitetura de software, comunicação entre equipes	Análise, projeto e documentação de sistemas de software

Fonte: Autor

2.1.5 Javascript

Segundo Flanagan (2012), Javascript é uma linguagem de programação de alto nível, criada no meio da década de 90, mais precisamente em 1996 por Brendan Eich. A linguagem permite ao desenvolvedor implementar diversos itens de alto nível de complexidade em páginas web, como animações, mapas, gráficos ou informações que se atualizam periodicamente. No passado, as páginas da web eram como páginas de um livro, ou seja, estáticas e com um layout fixo que exibia informações sem permitir muitas interações do usuário. Com o surgimento do JavaScript, tecnologia utilizada no lado do cliente, as aplicações web se tornaram mais dinâmicas.

O Javascript foi criado como uma linguagem de programação para ser executada no navegador, gerando novos conteúdos e manipulações usando lógica de interface com o usuário e modificando o conteúdo da página que já está no cliente. Entretanto, atualmente é possível utilizar a linguagem em diversas plataformas, como no servidor e também em aplicações mobile e desktop.

2.1.6 Typescript

De acordo com a documentação da linguagem, *Typescript*[1] é um *superset*[2] do Javascript de código aberto desenvolvida pela Microsoft, que surgiu em 2012. O Typescript adiciona recursos como tipagem estática, interfaces, classes e módulos ao JavaScript, tornando-o uma opção popular para desenvolvimento de aplicações web e móveis.

O TypeScript oferece diversas vantagens para os desenvolvedores. Uma delas é a tipagem estática, que permite a definição de tipos de dados para variáveis, parâmetros de função e propriedades de objetos. Essa característica ajuda a evitar erros comuns de digitação, detectando alguns erros em tempo de *transpilação*[3] e fornecendo um melhor suporte para ferramentas de edição de código. Além disso, o TypeScript inclui o conceito de interfaces, que possibilita a definição de contratos entre diferentes partes do código, como estruturas de objetos, classes e funções. A linguagem também possui suporte para módulos, permitindo a organização do código em unidades independentes e reutilizáveis, que podem ser importadas e exportadas. Outro recurso interessante são os *decorators*, que possibilitam a adição de funcionalidades extras a

[1] Disponível em: <https://www.typescriptlang.org/docs>. Acesso em 28.Mar 2023
[2] Disponível em: <https://www.eduardopires.net.br/2012/10/typescript/>. Acesso em 27.Mar 2023
[3] Processo em que um código é transformado em outro mantendo o mesmo nível de abstração

classes e métodos, como validação de entrada de dados, gerenciamento de estado e autorização de acesso.

Apesar de suas vantagens, o TypeScript apresenta algumas desvantagens a serem consideradas. Um ponto negativo é a curva de aprendizado adicional para desenvolvedores acostumados com JavaScript puro. É necessário um tempo adicional para se familiarizar com a sintaxe e os recursos adicionais oferecidos pela linguagem. Além disso, o TypeScript precisa passar por um processo de transpilação para ser convertido em JavaScript antes de ser executado no navegador ou no servidor, o que pode adicionar um tempo a mais ao fluxo de desenvolvimento. Outra desvantagem é o aumento do tamanho do arquivo gerado pelo TypeScript devido à adição de tipos estáticos. Esse aumento no tamanho pode afetar negativamente a velocidade de carregamento da aplicação em ambientes de desenvolvimento, tornando o processo mais lento do que com JavaScript puro.

2.1.7 Node.js

De acordo com Pereira (2014), Node.js é uma plataforma de desenvolvimento de software de código aberto que permite que os desenvolvedores criem *aplicativos*[4] usando Javascript no lado do servidor. Ele foi criado em 2009 por Ryan Dahl e é baseado no interpretador de Javascript V8 da Google.

O Node.js é projetado para facilitar a criação de aplicativos escaláveis, altamente disponíveis e de alto desempenho. A plataforma é construída com base na biblioteca *libuv*[5], na qual a deixa altamente eficiente e escalável para softwares com alta carga de I/O, porque o modelo de E/S não bloqueante e orientado a eventos permite que o Node.js execute várias operações de entrada/saída de forma assíncrona, em vez de bloquear o processo enquanto aguarda a conclusão de uma operação de E/S.

Além disso, o Node.js tem uma arquitetura modular e é compatível com uma ampla variedade de bibliotecas e módulos de terceiros, o que permite que os desenvolvedores criem aplicativos complexos de maneira rápida e eficiente. A plataforma também é conhecida por ter uma comunidade de desenvolvedores ativa e engajada, que contribuem com bibliotecas, módulos e *frameworks* de código aberto.

[4] No contexto deste trabalho, aplicativos são qualquer software que pode ser encapsulado em contêineres

[5] Disponível em: <https://libuv.org/>. Acesso em 28.Mar 2023

2.1.8 Nest.js

De acordo com a documentação oficial, *Nest.js*[6] é um *framework* do Node.js com foco no desenvolvimento web. Possui código aberto e foi criado para ajudar os desenvolvedores a construir aplicativos de servidor escaláveis e eficientes usando a arquitetura modular do Typescript. Ele foi criado em 2017 por Kamil Myśliwiec e sua equipe, e é construído em cima do *Express*[7], um dos *frameworks* mais populares do Node.js para construção de aplicativos web.

O Nest.js é baseado em vários princípios de *design*, incluindo o padrão de arquitetura Model-View-Controller (MVC), *Inversion of Control* (IoC), injeção de dependência e programação orientada a objetos (POO). Ele também oferece suporte integrado para a arquitetura de microserviços, tornando-o uma escolha popular para a construção de aplicativos escaláveis e complexos.

O *framework* oferece uma série de vantagens significativas para os desenvolvedores. Uma delas é a sua alta modularidade, seguindo o princípio de separação de responsabilidades, isso permite que os aplicativos sejam divididos em módulos menores e reutilizáveis, facilitando a manutenção e a escalabilidade do código. Além disso, o Nest.js possui suporte integrado para a arquitetura de microserviços, permitindo que os desenvolvedores criem aplicativos escaláveis e resilientes, que podem ser facilmente divididos em serviços independentes. O *framework* também oferece uma ampla gama de recursos integrados, simplificando o desenvolvimento de APIs, documentações, testes e muito mais. Outra vantagem é a flexibilidade que o Nest.js proporciona, permitindo que os desenvolvedores adotem diferentes arquiteturas, como a arquitetura *Hexagonal*[8], de acordo com as necessidades do projeto.

Apesar de suas vantagens, o Nest.js também apresenta algumas desvantagens. Uma delas é a curva de aprendizado associada à *framework*. Como o Nest.js é baseado no TypeScript e segue um conjunto específico de princípios de design, pode levar tempo para que os desenvolvedores se familiarizem com essas tecnologias e conceitos, especialmente para aqueles que não têm experiência prévia com elas. Além disso, a modularidade do Nest.js, embora seja uma vantagem, também pode levar a uma maior complexidade do código, caso a granularidade dos módulos não seja bem definida. Isso pode dificultar a compreensão do código, especialmente para desenvolvedores que não estão familiarizados com a arquitetura do Nest.js.

---

[6] Disponível em: <https://docs.nestjs.com/>. Acesso em 28.Mar 2023

[7] Disponível em: <https://expressjs.com/pt-br/>. Acesso em 28.Mar 2023

[8] Disponível em: <https://engsoftmoderna.info/artigos/arquitetura-hexagonal.html>. Acesso em 28.Mar 2023

2.1.9 Docker

De acordo com Vitalino e Castro (2016), Docker é uma plataforma de conteinerização de software que permite aos desenvolvedores empacotar, distribuir e executar aplicativos em qualquer lugar, independentemente do sistema operacional e do ambiente de execução. Essa tecnologia utiliza o conceito de contêineres, que são unidades isoladas de software que incluem tudo o que um software precisa para ser executado, como código-fonte, bibliotecas, dependências e configurações.

Além disso, o Docker torna o processo de empacotamento e distribuição de softwares mais rápido e eficiente, pois elimina a necessidade de enviar grandes arquivos de código ou depender de ambientes de desenvolvimento específicos.

A utilização de contêineres Docker traz diversas vantagens para o desenvolvimento de software. Uma delas é a redução do tempo de configuração do ambiente de desenvolvimento. Ao criar *imagens*[9] de contêineres, a configuração é feita apenas uma vez e pode ser replicada em diferentes máquinas, agilizando o processo de configuração. Além disso, o uso de contêineres permite a padronização do ambiente de desenvolvimento, garantindo que todas as equipes tenham o mesmo ambiente, independentemente do sistema operacional ou ferramentas individuais. Isso facilita a colaboração e a implantação em ambientes de homologação e produção, que serão idênticos ao ambiente de desenvolvimento. Outra vantagem é a escalabilidade proporcionada pelos contêineres. A inicialização rápida dos contêineres permite uma escalabilidade ágil do software, principalmente em escala horizontal, onde novos contêineres podem ser adicionados facilmente.

Apesar de suas vantagens, o uso de contêineres Docker também apresenta algumas desvantagens. Em ambientes complexos, a orquestração dos contêineres pode exigir um maior grau de conhecimento por parte da equipe. Configurar, orquestrar e monitorar os contêineres em ambientes mais complexos pode ser um desafio, demandando um maior domínio das ferramentas e conceitos envolvidos. Além disso, o uso de muitos contêineres em um software pode consumir significativamente os recursos de hardware, principalmente em máquinas de desenvolvimento. Isso pode exigir máquinas mais potentes para garantir um desempenho adequado e evitar problemas de lentidão ou falta de recursos durante o desenvolvimento.

---

[9] Disponível em: <https://www.mundodocker.com.br/o-que-e-uma-imagem/>. Acesso em 30.Mar 2023

2.1.10 Kubernetes

Segundo Santos (2019), Kubernetes ou K8s, é uma plataforma de código aberto amplamente utilizada para automatizar, implantar, escalar e gerenciar contêineres. Sendo assim, ele oferece um ambiente robusto e flexível para facilitar a orquestração e o balanceamento de carga em um ambiente distribuído. O Kubernetes foi desenvolvido pela Google e agora é mantido pela Cloud Native Computing Foundation (CNCF), o que garante seu desenvolvimento contínuo e apoio da comunidade.

O Kubernetes oferece vantagens significativas no gerenciamento de aplicativos em contêineres. Sua capacidade de escalabilidade permite que os aplicativos se adaptem facilmente às demandas de carga, adicionando ou removendo nós de trabalho conforme necessário. Além disso, sua resiliência avançada garante a disponibilidade contínua dos aplicativos, reiniciando automaticamente contêineres com falha e distribuindo a carga de forma equilibrada. Outra vantagem é a portabilidade do Kubernetes, que permite que os aplicativos sejam executados em diferentes ambientes de infraestrutura, incluindo provedores de nuvem variados, oferecendo flexibilidade e liberdade para escolher a melhor plataforma de hospedagem.

No entanto, é importante considerar as desvantagens, como a necessidade de recursos significativos para operar o Kubernetes, tanto em termos de processamento, armazenamento e largura de banda de rede. Além disso, a complexidade da plataforma requer um alto nível de conhecimento técnico para configurar e gerenciar adequadamente um cluster do Kubernetes em produção, o que pode representar um desafio para equipes sem experiência prévia com a tecnologia.

2.1.11 PostgreSQL

Segundo a documentação, *PostgreSQL*[10] também conhecido como Postgres, é um sistema de gerenciamento de banco de dados relacional de código aberto que utiliza a linguagem SQL para consultar e gerenciar dados. Ele é uma das opções mais populares para aplicações que exigem um grande volume de dados ou que possuem requisitos de alta disponibilidade e desempenho.

Entre as vantagens, destaca-se a capacidade de lidar com uma grande quantidade de dados e consultas complexas, sua segurança avançada com suporte a autenticação e criptografia,

[10] Disponível em: <https://www.postgresql.org/docs>. Acesso em 19.Abr 2023

linguagem procedural *PL/pgSQL*[11], além de ser uma plataforma de código aberto com uma grande comunidade de desenvolvedores e usuários ativos.

No entanto, o PostgreSQL também possui algumas desvantagens, como a complexidade de configuração e administração, que pode exigir conhecimento técnico avançado, e a sua performance em casos de operações de escrita massivas, que pode ser inferior a outros bancos de dados *NoSQL*[12].

2.1.12 RabbitMQ

O RabbitMQ se caracteriza como um software de mensageria de código aberto que funciona como um intermediário para processar e distribuir mensagens entre sistemas Roy (2017). Sendo assim, é capaz de suportar diversos protocolos, pórem o mais utilizado é o *Advanced Message Queuing Protocol (AMQP)*[13]. A ferramenta é capaz de gerenciar grandes volumes de mensagens de diferentes tipos de aplicações e serviços distribuídos. É um componente importante em muitas arquiteturas de microsserviços e permite que os desenvolvedores criem sistemas altamente escaláveis e distribuídos.

O uso do RabbitMQ como um sistema de mensageria traz várias vantagens para o desenvolvimento de aplicativos. Uma delas é a escalabilidade, permitindo dimensionar horizontalmente o sistema para lidar com um grande número de conexões e mensagens. Além disso, o RabbitMQ oferece opções de alta disponibilidade, como clusterização e espelhamento de filas, garantindo que as mensagens sejam processadas mesmo em caso de falhas. Outra vantagem é a integração fácil com várias linguagens de programação e sistemas, tornando-o flexível e adequado para diferentes projetos.

Apesar de suas vantagens, o uso do RabbitMQ também possui algumas desvantagens a serem consideradas. Uma delas é a complexidade do sistema, que pode exigir uma configuração e manutenção complexas, especialmente em ambientes de grande escala. Outra desvantagem é a possibilidade de alta latência adicionada pelo uso de uma camada intermediária, como o RabbitMQ, no processamento de mensagens. Isso pode ser um problema em casos em que a latência é um fator crítico para o desempenho do sistema. Além disso, o gerenciamento de múltiplas filas e consumidores pode exigir uma configuração cuidadosa para garantir que as mensagens sejam processadas corretamente, evitando atrasos ou perdas indesejadas de mensagens.

[11] Disponível em: <https://www.postgresql.org/docs/current/plpgsql.html>. Acesso em 19.Abr 2023
[12] Disponível em: <https://aws.amazon.com/pt/nosql/>. Acesso em 19.Abr 2023
[13] Disponível em: <https://www.iso.org/standard/64955.html>. Acesso em 19.Abr 2023

2.1.13 React

Segundo a documentação, *React*[14] é uma biblioteca de código aberto para construção de interfaces de usuário desenvolvida pelo Facebook. Ela permite a criação de interface dinâmicas e reutilizáveis com uma sintaxe declarativa, utilizando Javascript ou TypeScript.

A principal característica do React é o conceito de componentes. Os componentes são blocos de construção reutilizáveis para a criação de interfaces. Eles podem ser pensados como peças de um quebra-cabeça, onde cada peça é um componente que pode ser combinado com outros para formar a interface desejada. Esses componentes podem ser facilmente compartilhados entre diferentes partes do sistema, na qual ajuda a reduzir a duplicidade de código e a manter uma base de código limpa e organizada, a fim de melhorar a manutenibilidade e evolução da aplicação.

Outra característica importante do React é a capacidade de atualizar a interface de forma eficiente. O React utiliza uma técnica chamada de virtual DOM (DOM virtual), que permite atualizar apenas as partes da interface de usuário que foram alteradas, em vez de atualizar a interface inteira. Isso melhora significativamente o desempenho, pois reduz o número de operações necessárias para atualizar a interface.

Além disso, o React tem uma grande comunidade de desenvolvedores e uma vasta gama de bibliotecas e ferramentas disponíveis para melhorar e facilitar o desenvolvimento. Ele pode ser usado tanto para desenvolvimento web quanto para aplicativos móveis, por meio de *frameworks* como o React Native.

2.1.14 Next.js

De acordo com a documentação, *Next.js*[15] é um *framework* para React que permite criar aplicações web. A ferramenta oferece uma ampla gama de recursos simplificar o processo de desenvolvimento. A tecnologia combina funcionalidades do React com novos recursos, como renderização do lado do servidor e roteamento de páginas, tornando-a uma solução completa.

O Next.js possui várias vantagens para o desenvolvimento de aplicativos web. Uma delas é a capacidade de renderização no servidor, permitindo que a página HTML completa seja gerada no servidor e entregue ao navegador do usuário. Isso resulta em tempos de carregamento mais rápidos e oferece benefícios de SEO aprimorado. Além disso, o Next.js suporta a geração

[14] Disponível em: <https://react.dev/>. Acesso em 19.Abr 2023
[15] Disponível em: <https://nextjs.org/docs>. Acesso em 19.Abr 2023

de sites estáticos, o que é ideal para páginas com pouca interação ou mudança de interface, como páginas de vendas de produtos temporários. Outra vantagem é o roteamento nativo fornecido pelo Next.js, que simplifica o processo de roteamento de páginas em comparação com o React puro, eliminando a necessidade de implementação ou uso de bibliotecas de terceiros.

Apesar de suas vantagens, o Next.js também apresenta algumas desvantagens. Uma delas é a curva de aprendizado íngreme para desenvolvedores iniciantes, uma vez que o Next.js é um *framework* que requer uma compreensão sólida do React e da programação no lado do servidor para ser usado efetivamente. Além disso, o Next.js é mais complexo do que o React puro, adicionando mais complexidade ao processo de desenvolvimento. Isso pode resultar em um desenvolvimento mais demorado e aumentar a possibilidade de erros. Outra desvantagem é a possibilidade de sobrecarga de recursos, especialmente ao renderizar no servidor. O Next.js pode consumir muitos recursos do sistema, o que pode afetar o desempenho e a escalabilidade, especialmente em servidores de menor capacidade.

2.2 Algoritmos Implementados no projeto

Nesta seção, apresenta-se os algoritmos implementados neste projeto. A ênfase está em fornecer uma visão geral dos algoritmos desenvolvidos, descrevendo suas principais características e funcionalidades.

2.2.1 Dijkstra

De acordo com GOLDBARG (2012), o algoritmo de Dijkstra é um algoritmo que calcula o caminho de custo mínimo entre vértices de um grafo. Escolhido um vértice como raiz da busca, este algoritmo calcula o custo mínimo deste vértice para todos os demais vértices do grafo. Ele foi proposto pelo cientista da computação holandês Edsger W. Dijkstra em 1956 e é amplamente utilizado em aplicações de redes de computadores, sistemas de transporte e roteamento de pacotes de dados.

Uma das principais vantagens do algoritmo de Dijkstra é sua eficiência em encontrar o caminho mais curto em grafos com pesos não negativos. No entanto, ele pode ser ineficiente em

grafos com pesos negativos e, em tais casos, outros algoritmos, como o, *algoritmo de Bellman-Ford*[16] são mais adequados.

Por fim, a complexidade do algoritmo em sua implementação mais simples é $O(n^2)$, mas felizmente é possível obter melhores resultados, adicionando na implementação um *Heap* com uma lista de prioridade e adjacência a complexidade é reduzida para $O((m+n)\log n)$, sendo n e m, a quantidade de vértices e arestas, respectivamente.

2.2.2 Backtracking

Segundo Chang (2003), o algoritmo de *Backtracking* é uma técnica de busca utilizada em problemas de otimização combinatória, que busca encontrar soluções em um espaço de busca com diversas possibilidades. Dessa forma, a abordagem utilizada é baseada em tentativa e erro, onde a solução é construída incrementalmente, e em cada etapa, é realizada uma verificação para determinar se a solução parcial criada é satisfatória ou não.

Se a solução parcial for factível, o algoritmo avança para o próximo passo da construção da solução, caso contrário, ele retrocede um passo e tenta uma outra possibilidade. Esse processo é repetido até que a solução completa seja encontrada ou até que se prove que não existe uma solução.

Uma das principais desvantagens do algoritmo de *Backtracking* é o seu alto custo computacional, uma vez que ele pode gerar um grande número de soluções parciais que precisam ser avaliadas, ou seja, sua complexidade é exponencial.

[16] Disponível em: <https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/bellman-ford.html>. Acesso em 20.Abr 2023

3 METODOLOGIA DE DESENVOLVIMENTO

Este capítulo tem como objetivo principal demonstrar os procedimentos adotados e a aplicação prática dos conceitos, tecnologias e ferramentas utilizados no desenvolvimento do software criado. Por meio de uma abordagem passo a passo, são apresentados os processos e etapas seguidos, desde a concepção até a implementação do projeto. Além disso, são discutidos os desafios encontrados ao longo do caminho e as soluções adotadas para superá-los. Ao final deste capítulo, espera-se fornecer uma visão abrangente e detalhada do desenvolvimento do software, destacando a eficácia e a relevância das escolhas feitas em relação aos conceitos, tecnologias e ferramentas utilizados.

3.1 Procedimentos

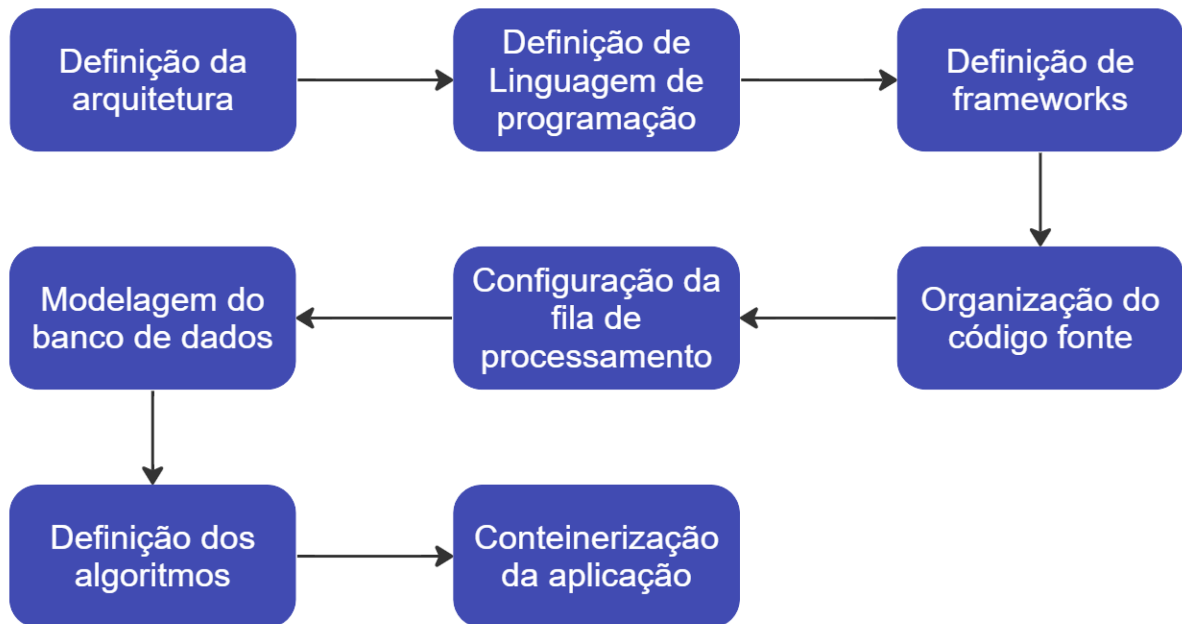
Nesta seção são apresentados os procedimentos realizados para a criação do software proposto. Iniciando-se com a definição da arquitetura, descrito na seção 3.2. Em seguida, destaca-se a definição da linguagem de programação e *frameworks*, na seção 3.3. A organização do código fonte é descrito na seção 3.4. Em seguida, a definição e configuração da fila de processamento, com RabbitMQ, mais detalhes na seção 3.5. Nas seções 3.6 e 3.7, é descrito a modelagem do banco de dados e a definição dos algoritmos para a resolução de *puzzles*, respectivamente. Por fim, a conteinerização da aplicação, na seção 3.8. A Figura 3.1 exemplifica de forma visual.

3.2 Definição da arquitetura do software

A arquitetura foi definida utilizando o *C4 model*. É uma abordagem visual para representar arquiteturas de software de forma clara e concisa. Inicialmente, foi modelado o nível de contexto, que tem como objetivo fornecer uma visão ampla dos contêineres arquiteturais, mostrando como a comunicação entre as partes ocorre e onde os usuários interagem com o software.

Na Figura 3.2 contém três containêres e um ator envolvido. Primeiramente, o usuário é o ator na qual irá interagir com o software, essa comunicação é possível através do contêiner ***Puzzle Application***, que representa a aplicação na qual será possível fazer a criação de um novo *puzzle* e a visualização dos mesmos.

Figura 3.1 – Passo a passo seguido



Fonte: Autor

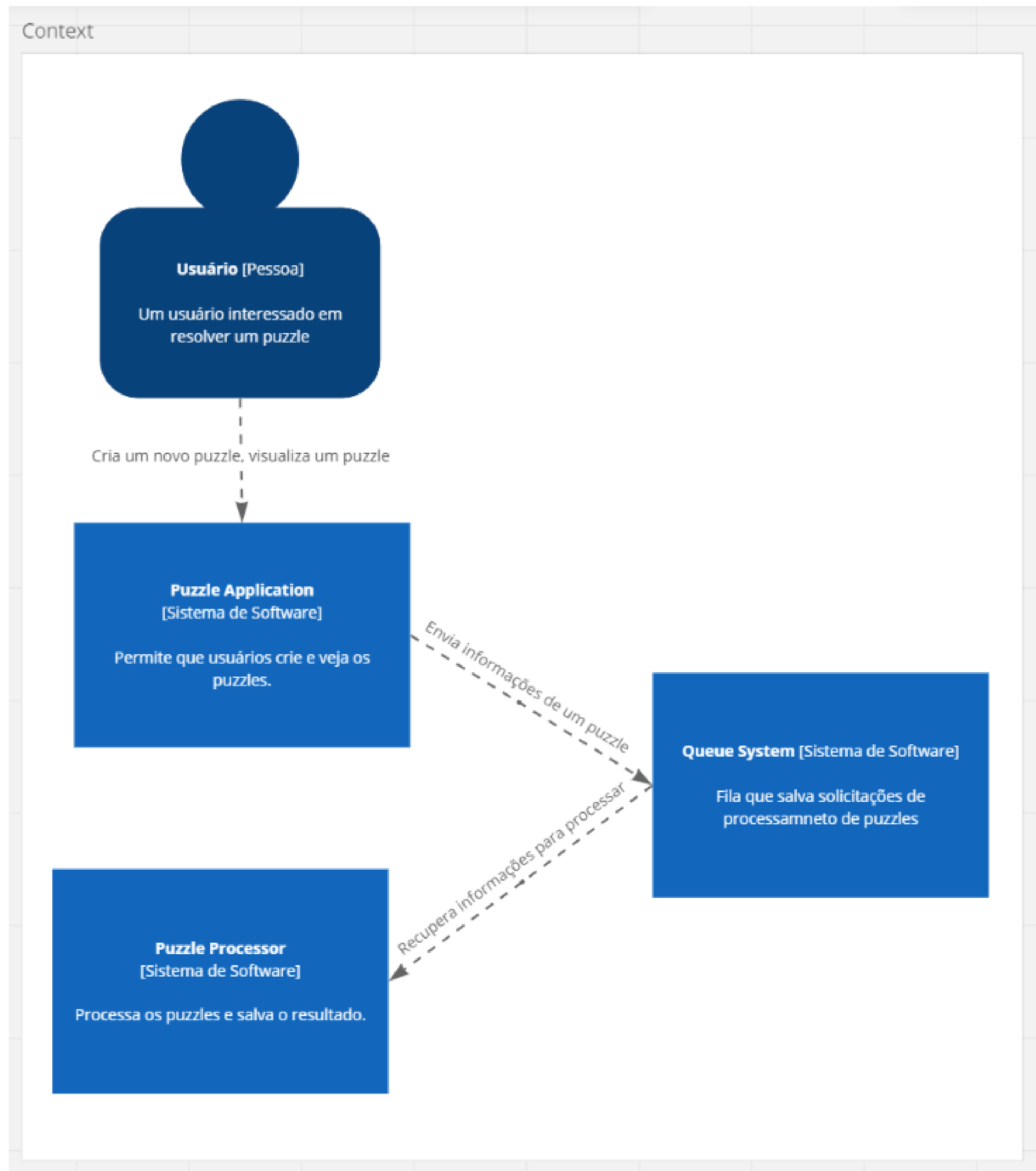
Visando escalabilidade e uma boa experiência para o usuário, foi modelado um processamento assíncrono para a resolução dos *puzzles*, tendo isso em vista, foi necessária a adoção de uma fila, ***Queue System***. Dessa forma, o tratamento dos dados segue conforme a capacidade da aplicação em processar as informações (XU, 2020). Essa decisão foi pautada na ideia de que um *puzzle* pode demorar para que sua resolução seja concluída, assim o usuário e a aplicação não fica parada esperando a requisição ser finalizada, possibilitando uma experiência mais fluída e satisfatória para o ator final.

Por fim, o último contêiner definido foi o ***Puzzle Processor***, que possui a responsabilidade de pegar os dados pendentes na fila e de fato realizar a resolução dos *puzzles*. Em termos de escala, caso a demanda por muitos processamentos aumente, é possível ampliar a capacidade dessa aplicação, consequentemente, tornando-a com alta disponibilidade e escalabilidade.

3.3 Linguagem de programação e *frameworks*

Após a definição da arquitetura em alto nível com contexto, foi realizado a modelagem a nível de contêineres, seguindo a ideia do *C4 Model*, nesse nível é dado um zoom em cada contêiner para se detalhar seus componentes. A nível de componente já é possível determinar quais linguagens e ferramentas foram utilizadas.

Figura 3.2 – Modelagem de contexto do *C4 model*



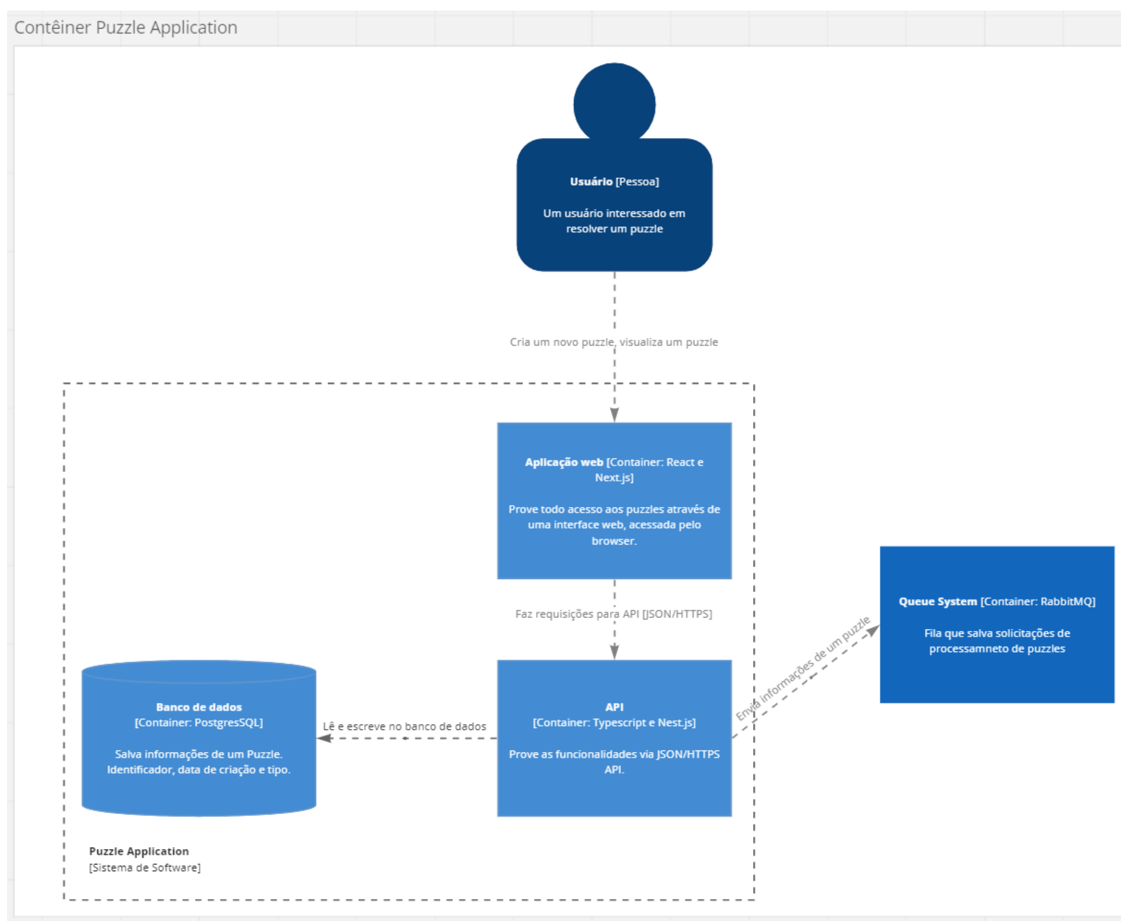
Fonte: Autor

Na Figura 3.3, é possível visualizar a modelagem do contêiner ***Puzzle Application***. O primeiro componente é a **Aplicação web**, que é responsável por prover todo acesso as funcionalidades atráves de uma interface web, acessada pelo *browser*. Para o desenvolvimento desse componente foi escolhido a linguagem de programação Javascript, juntamente com o Typescript, e também com React e seu *framework* Next.js. Segundo a pesquisa de 2023 do

Stackoverflow[1], React foi a ferramenta de *frontend* mais utilizada no ano, tanto por estudantes, quanto por profissionais da área, portanto considera-se como uma ferramenta promissora e com uma grande comunidade ativa de desenvolvedores.

O componente ***API*** é o *backend* da aplicação, é responsável por aplicar as regras de negócio, fazer a comunicação com o banco de dados e a salvar dados na fila de processamento. Seguindo a mesma linha de pensamento do *frontend*, a linguagem escolhida também foi o Javascript com Typescript, juntamente com Node.js e seu framework Nest.js. O Node.js foi a ferramenta de 2023 mais utilizada, segundo a pesquisa do *Stackoverflow*[2]. O *framework* Nest.js também é o mais popular, chegando a aproximadamente 2 milhões de *downloads* semanais, no *repósitorio oficial*[3], até a presente data desse trabalho.

Figura 3.3 – Modelagem do contêiner Puzzle Application

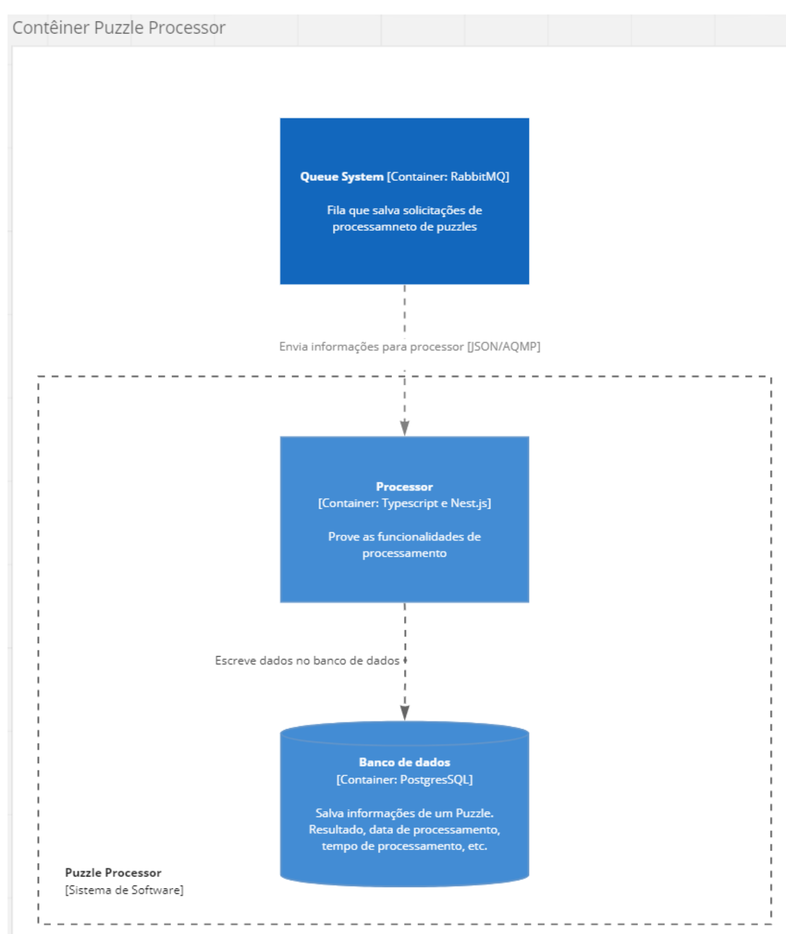


Fonte: Autor

[1] Disponível em: <https://survey.stackoverflow.co/2023>. Acesso em 23.Jun 2023
[2] Disponível em: <https://survey.stackoverflow.co/2023>. Acesso em 23.Jun 2023
[3] Disponível em: <https://github.com/nestjs/nest>. Acesso em 23.Jun 2023

Na Figura 3.4 o componente ***Processor***, que se trata de uma aplicação *backend* que segue a mesma arquitetura da ***API*** mencionada anteriormente. Esse componente é crucial para o funcionamento adequado do sistema, pois é responsável por processar as informações recebidas e executar as tarefas necessárias para o funcionamento do software. Além disso, em próximos tópicos, será feita uma análise mais aprofundada do componente ***Queue System***, que gerencia as filas de tarefas do sistema, e do banco de dados, que armazena as informações do software e permite a sua consulta e atualização pelos usuários.

Figura 3.4 – Modelagem do contêiner Puzzle Processor



Fonte: Autor

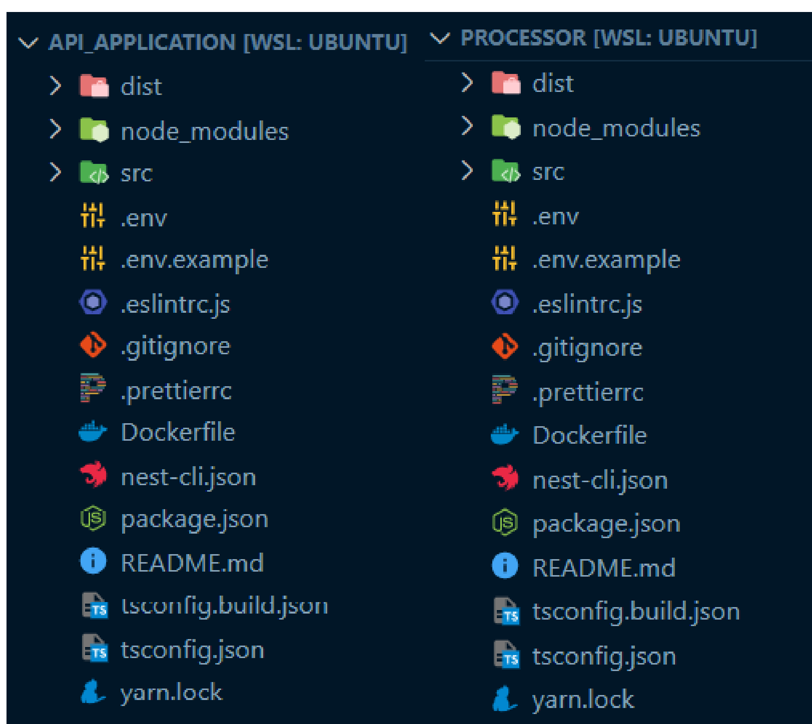
3.4 Organização dos códigos fonte

Segundo (MARTIN, 2019), quando um software é feito com boas práticas e uma boa organização na sua estrutura, é necessário uma pequena fração dos recursos humanos para ser criado e mantido. As mudanças são simples e rápidas, os poucos problemas que surgem não impactam em outras funcionalidades. O esforço é minimizado enquanto a manutenibilidade

e flexibilidade são maximizadas. Foi com esse conhecimento que os softwares descritos nessa seção foram criados, ambos os projetos *backend* e *frontend*, seguindo boas práticas previamente adotadas pelos *frameworks* para se obter mais agilidade, mas também com a qualidade em mente.

Inicialmente, será descrito a estrutura dos projetos de *backend*, ***API Application*** e ***Processor***, conforme mostra a Figura 3.5. Vale ressaltar que, ambas as estruturas de código são parecidas tendo diferenças sutis em arquivos internos dos módulos, pois utilizam o mesmo princípio de arquitetura como base. Posteriormente, de forma semelhante a estrutura utilizada no *frontend* será detalhada.

Figura 3.5 – Estrutura dos projetos de *backend*



Fonte: Autor

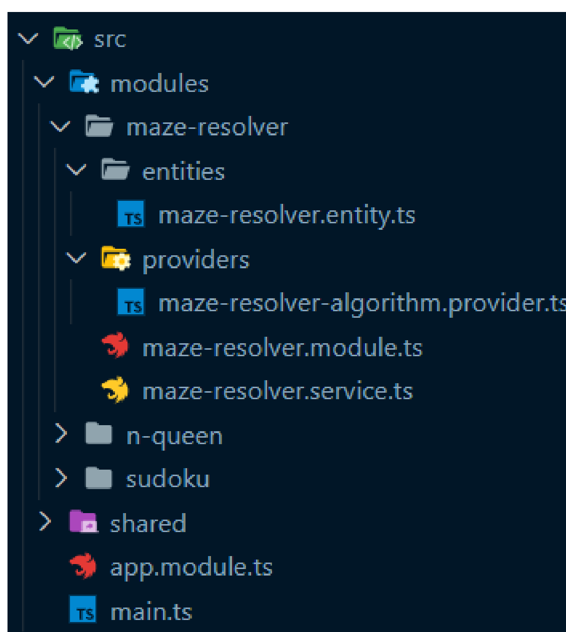
A imagem destaca diversas pastas e arquivos, entre as pastas podemos destacar:

- **dist**: armazena arquivos frutos da transpilação do Typescript, ou seja, são armazenados arquivos Javascript.
- **node_modules**: armazena arquivos de terceiros que o projeto necessita para funcionar, como bibliotecas, arquivos de frameworks, etc.
- **src**: armazena arquivos do código fonte do projeto, regras de negócio, conexão com o banco de dados, etc. A Figura 3.6 detalha esta pasta.

Os demais arquivos são de configuração, destaco os mais relevantes:

- **Dockerfile**: arquivo de configuração do Docker, utilizado para criar uma imagem do projeto.
- **nest-cli.json**: utilizado na configuração de comandos via cli do *framework* Nest.js.
- **package.json**: arquivo utilizado para salvar todas as dependências externas que o projeto necessita, utilizados pelo gerenciador de dependências.
- **tsconfig.json**: Arquivo de configuração do Typescript.

Figura 3.6 – Estrutura da pasta src do *backend* ***Processor***



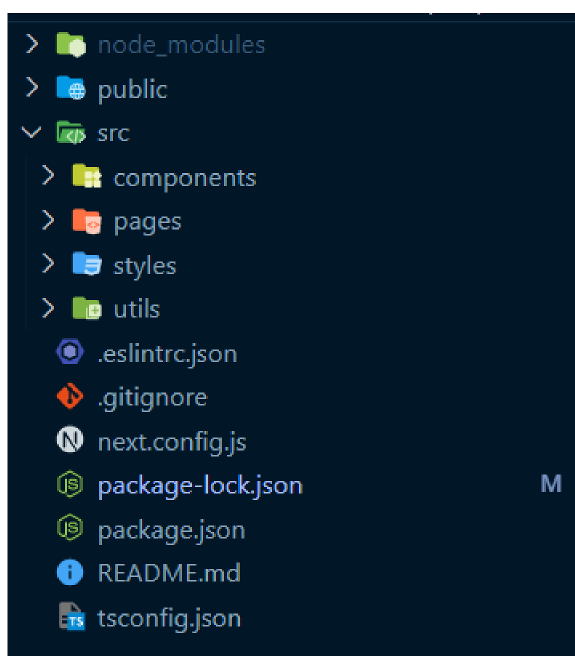
Fonte: Autor

Como já citado anteriormente, o *backend* utiliza o *framework* Nest.js, ele sugere uma estrutura de módulos para se arquitetar os projetos. Com os módulos é possível ter um contexto limitado e isolado de outras partes do código, tornando-o com baixo acoplamento e altamente reutilizável. Através da imagem é possível ver a pasta de módulos e uma pasta compartilhada, ***shared***. A pasta de módulos é composta por serviços, provedores, entidades e um arquivo de configuração do módulo, com esse conjunto é possível definir toda a regra de negócio e configurações de comunicação com o banco de dados. Na pasta compartilhada, é definido arquivos que são reaproveitados em todos os outros módulos, como arquivos de configuração do RabbitMQ.

Por fim, a estrutura do *frontend* segue uma ideia de pastas parecidas com o *backend*, principalmente em relação a arquivos de configuração, como **tsconfig.json** e **package-json**. A diferença se da na estruturação do código fonte na pasta **src** e também nas configurações como o **next.config.js**. Como relata a Figura 3.7, podemos destacar algumas pastas:

- **public**: utilizada para salvar arquivos de imagens, como favicons.
- **componentes**: utilizada para a criação de componentes reutilizáveis por todo o projeto.
- **pages**: utilizada para a criação de páginas como um todo, uma página é composta por um conjunto de componentes.
- **styles**: salva configurações de estilo, como arquivos css.
- **utils**: salva funções úteis que podem ser utilizadas em todo o projeto.

Figura 3.7 – Estrutura do projeto *frontend*



Fonte: Autor

Github das aplicações, *frontend*: https://github.com/jeversonjv/tcc-frontend, *backend*: https://github.com/jeversonjv/tcc-backend, para mais detalhes de implementação.

3.5 Fila de processamento

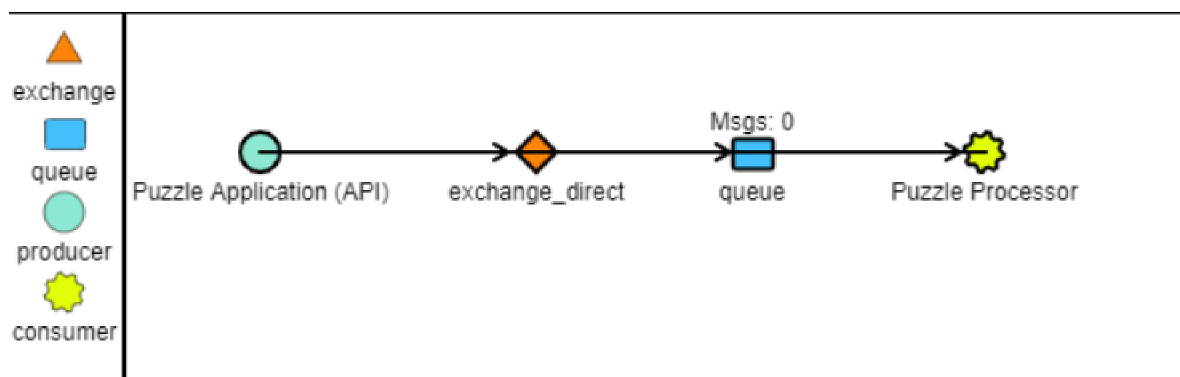
Tendo em vista os aspectos de escalabilidade de sistemas, ao se utilizar filas de processamento de dados, os dados são processados por consumidores conforme a demanda, capacidade

e disponibilidade individual de cada servidor (XU, 2020). Sendo assim, a ferramenta escolhida para se criar esse modelo arquitetural foi o RabbitMQ.

O RabbitMQ usa um modelo de *publisher/subscriber* para gerenciar as mensagens na fila. A princípio, um *publisher* é um software que deseja enviar uma mensagem para uma fila e o *subscriber* é alguém interessado em receber essas informações. Um *publisher* não consegue enviar os dados diretamente para uma fila, para isso é necessário que ele se comunique com um *exchange*, assim, através de regras o *exchage* define para qual fila a mensagem será direcionada. No projeto, foi configurado para que o *exchange* envie mensagens diretamente para uma fila baseado em seu nome.

Em sequência, para que as mensagens sejam lidas da fila é utilizado o conceito de *subscribers*, que nada mais são do que consumidores que ficam observando a fila e quando uma mensagem é recebida ela é direcionada para o mesmo. Pode-se ter vários consumidores ouvindo uma fila, assim o RabbitMQ decide para qual será enviado baseado em configurações, no software projetado foi configurado para que ele use a estratégia de *Round-Robin*, que de forma sucinta, o RabbitMQ irá balancear a carga entre os consumidores e enviar de forma igual entre eles. Por fim, quando o consumidor termina de processar a mensagem ele devolve um sinal para que o servidor do RabbitMQ entenda que a mensagem foi processada e que pode ser removida da fila, esse conceito é chamado de *acknowledgement*. A Figura 3.8 detalhada como foi feita a configuração do RabbitMQ no projeto.

Figura 3.8 – Modelagem do RabbitMQ



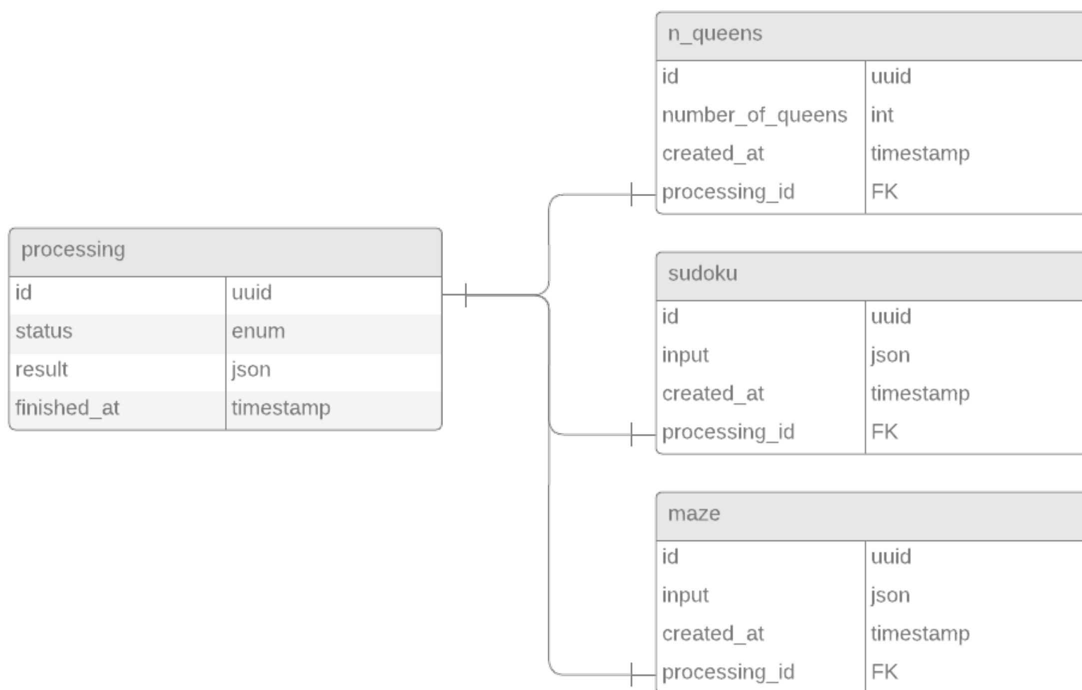
Fonte: Autor

3.6 Definição e modelagem do banco de dados

O banco de dados é um elemento fundamental para qualquer aplicação de software. É através dele que as informações utilizadas pela aplicação são armazenadas e gerenciadas

(DATE, 2004). Conforme a tecnologia avança, o volume de dados gerados pelas empresas tem crescido exponencialmente, tornando a utilização de um banco de dados ainda mais importante. Com um banco de dados bem estruturado e otimizado, é possível ter acesso rápido e seguro às informações necessárias para o funcionamento da aplicação. Por isso, é essencial definir uma boa estrutura e modelagem do banco de dados desde o início do projeto, para garantir a integridade e a eficiência da aplicação. Tendo isso em mente, para o projeto foi escolhido um banco de dados relacional, pois as informações armazenadas possuem uma estrutura definida e também considerando escopo do projeto, a linguagem SQL irá suprir bem a necessidade, uma vez que ela permite realizar extrações sofisticadas e complexas de dados. Na Figura 3.9 é possível ver a modelagem dos dados.

Figura 3.9 – Modelagem do banco de dados



Fonte: Autor

A tabela ***processing*** é responsável por salvar dados comuns a qualquer tipo de processamento de *puzzle*, salvando informações como o *status*, resultado e a data que o processamento foi finalizado. Vale ressaltar que, o status é um campo do tipo *enum* que pode ter os valores *PENDING* para quando o processamento ainda está pendente e *COMPLETED* para quando já

foi finalizado. As demais tabelas possui uma relação *one-to-one* com a tabela ***processing*** e as mesmas são responsáveis por salvar dados que são intrínsecos de cada *puzzle*.

Após estabelecer a estrutura das tabelas e como as informações serão armazenadas, é crucial selecionar um Sistema Gerenciador de Banco de Dados (SGBD) adequado para o projeto. Dessa maneira, o PostgreSQL foi escolhido como o SGDB a ser utilizado, por ser um banco de dados relacional popular e amplamente utilizado. Em complemento, é uma opção robusta e confiável, com uma comunidade ativa de desenvolvedores e usuários. Além disso, oferece uma variedade de extensões e plug-ins para personalização e otimização do desempenho, tornando-se uma escolha sólida para o projeto.

3.7 Algoritimos para resolução de *puzzles*

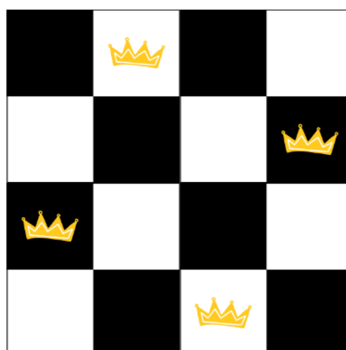
3.7.1 *NQueens*

O problema das n rainhas é o problema matemático de dispor n rainhas em um tabuleiro de dimensão nxn, de forma que nenhuma delas seja ameaçada. Para tanto, é necessário que duas rainhas quaisquer não estejam em uma mesma linha, coluna ou diagonal.

A proposta desenvolvida retorna todas as soluções possíveis para uma quantidade informada, ou seja, o usuário informa um n >= 4 e o sistema devolve todas as organizações possíveis das rainhas no tabuleiro.

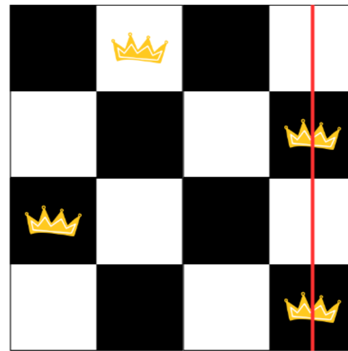
O algoritmo utilizado foi o *backtracking*, pois é um método geral que pode ser aplicado para resolver o problema das n rainhas. Ele explora todas as possibilidades de colocação das rainhas no tabuleiro, utilizando recursão para testar cada configuração e voltando atrás (*backtracking*) quando uma solução inválida é encontrada. Embora seja simples de implementar, o desempenho do algoritmo de *backtracking* pode ser limitado para tamanhos grandes de n.

Figura 3.10 – Solução válida para um tabuleiro 4x4



Fonte: Autor

Figura 3.11 – Solução inválida para um tabuleiro 4x4



Fonte: Autor

3.7.2 *Sudoku*

O *Sudoku* é um jogo de quebra-cabeças, o objetivo é preencher uma grade 9x9 dividida em nove sub grades 3x3 com números de 1 a 9, sem repetições. O quebra-cabeça começa com algumas células pré-preenchidas, e o jogador deve usar a lógica e a dedução para preencher as células vazias corretamente.

A proposta desenvolvida recebe a uma matriz de números pré-preenchida e retorna-a com todos os números posicionados. O algoritmo utilizado foi o *backtracking*, ele tenta todas as possibilidades de preenchimento das células, utilizando recursão para testar cada número em cada célula e voltando atrás (*backtracking*) quando uma solução inválida é encontrada.

Figura 3.12 – Exemplo de grade pré-preenchida

			3			2		
				9		6		1
				1	2			
			9	2	1	5		
		1	7			8		3
9	5	7				1		
	8	6						2
	7			5		9		
	9							5

Fonte: Autor

Figura 3.13 – Exemplo de grade preenchida

4	1	5	3	6	7	2	8	9
7	2	8	5	9	4	6	3	1
6	3	9	8	1	2	4	5	7
8	4	3	9	2	1	5	7	6
2	6	1	7	4	5	8	9	3
9	5	7	6	3	8	1	2	4
5	8	6	1	7	9	3	4	2
1	7	2	4	5	3	9	6	8
3	9	4	2	8	6	7	1	5

Fonte: Autor

3.7.3 *Maze solver*

O *Maze Solver*, também conhecido como solucionador de labirintos, é um algoritmo que busca encontrar uma solução para um labirinto. Um labirinto é uma estrutura composta por caminhos e obstáculos, onde o objetivo é encontrar um caminho que leve da entrada até a saída.

A proposta desenvolvida recebe uma matriz com os números 1 e 0, em que 1 representa caminhos que podem ser percorridos e 0 os que não podem passar. É fixado a posição 0,0 como o início e a última posição como a saída do labirinto. Para encontrar o caminho, foi utilizado o algoritmo de Dijkstra com fila de prioridade, na qual encontra o menor caminho possível para se chegar a saída.

3.8 Conteinerização da aplicação

A conteinerização da aplicação foi alcançada por meio do Docker e Kubernetes. O uso do Docker no projeto traz diversas vantagens significativas. Primeiramente, o Docker através do uso de contêineres permite a criação de ambientes isolados e portáteis, garantindo a consistência e padronização das configurações em diferentes ambientes de desenvolvimento e produção (VITALINO; CASTRO, 2016). Já o Kubernetes permite o gerenciamento e o provisionamento de recursos, possibilitando a rápida implantação e escalabilidade de aplicações (SANTOS, 2019).

A configuração do Docker se encontra nos arquivos **Dockerfile** e **docker-compose.yaml**. O Dockerfile é encontrado na raiz de cada projeto, tanto do *backend*, quanto *frontend*, e ele é responsável por definir toda estrutura necessária para se criar uma imagem das aplicações, sendo

Figura 3.14 – Exemplo de caminho encontrado no labirinto

1	0	0	0	0
1	1	0	0	0
0	1	1	1	0
0	0	0	1	0
0	0	0	1	1

Fonte: Autor

a base para a criação de contêineres. Em sequência, foi criado o Docker Compose, em que é utilizado como um orquestrador de contêineres. Dessa forma, ele é amplamente utilizado em ambientes de desenvolvimento para definir e gerenciar serviços externos necessários pela aplicação. Na aplicação em questão, foi configurado o RabbitMQ e o Postgres através do Docker Compose.

Por fim, a configuração do Kubernetes se encontra na pasta **k8s**, a estrutura definida foi:

- **kind.yaml**: É o arquivo de configuração do servidor do Kubernetes utilizado localmente, com o *Kind*[4] é possível criar um cluster através do Docker.
- **config-map-envs**: Arquivos responsáveis por criar as variáveis de ambiente que serão utilizados pods em execução.
- **deployments**: Arquivos responsáveis por definir a criação de pods, os pods são a menor unidade de execução dentro de um cluster. Em geral, utiliza-se um contêiner para cada Pod. Aqui é possível definir quantos pods serão executados em paralelo, limitar a memória, definir a imagem de criação, etc.
- **services**: Arquivos responsáveis por fazer o gerenciamento da rede dentro do cluster, aqui é definido portas e protocolos, nas quais os pods irão utilizar para fazer a comunicação internamente e externamente.

[4] Disponível em: <https://kind.sigs.k8s.io/>. Acesso em 24.Mai 2023

4 RESULTADOS

Neste capítulo, serão detalhados os resultados obtidos durante o processo de desenvolvimento do sistema de software proposto. Serão apresentados de forma abrangente os principais resultados alcançados, fornecendo informações relevantes sobre as funcionalidades e impactos do sistema desenvolvido.

4.1 *Backend*

Para o pleno funcionamento da aplicação do *backend* foi gerado duas aplicações descritas subsequentemente nas seções 4.1.1 e 4.1.2.

4.1.1 *API Application*

A *API Application* disponibiliza *endpoints* via protocolo HTTP para que os recursos sejam manipulados, se conecta via AMQP para enviar dados à fila de processamento e por fim se comunica com o banco de dados Postgres para persistir as informações. Os *endpoints* criados são:

- Para o *Puzzle n-queen*:
  - **[GET] /api/v1/n-queen**: Lista de forma resumida todos os *puzzles n-queens* criado. Dessa forma, é retornado campos como o número de rainhas, identificador, data de criação, status e tempo de processamento
  - **[GET] /api/v1/n-queen/:id**: Retorna todos os dados relacionados a uma *n-queen* baseado em seu identificador.
  - **[POST] /api/v1/n-queen**: Cria um novo *puzzle* do problema *n-queens*, é enviado no corpo da requisição a quantidade de rainhas desejada. Nesse *endpoint* é criado um novo registro no banco de dados com o status de processamento *PENDING* e enviado para o RabbitMQ para processamento posterior.
- Para o *Puzzle sudoku*:
  - **[GET] /api/v1/sudoku**: Lista de forma resumida todos os *puzzles sudoku* criado. Dessa forma, é retornado campos como o identificador, data de criação, status e tempo de processamento

  - **[GET] /api/v1/sudoku/:id**: Retorna todos os dados relacionados a um *sudoku* baseado em seu identificador.
  - **[POST] /api/v1/sudoku**: Cria um novo *puzzle* do problema *sudoku*, é enviado no corpo da requisição a quadro pré-preenchido. Nesse *endpoint* é criado um novo registro no banco de dados com o status de processamento *PENDING* e enviado para o RabbitMQ para processamento posterior.

- Para o *Puzzle maze*:
  - **[GET] /api/v1/maze-resolver**: Lista de forma resumida todos os *puzzles maze* criado. Dessa forma, é retornado campos como o identificador, data de criação, status e tempo de processamento
  - **[GET] /api/v1/maze-resolver/:id**: Retorna todos os dados relacionados a um *maze* baseado em seu identificador.
  - **[POST] /api/v1/maze-resolver**: Cria um novo *puzzle* do problema *maze*, é enviado no corpo da requisição a matriz representando o labirinto. Nesse *endpoint* é criado um novo registro no banco de dados com o status de processamento *PENDING* e enviado para o RabbitMQ para processamento posterior.

4.1.2 Processor

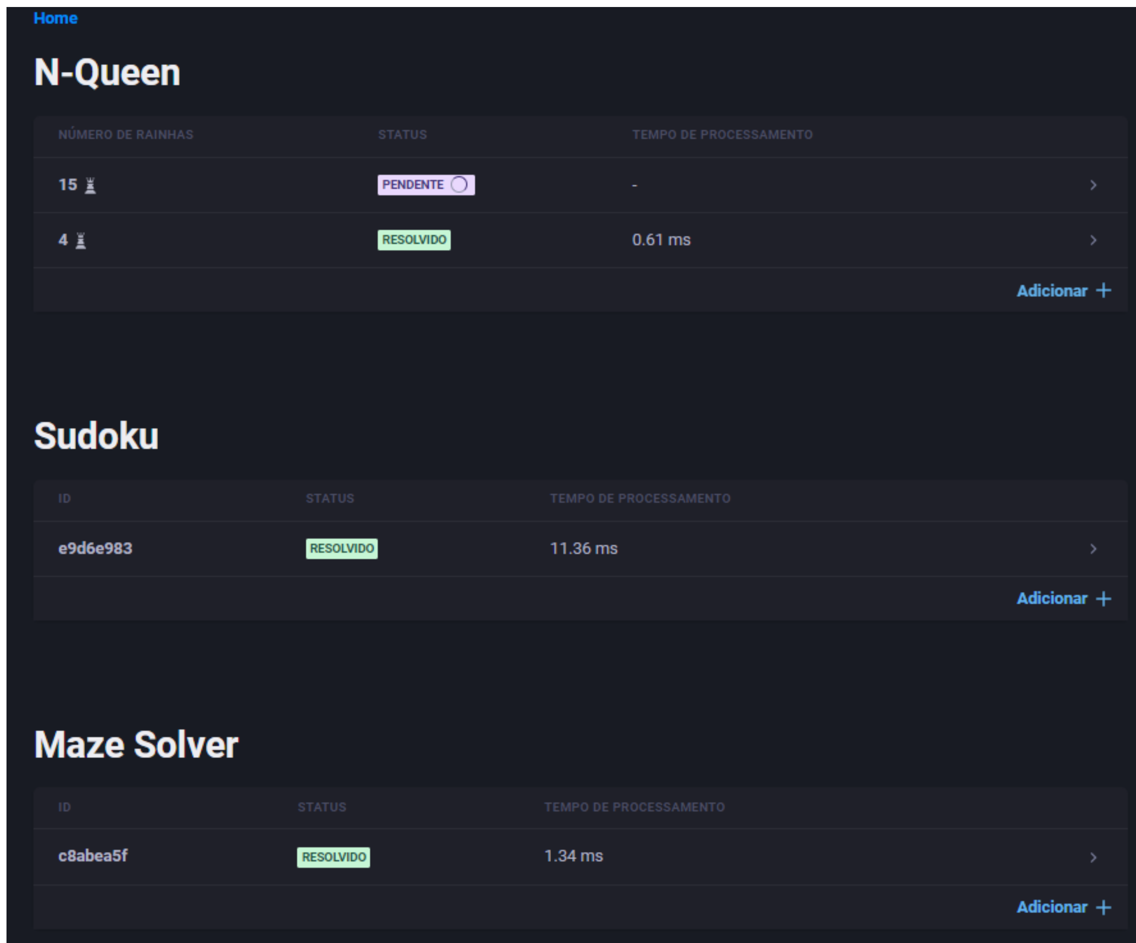
Esta aplicação tem o objetivo de se conectar via AMQP no RabbitMQ, realizar o processamento dos *puzzles* e atualizar o banco de dados. Ao se conectar à fila, a mesma recebe o ID do *puzzle* correspondente, recupera informações necessárias para o processamento na base de dados, envia os dados para os algoritmos e, ao receber o retorno dos mesmos, atualiza os dados no Postgres.

4.2 Frontend

Para viabilizar a manipulação dos recursos, optou-se pelo desenvolvimento uma aplicação web para acesso através do navegador. Como já supracitado, as tecnologias utilizadas foi o React com Next.js, a seguir será descrito as funcionalidades disponíveis para uso.

Na tela inicial, possui três tabelas, uma para cada *puzzle* desenvolvido, em que possui uma listagem resumida de todos os registros já criados. Essa tabela é atualizada periodicamente, a fim de manter sempre o usuário informado sobre o status mais atual do *puzzle*.

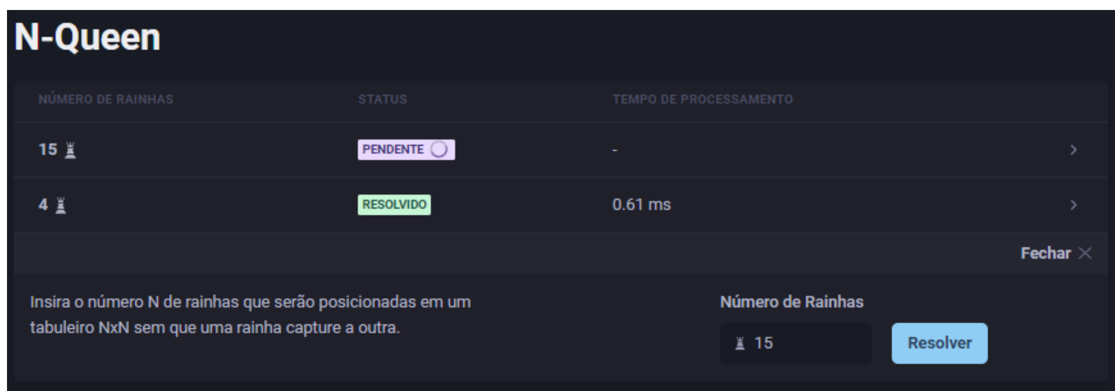
Figura 4.1 – Tela inicial



Fonte: Autor

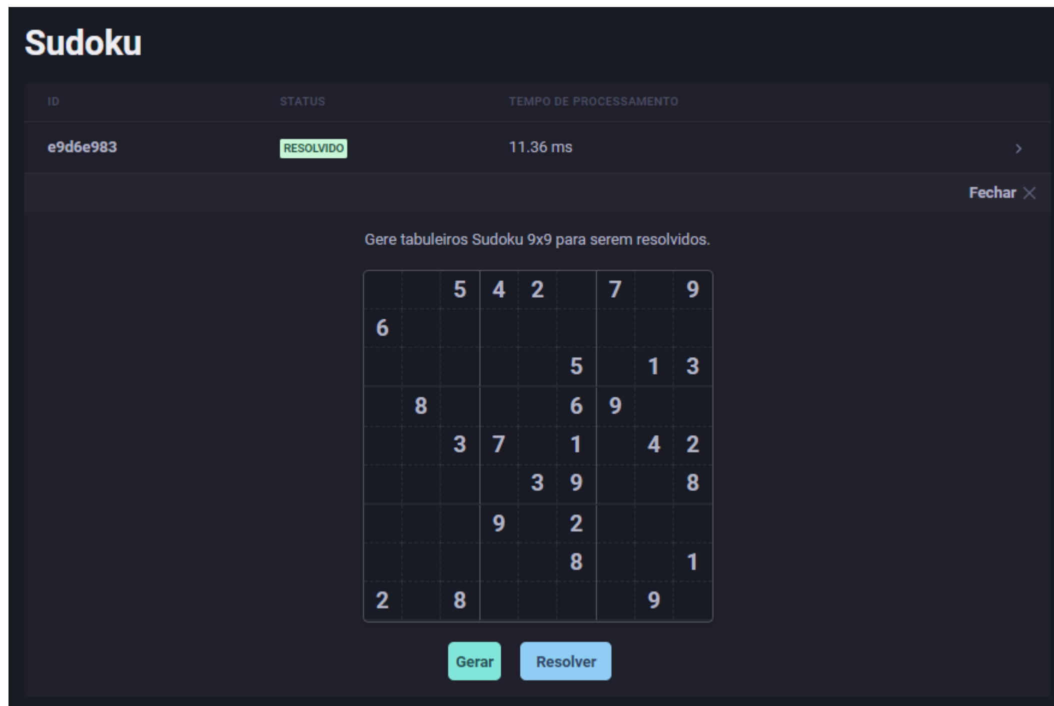
Nessa tela, também é possível criar novos registros. Ao final de cada tabela, há um botão de adicionar. Para *nQueen* o usuário deve informar a quantidade de rainhas desejada. Para os demais problemas, basta gerar os quadros que mais agradarem e instruir o sistema a resolvê-los.

Figura 4.2 – Adicionando uma *nQueen*



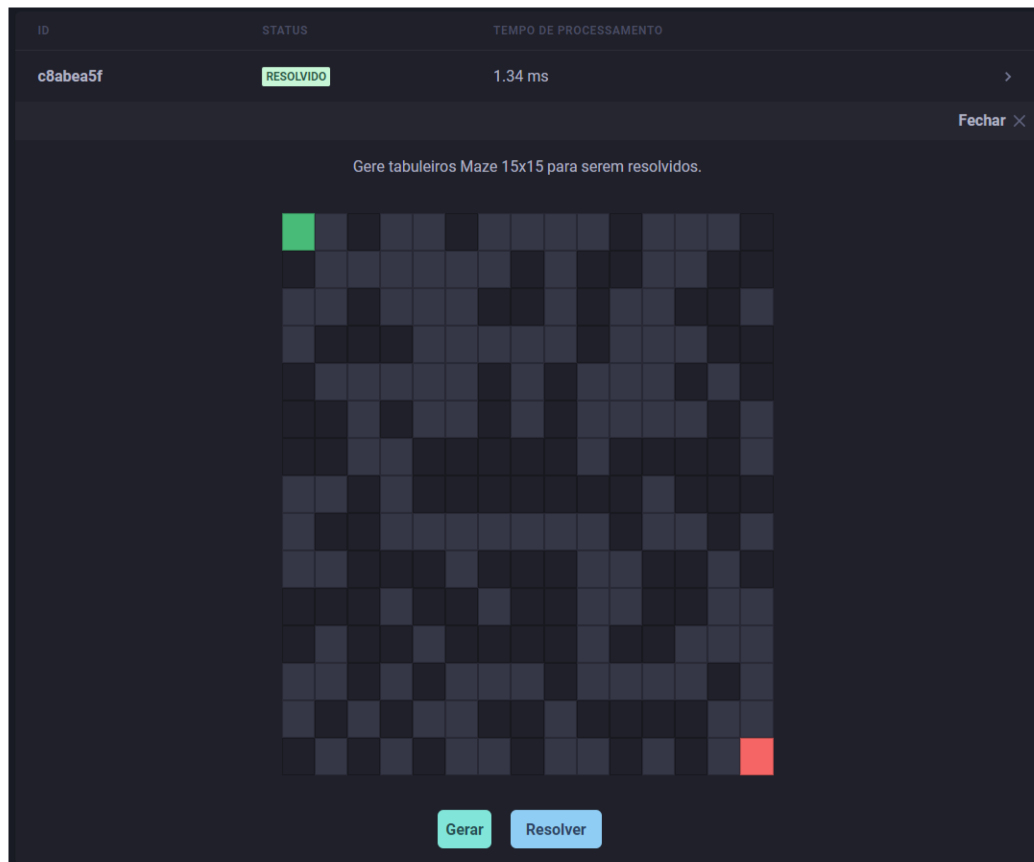
Fonte: Autor

Figura 4.3 – Adicionando um *Sudoku*



Fonte: Autor

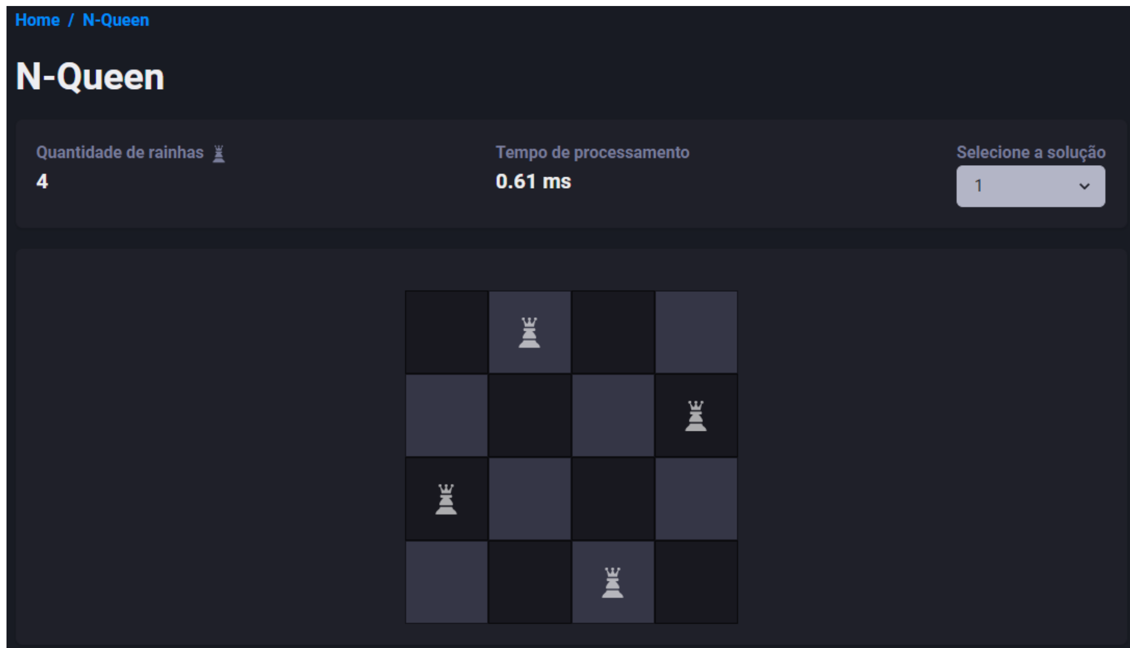
Figura 4.4 – Adicionando um *Maze*



Fonte: Autor

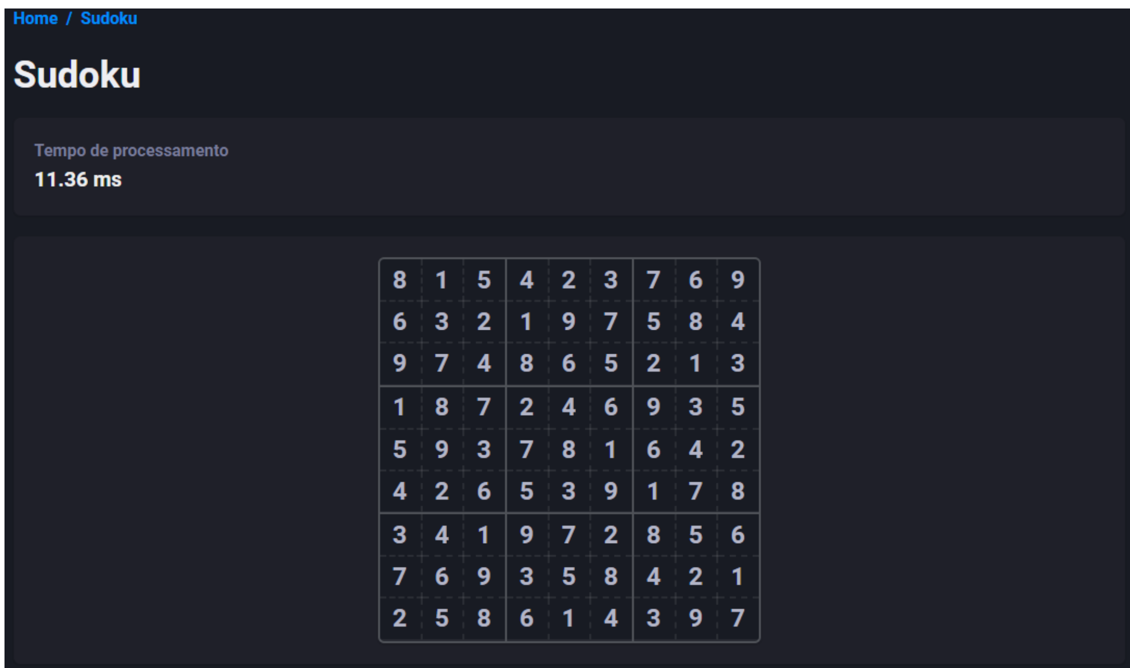
Por fim, cada *puzzle* possui uma tela de detalhes disponível, na qual são apresentados os resultados dos quebra-cabeças. Para acessar cada resultado, basta clicar na linha da tabela desejada.

Figura 4.5 – Detalhes de um *nQueen*



Fonte: Autor

Figura 4.6 – Detalhes de um *Sudoku*



Fonte: Autor

Figura 4.7 – Detalhes de um *Maze Solver*

Fonte: Autor

4.3 Infraestrutura

No âmbito deste trabalho, foi feito uso do Docker e do Docker Compose como parte do ambiente de desenvolvimento. A utilização dessas ferramentas trouxe resultados significativos. O Docker permitiu empacotar e isolar as aplicações em contêineres, facilitando a configuração e garantindo a consistência do ambiente. Com o Docker Compose, foi possível definir e executar facilmente vários serviços localmente, como o banco de dados e serviço de mensageria.

Um dos principais benefícios do uso do Docker e do Docker Compose é a facilidade de implantação do sistema em diferentes ambientes. Como por exemplo, se fosse necessário compartilhar o projeto com diferentes desenvolvedores, a configuração do ambiente não seria problema e os mesmos poderiam focar no desenvolvimento mais rapidamente.

Em sequência, outra ferramenta que desempenhou um papel fundamental foi o Kubernetes, como uma plataforma de orquestração de contêineres. Utilizado em conjunto com o Docker para gerenciar e implantar o sistema em diferentes ambientes, como homologação e produção. O Kubernetes permitiu dimensionar verticalmente e horizontalmente os contêineres, garantindo que a aplicação seja capaz de lidar com o aumento da carga de trabalho de forma eficiente. Isso se torna especialmente relevante em cenários em que o sistema precisa suportar um grande número de usuários ou processar grandes volumes de dados. Além disso, o K8s oferece

recursos avançados de gerenciamento de implantações, permitindo a execução de atualizações e rollbacks de forma automatizada e controlada.

Outro benefício importante do Kubernetes é a facilidade de implantação. Com o uso de arquivos de configuração YAML, é possível descrever toda a infraestrutura do sistema, incluindo serviços, volumes, segredos e políticas de rede. Essa abordagem permite que o sistema seja implantado em diferentes provedores de nuvem, como o *Amazon EKS*[1] ou *Google Kubernetes Engine (GKE)*[2], mantendo a consistência e garantindo a portabilidade.

Dessa forma, o Docker e Kubernetes se mostrou uma escolha acertada para este projeto, pois trouxe benefícios significativos, como a escalabilidade do sistema, o gerenciamento automatizado de implantações e a portabilidade entre diferentes ambientes de nuvem.

4.3.1 Replicabilidade

Como salientado na seção anterior, o trabalho foi pensado para facilitar a replicabilidade. Dessa forma, utilizou-se *containers* - Docker e Kubernetes. Assim, essa seção apresenta os passos para a replicação em um ambiente local para testar a *stack* apresentada neste trabalho.

Note que, para executar os passos descritos nesta subseção, deve-se ter instalado, apenas o Docker e utilizando o sistema operacional Linux.

- **Instalando o Kind (Kubernetes in Docker):**
  - curl -Lo ./kind https://kind.sigs.k8s.io/dl/v0.20.0/kind-linux-amd64
  - chmod +x ./kind
  - mv ./kind /usr/local/bin/kind
- **Instalando kubectl**
  - curl -LO https://dl.k8s.io/release/v1.21.0/bin/linux/amd64/kubectl
  - chmod +x ./kubectl
  - mv ./kubectl /usr/local/bin/kubectl
- **Clonando o projeto**
  - git clone https://github.com/jeversonjv/tcc-backend.git

---

[1] Disponível em: <https://aws.amazon.com/pt/eks/>
[2] Disponível em: <https://cloud.google.com/kubernetes-engine?hl=pt-br>. Acesso em 08.Jun 2023

  - cd tcc-backend

- **Criando o cluster Kubernetes**
  - kind create cluster –config=k8s/kind.yaml –name=tcc-backend

- **Criando os objetos do Kubernetes**
  - for file in k8s/config-map-envs/*.yaml; do kubectl apply -f "$file"; done
  - for file in k8s/deployments/*.yaml; do kubectl apply -f "$file"; done
  - for file in k8s/services/*.yaml; do kubectl apply -f "$file"; done

- **Expondo as portas dos Pods**: Depois que os Pods se iniciarem corretamente - é possível monitorar com o comando: **kubectl get pods** - rodar os comandos:
  - kubectl port-forward service/api-application-service 3000:3000
  - kubectl port-forward service/frontend-service 3002:3002

Portanto, ao final da configuração a aplicação *frontend* e *backend* estarão disponíveis na porta 3002 e 3000, respectivamente, sendo acessíveis através do protocolo HTTP.

4.4 Principais desafios enfrentados

Durante a realização deste trabalho, alguns desafios relevantes foram enfrentados. Um dos principais desafios ocorreu no contexto da implantação e gerenciamento do sistema utilizando o Kubernetes. A configuração e orquestração dos contêineres exigiram um estudo aprofundado da ferramenta, assim como a resolução de problemas relacionados à escalabilidade, disponibilidade e comunicação entre os serviços.

Além disso, a escrita acadêmica do texto também se apresentou como um desafio importante. A adequação às normas e padrões da escrita científica, a clareza na exposição das ideias, a organização coerente dos capítulos e a revisão minuciosa para garantir a precisão e a consistência das informações foram aspectos que demandaram esforço adicional e atenção aos detalhes.

Por fim, outro desafio ocorreu no contexto da estilização do frontend, em especial na manipulação e implementação do CSS. Lidar com as particularidades do CSS e garantir uma aparência visual coesa e atraente para o sistema exigiu um esforço adicional.

5 CONSIDERAÇÕES FINAIS

O presente trabalho teve como objetivo o desenvolvimento de um sistema que utiliza ferramentas de mercado para a resolução de *puzzles*, explorando conceitos e técnicas utilizadas atualmente na construção de sistemas modernos. Durante a pesquisa, foram abordadas diversas etapas, desde as revisões literárias, definição da arquitetura, escolha de linguagem e *frameworks*, modelagem do banco de dados, configuração da fila de processamento, definição de algoritmos para resolução dos *puzzles*, até a estruturação da infraestrutura com Docker e Kubernetes.

É importante destacar a vantagem do uso de linguagens de programação e *frameworks* modernos. Neste trabalho, optou-se por utilizar linguagens e *frameworks* atualizados, como o Nest.js e o Next.js, devido à sua ampla adoção e comunidade ativa. Essas tecnologias proporcionaram maior produtividade no desenvolvimento, com recursos avançados e robustos que agilizaram o processo de implementação.

Outro aspecto relevante foi a utilização do RabbitMQ como sistema de mensageria. Essa escolha se deu pelo seu desempenho, confiabilidade e capacidade de gerenciar filas de mensagens de forma eficiente. O RabbitMQ permitiu a comunicação assíncrona entre os componentes do sistema, tornando-o mais escalável e resiliente. A utilização de um sistema de mensageria como o RabbitMQ é especialmente benéfica em ambientes distribuídos, onde diferentes componentes precisam trocar informações de maneira eficiente e confiável.

Podemos concluir que a utilização do Docker e do Kubernetes trouxe inúmeros benefícios para o desenvolvimento e implantação do sistema. A facilidade de uso, a portabilidade e a escalabilidade foram aspectos chave que contribuíram para o sucesso do projeto. Além disso, as habilidades adquiridas ao longo dessa jornada, tanto em termos técnicos quanto na compreensão das melhores práticas de desenvolvimento, certamente serão valiosas para a minha carreira profissional.

5.1 Disciplinas que contribuíram para o desenvolvimento do trabalho

Durante a jornada de desenvolvimento deste trabalho, pude contar com o conhecimento adquirido em diversas disciplinas que foram fundamentais para sua realização. As disciplinas de Programação proporcionou a base sólida para a implementação dos algoritmos e a lógica de programação necessária para a criação do sistema. Já a disciplina de Redes ofereceu uma compreensão aprofundada sobre a comunicação entre os componentes do sistema e as boas práticas de segurança. O conhecimento em Banco de Dados permitiu a modelagem e o ar-

mazenamento eficiente das informações, garantindo a integridade dos dados. A disciplina de Engenharia de Software trouxe conceitos importantes para a organização do projeto, como a definição de requisitos, planejamento e gerenciamento do desenvolvimento. Por fim, o estudo de Grafos contribuiu para a aplicação de algoritmos e estruturas de dados eficientes na resolução de problemas complexos relacionados ao projeto. O conjunto de conhecimentos adquiridos nessas disciplinas foi essencial para a realização bem-sucedida do TCC, fornecendo uma base sólida para a compreensão dos desafios enfrentados e para a aplicação das melhores práticas no desenvolvimento do sistema.

5.2 Trabalhos futuros

Como recomendações para trabalhos futuros, sugere-se a exploração de outras ferramentas e técnicas relacionadas ao desenvolvimento de sistemas modernos, como a utilização de serviços em nuvem, implementação de testes automatizados e adição de ferramentas de observabilidade. Além disso, a realização de estudos comparativos entre diferentes plataformas de orquestração, como o Kubernetes e o Docker Swarm, poderia fornecer ideias valiosas para a escolha da melhor ferramenta em diferentes cenários.

Em relação aos algoritmos utilizados neste trabalho, os mesmos apresentaram resultados satisfatórios na resolução dos *puzzles* propostos, mas é importante ressaltar que existem limitações associadas a esses algoritmos. Alguns *puzzles* de maior complexidade ou com características específicas podem apresentar desafios adicionais e exigir abordagens diferentes, como no caso de *nQueen* com valores altos. Sendo assim, como sugestão para trabalhos futuros, seria interessante explorar e aprimorar os algoritmos utilizados, bem como a adição de novos *puzzles* ao sistema. Essa abordagem permitiria ampliar as capacidades do sistema, tornando-o capaz de lidar com uma variedade ainda maior de desafios.

Além disso, seria válido investigar a utilização de técnicas de inteligência artificial, como algoritmos de aprendizado de máquina e redes neurais, para aprimorar a capacidade de resolução dos *puzzles*. Essas técnicas podem permitir uma abordagem mais sofisticada e adaptativa, que se ajuste dinamicamente às características de cada puzzle, levando a soluções ainda mais eficientes e precisas.

REFERÊNCIAS

CHANG, S. K. **Data Structures and Algorithms**. [S.l.]: World Scientific, 2003. ISBN 9789812791245, 9812791248.

DATE, C. J. **Introdução a Sistemas de Bancos de Dados**. [S.l.]: GEN LTC, 2004. ISBN 8535212736.

FLANAGAN, D. **JavaScript: O Guia Definitivo**. [S.l.]: Bookman Editora, 2012. ISBN 9788565837484, 8565837483.

FOWLER, S. J. **Microsserviços prontos para a produção**. [S.l.]: Novatec, 2019. ISBN 9788575227473, 8575227475.

GOLDBARG, E. **GRAFOS: conceitos, algoritmos e aplicações**. [S.l.]: Elsevier, 2012. ISBN 8535257187, 9788535257182.

MARTIN, R. C. **Arquitetura Limpa**. [S.l.]: Alta Books, 2019. ISBN 9788550808161, 8550808164.

NEWMAN, S. **Migrando sistemas monolíticos para microsserviços**. [S.l.]: Novatec, 2020. ISBN 9786586057041.

PEREIRA, C. R. **Node.js Aplicaçãoes web real-time com Node.js**. [S.l.]: Casa do Código, 2014.

PRESSMAN, R. S. **Engenharia de Software**. [S.l.]: AMGH, 2011. ISBN 9788563308337.

ROY, G. M. **RabbitMQ in Depth**. [S.l.]: Manning, 2017. ISBN 9781638353225.

SANTOS, L. **Kubernetes - Tudo sobre orquestração de contêineres**. [S.l.]: Casa do Código, 2019. ISBN 9788572540254.

VITALINO, J. F. N.; CASTRO, M. A. N. **Descomplicando o Docker**. [S.l.]: Brasport, 2016. ISBN 9788574527970.

XU, A. **System Design Interview - An Insider's Guide · Volume 1**. [S.l.]: Independently Published, 2020. ISBN 9798664653403.