

**Ânima Books Ltda**

**Aluna:** Alice Gomes Nascimento

**RA:** 1272328052

**Aluno:** Dinaldo Ribeiro de Carvalho

**RA:** 1272329374

**Aluno:** Jonathan Cortopassi

**RA:** 12723213891

**Aluno:** Rogério Carmo Rodrigues

**RA:** 1272316474

**Aluno:** Samuel Pereira de Souza

**RA:** 132723112947

Salvador-Bahia

2025

**Relatório**

**Introdução**

Este relatório apresenta os pontos relevantes no desenvolvimento do sistema da Ânima Books, proposta como parte da avaliação A3 da Unidade Curricular Sistemas Distribuídos e Mobile. A atividade exigiu a criação de um sistema que simulasse a captação de dados de vendas em uma rede de lojas, sendo livre a escolha do tema. A equipe optou por desenvolver uma livraria chama Ânima Books Ltda, unindo o interesse comum de seus membros pelo universo da leitura com a oportunidade de aplicar os conceitos fundamentais de sistemas distribuídos em um cenário realista e funcional. A solução desenvolvida visa atender aos requisitos definidos pelo professor, utilizando tecnologias modernas e boas práticas de engenharia de software.

A arquitetura do sistema foi baseada no padrão MVC (Model-View-Controller), com a adição das camadas Service e Repository, promovendo organização, modularidade e separação clara de responsabilidades. Atendendo a uma exigência específica do professor, o Repository foi implementado com SQL puro, mesmo com a utilização do Sequelize na camada de Models. Essa decisão trouxe um desafio técnico adicional, mas também permitiu maior controle sobre as consultas e um aprofundamento no uso da linguagem SQL na programação. O sistema foi dividido em duas aplicações independentes: a principal, voltada para o gerenciamento de clientes, vendedores, produtos, vendas e estoque; e a aplicação de relatórios, responsável por gerar estatísticas como produtos mais vendidos, consumo médio dos clientes e estoque crítico.

Ambas as aplicações foram containerizadas com Docker, como requisito obrigatório, mas que de qualquer forma, garantiu ambientes isolados e consistentes para desenvolvimento, testes e deploy. A separação entre os serviços e o uso de containers reforçam o caráter distribuído da solução, conforme requerido. O projeto da Ânima Books não apenas cumpriu os objetivos técnicos e pedagógicos da avaliação, mas também proporcionou à equipe uma experiência prática alinhada às exigências do mercado, combinando teoria, criatividade e profundo aprendizado das ferramentas utilizadas no desenvolvimento de sistemas modernos e escaláveis.

**Fundamentação Teórica**

**Arquitetura de Software**

**Conceito e Importância na Engenharia de Software**

A arquitetura de sistemas ocupa uma posição central dentro da engenharia de software, uma vez que se preocupa diretamente com as características do sistema, os elementos necessários para seu funcionamento e a forma como esses elementos se inter-relacionam. Enquanto a engenharia de software abrange todo o ciclo de desenvolvimento — desde a concepção até o gerenciamento e entrega do produto —, a arquitetura de software atua como uma disciplina interna, essencial para garantir que o sistema seja construído de forma organizada, escalável e sustentável.

Embora, por vezes, a arquitetura de software seja comparada à arquitetura civil, Pressman e Maxim (2016, p. 253) fazem questão de diferenciar essas áreas, destacando que, na engenharia de software, a arquitetura não é uma atividade independente, mas sim um subconjunto fundamental dentro do próprio processo de desenvolvimento.

**Definições Fundamentais**

Segundo Pressman e Maxim (2016), “a arquitetura de software de um programa ou sistema computacional é a estrutura ou as estruturas do sistema que abrange os componentes de software, as propriedades externamente visíveis desses componentes e as relações entre eles”. Essa definição mostra que a arquitetura oferece não apenas uma visão estrutural do sistema, mas também descreve como seus componentes interagem e se comportam, servindo como um guia para toda a equipe técnica envolvida no projeto.

Gallotti (2016) reforça essa compreensão ao afirmar que a arquitetura de software está diretamente relacionada à estrutura interna do sistema, elucidando como ele se organiza, opera e de que maneira será implementado.

**Funções Estratégicas da Arquitetura**

O papel da arquitetura de software transcende a simples organização de componentes. Pressman (2011) aponta três aspectos fundamentais:

* **Fornece uma representação abstrata** que favorece a comunicação entre desenvolvedores, gerentes, analistas e clientes.
* **Destaque para decisões de projeto** com impacto direto na engenharia subsequente.
* **Modelo intelectualmente compreensível**, suficientemente pequeno para representar como o sistema será estruturado.

Sommerville (2007) complementa que a arquitetura de software é o **elo entre a engenharia de requisitos e o desenvolvimento**. O projeto arquitetural é o primeiro passo efetivo no desenvolvimento de software, pois identifica os principais componentes estruturais e suas interações. A definição da arquitetura influencia diretamente no desempenho, na robustez, na manutenibilidade e na capacidade de evolução do sistema.

**Benefícios e Vantagens Arquiteturais**

Segundo Sommerville (2007), três grandes vantagens justificam a adoção de um projeto arquitetural bem definido:

* **Facilita a comunicação** entre os stakeholders por meio de uma visão de alto nível do sistema;
* **Permite análise antecipada**, identificando e solucionando questões técnicas desde as fases iniciais do projeto;
* **Favorece o reuso de componentes**, especialmente em sistemas com requisitos semelhantes.

Além disso, a definição arquitetural serve como **documentação técnica**, que explicita os componentes, suas propriedades e as formas como se relacionam, garantindo que o desenvolvimento siga uma direção clara e consistente.

**Padrões Arquiteturais**

Dentro desse contexto, os **padrões arquiteturais** surgem como instrumentos indispensáveis para orientar a construção de sistemas. Eles **normalizam o desenvolvimento**, promovem boas práticas, facilitam a comunicação entre os desenvolvedores e aumentam consideravelmente a produtividade.

Como destaca Martin (2017), um dos principais objetivos dos padrões arquiteturais é reduzir a necessidade de recursos humanos na construção e manutenção de sistemas, por meio de soluções previamente conhecidas e testadas para problemas recorrentes.

A adoção de padrões, porém, exige **conhecimento, experiência e análise cuidadosa** do projeto, da equipe e dos recursos. Quando bem aplicados, os padrões proporcionam:

* Aumento da produtividade
* Redução da complexidade
* Padronização da estrutura do software
* Facilidade de manutenção
* Criação de um vocabulário comum
* Reutilização de módulos
* Construção de sistemas mais robustos e confiáveis

**O Padrão Arquitetural MVC (Model–View–Controller)**

Entre os diversos padrões existentes, destaca-se o **MVC (Model–View–Controller)**, amplamente utilizado no desenvolvimento de aplicações que exigem uma separação clara entre lógica de negócio, interface e controle.

Criado em 1979 por **Trygve Reenskaug**, o padrão MVC visa separar as responsabilidades do software em três camadas:

* **Modelo (Model)**: responsável pelos dados e pelas regras de negócio.
* **Visão (View)**: encarregada da apresentação e exibição das informações ao usuário.
* **Controlador (Controller)**: age como mediador, recebendo as entradas do usuário e acionando operações no modelo.

**Gamma et al. (1994)** explicam que, dentro desse modelo, o fluxo funciona da seguinte forma: o usuário interage com a **visão**, que aciona o **controlador**; este interpreta as ações e as traduz em comandos para o **modelo**, que manipula os dados e devolve o resultado à **visão**.

**Vantagens do Padrão MVC:**

* **Redução do acoplamento** entre as partes do sistema, permitindo evolução independente das camadas;
* **Facilidade de alterações na interface gráfica**, sem impacto nas regras de negócio;
* **Reutilização de modelos** em diferentes interfaces;
* **Organização clara do fluxo da aplicação**, favorecendo testes e manutenção.

A adoção do MVC proporciona uma série de **vantagens práticas** no desenvolvimento de sistemas, especialmente no que diz respeito à organização, à escalabilidade e à manutenibilidade. Além disso, por ser um padrão amplamente difundido, desenvolvedores que o utilizam compartilham **uma linguagem comum**, o que facilita a integração de novos membros às equipes.

**Sistemas Distribuídos**

**Conceito**

Um **sistema distribuído** é aquele no qual os componentes de hardware ou software, localizados em computadores interligados por rede, comunicam-se e coordenam suas ações exclusivamente por meio do envio de mensagens. Essa definição abrange uma ampla variedade de sistemas em que a distribuição dos componentes é útil, como destaca Coulouris et al. (2005, p. 18). Os computadores podem estar em diferentes locais geográficos, desde a mesma sala até continentes distintos.

**Características**

**a) Compartilhamento de Recursos**

A principal característica dos sistemas distribuídos é permitir que diversos recursos — como impressoras, arquivos, bancos de dados, fluxos de vídeo ou chamadas — sejam compartilhados de maneira eficiente e controlada.

**b) Abstração e Transparência**

Os usuários não percebem que os aplicativos estão sendo executados em diferentes máquinas. Essa transparência é alcançada por meio da **abstração** de localização, acesso, concorrência, migração, falhas e replicação.

**c) Heterogeneidade**

Um sistema distribuído pode envolver diferentes arquiteturas de hardware, sistemas operacionais e linguagens. Segundo Tanenbaum e Van Steen (2007), o uso de **middleware** é essencial para fornecer uma camada de interoperabilidade entre esses componentes heterogêneos.

**d) Abertura**

Sistemas distribuídos devem ser **abertos**, ou seja, seguir padrões definidos por organizações reguladoras. É necessário que os serviços tenham suas **interfaces bem especificadas**, utilizando linguagens como a IDL (Interface Definition Language), que define sintaticamente a comunicação entre os componentes.

**e) Modularidade**

Para alcançar flexibilidade, Tanenbaum e Van Steen (2007) sugerem fragmentar o sistema em pequenos módulos adaptáveis. A customização de parâmetros internos (como tempo de atualização de cache) contribui para maior controle e eficiência.

**f) Escalabilidade**

Um sistema distribuído deve ser escalável em três dimensões:

* **Recursos**: adicionar mais computadores ou dispositivos.
* **Geografia**: expandir-se em termos físicos.
* **Administração**: manter a gerenciabilidade mesmo com o crescimento organizacional.

A escalabilidade simultânea em todas essas dimensões é um grande desafio, especialmente com a explosão de dados e a crescente demanda por inteligência artificial. Abadi et al. (2016) destacam que sistemas distribuídos são fundamentais para o processamento de grandes volumes de dados e para o avanço do aprendizado de máquina.

**Vantagens**

* Acesso remoto a recursos;
* Melhor aproveitamento de hardware ocioso;
* Tolerância a falhas localizadas;
* Expansão modular e econômica;
* Apoio à computação de alto desempenho (Big Data, IA).

Sistemas distribuídos são fundamentais na infraestrutura da computação moderna. Sua complexidade exige um equilíbrio entre desempenho, abstração, flexibilidade e padronização. Apesar dos desafios, sobretudo na manutenção da escalabilidade e na heterogeneidade, esses sistemas representam a base de tecnologias críticas atuais — como serviços em nuvem, redes sociais, plataformas de streaming e aplicações de inteligência artificial. O contínuo avanço dessas áreas torna os sistemas distribuídos não apenas relevantes, mas indispensáveis.

**Node.js**

**Conceito e Definições Fundamentais**

Node.js é uma plataforma de código aberto que permite a execução de JavaScript no lado do servidor, oferecendo uma abordagem eficiente e moderna para o desenvolvimento de aplicações web e sistemas distribuídos. Criado em 2009 por Ryan Dahl, o Node.js surgiu com o propósito de superar as limitações dos servidores tradicionais, que utilizavam modelos síncronos e sujeitos a bloqueios em operações de entrada e saída (I/O).

Seu funcionamento é baseado em um modelo assíncrono e orientado a eventos, permitindo que múltiplas requisições sejam tratadas simultaneamente sem a necessidade de criar múltiplos processos ou threads. Esse modelo garante maior desempenho e escalabilidade, especialmente em aplicações que demandam alto volume de conexões simultâneas.

O Node.js utiliza o motor V8, desenvolvido pela Google, responsável por compilar o código JavaScript diretamente em linguagem de máquina, o que contribui significativamente para sua alta performance. A partir dele, tornou-se possível utilizar JavaScript tanto no desenvolvimento de aplicações front-end quanto back-end, promovendo maior integração entre as camadas da aplicação e simplificando o desenvolvimento.

**Recursos Principais**

Entre seus principais recursos, Node.js disponibiliza uma biblioteca padrão composta por diversos módulos nativos que facilitam o desenvolvimento de aplicações. Dentre os principais, destacam-se:

* **HTTP**: Permite a criação de servidores web, bem como o gerenciamento de requisições e respostas HTTP.
* **FS (File System)**: Oferece métodos para manipulação de arquivos, tanto de forma síncrona quanto assíncrona.
* **Path**: Auxilia nas operações relacionadas a caminhos de arquivos e diretórios.
* **EventEmitter**: Proporciona a criação e o gerenciamento de eventos personalizados.
* **Crypto**: Disponibiliza funções para realização de operações de criptografia.

Além dos módulos nativos, destaca-se o uso do **Node Package Manager (npm)**, que representa o maior ecossistema de bibliotecas de software do mundo. Por meio do **npm**, desenvolvedores podem instalar, atualizar e compartilhar pacotes, promovendo a agilidade no desenvolvimento e na resolução de desafios técnicos.

**Benefícios do Node.js**

O Node.js apresenta um conjunto de vantagens que justificam sua ampla adoção no mercado tecnológico, entre as quais se destacam:

* **Alta Performance**: Graças ao motor V8 e ao modelo de execução orientado a eventos, oferece elevado desempenho e baixa latência.
* **Escalabilidade**: Sua arquitetura permite a gestão eficiente de milhares de conexões simultâneas, sendo altamente escalável horizontalmente.
* **Unificação de Linguagem**: A utilização do JavaScript tanto no front-end quanto no back-end simplifica o desenvolvimento e promove maior consistência no código.
* **Ampla Comunidade e Ecossistema**: A expressiva comunidade global contribui com soluções, bibliotecas e suporte contínuo.
* **Eficiência no Uso de Recursos**: A ausência de múltiplas threads reduz significativamente o consumo de memória e processamento, tornando-o ideal para ambientes de alta concorrência.

**Aplicações Típicas**

O Node.js é amplamente utilizado no desenvolvimento de:

* **Servidores Web e APIs:** Ideal para desenvolvimento de APIs REST, GraphQL e serviços de backend escaláveis.
* **Aplicações em Tempo Real:** Plataformas de comunicação, jogos multiplayer e sistemas de streaming se beneficiam da baixa latência proporcionada.
* **Processamento de Dados em Larga Escala:** Ferramentas que lidam com grandes volumes de dados, como sistemas de análise e monitoramento.
* **Soluções para Internet das Coisas (IoT):** Sua leveza e baixo consumo de recursos o tornam adequado para dispositivos conectados e sistemas embarcados.

Node.js consolidou-se como uma tecnologia robusta e eficiente, especialmente para aplicações que exigem escalabilidade, rapidez e desenvolvimento ágil. Seu modelo assíncrono e a utilização do JavaScript no servidor proporcionam vantagens que atendem às demandas atuais do desenvolvimento de software.

**Banco de Dados Relacionais**

**Conceito e Definições Importantes**

Os bancos de dados relacionais representam um dos modelos mais utilizados na gestão e organização de dados no contexto tecnológico atual. Fundamentados na lógica de relações matemáticas, esses bancos estruturam os dados em tabelas compostas por linhas e colunas. Cada linha corresponde a um registro único, enquanto cada coluna define um atributo específico, como nome, idade ou identificação.

O modelo relacional foi desenvolvido por Edgar F. Codd, em 1970, com o objetivo de superar limitações de modelos anteriores, oferecendo uma abordagem mais flexível e consistente na organização de dados. Sua principal característica está na definição de relações entre tabelas, estabelecidas por meio de **chaves primárias** e **chaves estrangeiras**, que garantem a integridade e a coerência das informações. Por exemplo, em um sistema de gestão de vendas, a tabela de clientes pode se relacionar diretamente com a tabela de pedidos, criando uma conexão lógica entre ambas.

Os bancos relacionais utilizam a **Structured Query Language (SQL)**, uma linguagem padrão para definição, manipulação e consulta de dados. Por meio dela, é possível executar comandos que permitem criar tabelas (DDL), inserir, alterar e excluir registros (DML), além de realizar consultas sofisticadas e gerar relatórios.

**Funcionamento e Estrutura**

A organização dos dados ocorre através de tabelas interconectadas, que seguem um esquema previamente definido. Este esquema especifica não apenas os atributos, mas também as relações existentes entre as tabelas, permitindo uma estrutura lógica e eficiente. Um conceito fundamental nesse processo é a **normalização**, que busca eliminar redundâncias e preservar a consistência dos dados.

As relações são estabelecidas utilizando **chaves primárias**, que identificam unicamente cada registro, e **chaves estrangeiras**, que fazem referência à chave primária de outra tabela. Isso possibilita a construção de bancos robustos e com forte integridade referencial, facilitando tanto a organização quanto o cruzamento de informações.

**Propriedades ACID**

Os bancos de dados relacionais se destacam por garantir as propriedades **ACID**, fundamentais para a segurança e a confiabilidade das transações:

* **Atomicidade**: assegura que uma transação seja executada por completo ou, em caso de falha, não seja realizada parcialmente.
* **Consistência**: garante que o banco de dados esteja sempre em um estado válido antes e após cada transação.
* **Isolamento**: impede que transações simultâneas causem interferências entre si.
* **Durabilidade**: assegura que, uma vez concluída, a transação permaneça registrada, mesmo diante de falhas no sistema.

Esses princípios são essenciais para assegurar a integridade dos dados, especialmente em sistemas que lidam com grandes volumes de transações simultâneas.

**Aplicações e Benefícios**

Os bancos de dados relacionais são amplamente utilizados em sistemas que exigem precisão, integridade e segurança no gerenciamento de dados. Estão presentes em setores como comércio eletrônico, instituições financeiras, sistemas de saúde, gestão empresarial e diversas outras áreas.

Ferramentas como o **MySQL**, um dos sistemas de gerenciamento de banco de dados relacional (SGBDR) mais populares, exemplificam a aplicação prática desse modelo. De código aberto, o MySQL oferece suporte a diversas linguagens de programação, além de ser escalável, seguro e adequado tanto para pequenas aplicações quanto para sistemas corporativos de grande porte. Sua utilização se estende a ambientes locais, plataformas na nuvem e aplicações web, graças aos seus recursos como procedimentos armazenados, gatilhos, criptografia de dados e controle de acesso.

Os SGBDRs, de maneira geral, oferecem uma interface robusta que permite aos usuários e sistemas acessarem, manipularem e analisarem dados de maneira eficiente. Além disso, proporcionam suporte para análise em tempo real, essencial para tomadas de decisão estratégicas.

Os bancos de dados relacionais são fundamentais na gestão de dados estruturados, permitindo uma organização lógica, segura e eficiente. Com base na utilização de tabelas inter-relacionadas e nas propriedades ACID, esses sistemas garantem integridade, consistência e escalabilidade. A adoção de bancos de dados relacionais continua sendo uma escolha preferencial em diversos setores, atendendo às demandas de armazenamento, manipulação e análise de dados de forma confiável e eficaz.

**Containers e Docker**

**Conceito de Containers**

Containers são uma forma de virtualização leve que permite executar aplicações e seus ambientes de forma isolada sobre o mesmo kernel do sistema operacional. Diferentemente das máquinas virtuais, que exigem um sistema operacional completo em cada instância, os containers compartilham o kernel do sistema anfitrião, tornando-os mais leves, rápidos e eficientes.

Ao contrário da virtualização do hardware, os containers virtualizam o sistema operacional — geralmente Linux — permitindo que cada container inclua apenas a aplicação, juntamente com suas bibliotecas, arquivos de configuração e dependências necessárias. Essa abordagem elimina a necessidade de um sistema operacional convidado, o que reduz drasticamente o consumo de recursos e melhora a portabilidade entre ambientes.

**Características Principais dos Containers**

* **Isolamento:** Cada container opera de forma independente, com seu próprio sistema de arquivos, variáveis de ambiente, rede e processos, o que garante segurança e previsibilidade na execução das aplicações.
* **Portabilidade:** Como os containers incluem todas as dependências da aplicação, eles podem ser executados de forma consistente em qualquer ambiente compatível com a tecnologia de containerização, seja em máquinas locais, servidores remotos ou na nuvem.
* **Eficiência:** Compartilhando o kernel do sistema operacional, os containers consomem menos recursos que as máquinas virtuais, inicializam rapidamente e são ideais para sistemas distribuídos e escaláveis.

**Fundamentos Técnicos**

A tecnologia de containers é sustentada por três pilares principais do kernel Linux:

* **Namespaces:** Garantem o isolamento de recursos, como processos, IDs de usuários, sistemas de arquivos, dispositivos e rede, entre os containers e o sistema anfitrião.
* **Control Groups (cgroups):** Controlam, monitoram e limitam o uso de recursos de hardware (CPU, memória, I/O, etc.) por cada container, prevenindo sobrecargas no sistema.
* **Union File Systems (UnionFS):** Permitem a sobreposição de camadas de arquivos, o que facilita a criação de containers leves e modulares, com partes compartilhadas entre múltiplas instâncias.

**Docker: Um Facilitador de Containers**

Docker é uma plataforma amplamente utilizada que automatiza a criação, distribuição e execução de containers. Introduzido em 2013, o Docker popularizou a tecnologia de containers ao oferecer uma interface simples e padronizada para desenvolvedores e equipes de operações.

**Componentes do Docker**

* **Docker Engine:** O serviço que gerencia containers no sistema operacional, sendo responsável por criar, iniciar e parar containers.
* **Docker Images:** Pacotes imutáveis que contêm tudo o que uma aplicação precisa para rodar — código, dependências, bibliotecas, variáveis de ambiente e configurações.
* **Docker Containers:** Instâncias em execução baseadas nas imagens, isoladas e com comportamento previsível.
* **Docker Hub:** Um repositório público e centralizado para armazenar e compartilhar imagens Docker, promovendo reutilização e colaboração entre desenvolvedores.

**Vantagens do Docker**

* **Consistência:** A aplicação roda da mesma forma em qualquer ambiente, seja local, em servidores ou na nuvem.
* **Ecossistema robusto:** Ferramentas para orquestração (como Kubernetes), monitoramento, versionamento de imagens, segurança e integração contínua.
* **Integração com DevOps:** Acelera pipelines de CI/CD, facilita testes, liberações e implantações contínuas, promovendo alinhamento entre equipes de desenvolvimento e operações.

A adoção de containers e ferramentas como Docker representa um avanço significativo no desenvolvimento moderno de software. Eles promovem **eficiência, portabilidade e escalabilidade**, simplificando o ciclo de vida das aplicações — do desenvolvimento à produção. Empresas de diferentes setores utilizam containers para reduzir custos, acelerar lançamentos e melhorar o desempenho operacional.

Com base em tecnologias sólidas de sistemas operacionais e com forte integração ao ecossistema DevOps, os containers se consolidaram como uma solução essencial para ambientes distribuídos e aplicações modernas. O entendimento conceitual e técnico dessas ferramentas é crucial para sua adoção estratégica e bem-sucedida em ambientes corporativos e acadêmicos.

**Projeto Implementação**

O presente projeto trata-se de uma livraria “Ânima Books Ltda” e foi estruturado em uma pasta central chamada **código-fonte**, que está subdividida em duas pastas principais. A primeira, denominada **report-engine**, concentra todos os arquivos e recursos relacionados à aplicação principal. Nessa pasta, encontram-se as *migrations* responsáveis por criar a estrutura inicial do banco de dados, incluindo o cadastro mínimo de 10 (dez) produtos, 5 (cinco) clientes e 2 (dois) vendedores, conforme especificado nos requisitos.

Além disso, a pasta **src** dentro do *report-engine* abriga a organização das rotas (*routes*), controladores (*controllers*), serviços (*services*), repositórios (*repositories*) e modelos (*models*) da aplicação. Assim, a pasta **report-engine** representa a aplicação principal, cujo tema escolhido pela equipe foi o gerenciamento de uma rede de venda de livros.

A segunda pasta principal, chamada **store-manager**, é dedicada à aplicação responsável pela geração dos relatórios estatísticos. Esta aplicação funciona de forma independente, acessando os dados da aplicação principal para produzir os relatórios exigidos, como produtos mais vendidos, consumo médio por cliente e produtos com baixo estoque. Segue a estrutura geral:

/ codigo-fonte

├── report-engine/

│ ├── src/

│ │ ├── controllers/

│ │ ├── routes/

│ │ ├── services/

│ │ ├── repositories/

│ │ ├── models/

│ │ ├── migrations/

│ │ └── config/

│ ├── package.json

│ └── ... (outros arquivos da aplicação principal)

│

├── store-manager/

│ ├── src/

│ │ ├── controllers/

│ │ ├── routes/

│ │ ├── services/

│ │ └── repositories/

│ ├── package.json

│ └── ... (outros arquivos da aplicação de relatórios)

│

└── docker-compose.yml

A pasta models, localizada em report-engine/src, agrupa todos os arquivos responsáveis por definir as entidades do sistema da Ânima Boks utilizando a ORM Sequelize. Cada arquivo representa uma tabela no banco de dados, com seus respectivos campos, tipos e associações:

* Customer.js: define os dados do cliente.
* Product.js: define os produtos disponíveis para venda.
* Seller.js: representa os vendedores cadastrados no sistema.
* Sale.js: armazena informações sobre cada venda realizada.
* SaleItem.js: entidade associativa que detalha os itens vendidos em cada venda, permitindo o relacionamento entre produtos e vendas.
* index.js: realiza a integração entre os modelos e estabelece as associações entre eles.

Essa organização facilita a manutenção do código e garante a clareza na definição do esquema relacional da aplicação. Segue estrutura dos models:

/codigo-fonte

├── report-engine/

│ ├── src/

│ │ ├── models/

│ │ │ ├── index.js

│ │ │ ├── Customer.js

│ │ │ ├── Product.js

│ │ │ ├── Sale.js

│ │ │ ├── SaleItem.js

│ │ │ └── Seller.js

A seguir, apresentamos também o **Diagrama Entidade-Relacionamento (DER)**, que representa a estrutura lógica do banco de dados. Nele, é possível visualizar de forma clara as **chaves primárias**, que identificam unicamente cada registro em suas respectivas tabelas, bem como as **chaves estrangeiras**, que estabelecem os relacionamentos entre as entidades. O DER oferece uma visão precisa e detalhada de como os dados estão organizados e interligados no sistema, sendo fundamental para garantir a integridade e a consistência das informações armazenadas.

Interface gráfica do usuário

O conteúdo gerado por IA pode estar incorreto.

Para facilitar a compreensão da estrutura do sistema desenvolvido, elaboramos o Diagrama de Classes, que representa as principais entidades do projeto e seus relacionamentos. Neste diagrama, é possível visualizar claramente as **chaves primárias** de cada classe, que identificam unicamente cada registro, bem como as **chaves estrangeiras**, que estabelecem as ligações entre as entidades.

Além disso, algumas funcionalidades importantes foram implementadas como **métodos** dentro das classes correspondentes. A classe **Product**, por exemplo, possui o método **stock**, que calcula a quantidade disponível em estoque subtraindo o total vendido da quantidade inicialmente cadastrada. Assim como, havendo um cancelamento de venda soma-se o produto ao estoque. Isso permite um controle dinâmico e preciso do estoque real disponível.

Já a classe **SaleItem** inclui o método **total\_value**, responsável por calcular o valor total de itens vendido, multiplicando a quantidade vendida de cada item pelo seu respectivo preço unitário.

A presença desses métodos no diagrama evidencia a responsabilidade de cada entidade não apenas pelo armazenamento de dados, mas também pela execução da lógica de negócio relacionada, tornando o sistema mais organizado e facilitando a manutenção.

A identificação explícita das chaves e desses métodos é fundamental para garantir a integridade dos dados e o correto funcionamento das funcionalidades do sistema.

A seguir, apresentamos o diagrama de classes elaborado para o projeto.

Diagrama

O conteúdo gerado por IA pode estar incorreto.

É importante destacar que, conforme exigência do projeto, a camada de **repositório (Repository)** foi implementada utilizando **SQL puro** para a realização de operações com o banco de dados. Embora a **ORM Sequelize** tenha sido utilizada para mapear e definir os modelos (models), as interações com o banco foram executadas diretamente por meio de instruções SQL, garantindo maior controle sobre as consultas e adequação ao critério estabelecido.

A seguir, apresentamos um trecho de código como exemplo dessa abordagem. Nele, é realizada a operação de criação de um novo produto na base de dados, utilizando sequelize.query() com instrução SQL pura:

const { sequelize } = require('../models');

exports.createProduct = async (body) => {

const { title, author, publisher, price, stock } = body;

try {

const createdAt = new Date();

const updatedAt = new Date();

const [result] = await sequelize.query(

`INSERT INTO loja.Products (title, author, publisher, price, stock, createdAt, updatedAt)

VALUES (:title, :author, :publisher, :price, :stock, :createdAt, :updatedAt);`,

{

replacements: { title, author, publisher, price, stock, createdAt, updatedAt },

type: sequelize.QueryTypes.INSERT,

}

);

} catch (error) {

// omissão do resto do código para fins de simplificação

}

};

Por fim, como parte dos requisitos fundamentais do projeto, em consonância com os objetivos da UC Sistemas Distribuídos e Mobile, todas as aplicações desenvolvidas foram **containerizadas com Docker**. A utilização de containers visa simular um ambiente distribuído real, promovendo maior portabilidade, escalabilidade e isolamento entre os serviços. O arquivo docker-compose.yml, localizado na raiz do projeto, é responsável por orquestrar os serviços de forma unificada, facilitando tanto a execução local quanto a futura implantação em ambientes produtivos. Essa abordagem é essencial para compreender a arquitetura de sistemas modernos, nos quais componentes independentes interagem em diferentes containers ou servidores, mantendo a integridade e o desempenho da aplicação como um todo.

No contexto do docker-compose, todos os containers definidos no mesmo arquivo são automaticamente adicionados a uma **rede virtual interna** criada pelo próprio Docker, o que permite que se comuniquem entre si utilizando o nome do serviço como se fosse um endereço. Por exemplo, o serviço de geração de relatórios store-manager pode se conectar ao banco de dados da aplicação principal referindo-se a ele pelo nome do serviço definido no docker-compose.yml, como report-engine. Esse recurso facilita a troca de dados entre os serviços, elimina a necessidade de configurações manuais de rede e simula o comportamento típico de sistemas distribuídos, onde diferentes aplicações interagem entre si em ambientes isolados, mas interconectados.

**Considerações Finais**

Fazer o projeto foi muito bacana, pois aprendi a usar uma nova ferramenta que é fazer um projeto em equipe pelo Github e VS Code! A elaboração de um projeto nem sempre é fácil, mas quando se pensa e trabalha em conjunto, o desenvolvimento se torna mais prazeroso e animador! (Alice)

A parte que achei mais interessante do projeto foi trabalhar em grupo e tentar entender as lógicas das pessoas envolvidas. Foi uma experiência legal perceber como cada um pensa de um jeito diferente e, ao mesmo tempo, conseguir me adaptar para entender o que eles queriam passar. Outra novidade para mim foi usar a parte de projetos do GitHub, onde colocamos o Kanban em prática em equipe. No começo parecia complicado, mas acabou sendo tranquilo. Também tive contato com uma tecnologia nova, o Sequelize, o que tornou o desenvolvimento um pouco desafiador até eu descobrir que o esperado era usar SQL puro, "rsrs". No geral, não tive grandes dificuldades, só precisei focar em entender as ideias dos colegas e colaborar para o sucesso do projeto. (Samuel)

Acredito que o projeto atingiu um nível profissional. Alguém sem uma base de conhecimento sólida e sem um tempo mínimo de estudo não conseguiria realizá-lo com a mínima qualidade.

As tecnologias utilizadas foram escolhidas justamente por estarem presentes no mercado de trabalho atual. Optamos pelo modelo arquitetural MVC, que, na minha visão, é um dos mais utilizados atualmente em aplicações reais.

Pessoalmente, o maior desafio foi o trabalho em grupo. Lidar com pessoas com diferentes níveis de conhecimento exige paciência, flexibilidade e boa comunicação. Além disso, montar toda a estrutura do projeto do zero me obrigou a revisitar conceitos que eu não utilizava há algum tempo. A decisão de utilizar SQL puro nas consultas me tirou da zona de conforto. Confesso que, em alguns momentos, precisei da ajuda da inteligência artificial — e considero isso totalmente normal. Nem sempre saberemos tudo ou lembraremos de tudo, mas é essencial saber que algo é possível e ir atrás do conhecimento necessário para realizá-lo.

Um filósofo disse que ensinar é como acender uma chama, e não como transbordar um copo. O que vimos em sala e realizamos nesse trabalho representa exatamente isso: a chama do conhecimento sendo alimentada. E, sem dúvida, esse projeto ajudou a “encher um pouco mais o copo”, como uma metáfora para a evolução do nosso saber.

Finalizo agradecendo ao professor Adailton pelo ensino e pelo trabalho proposto, que foi de grande valia para o aprimoramento do meu conhecimento técnico em várias tecnologias. (John)

Para mim esse projeto foi bastante desafiador em muitos aspectos. Primeiro que tive que me dedicar em poucos meses a aprender a programar em JavaScript, mas até ia tudo bem, até quando me deparei pela primeira vez com uma estrutura real de projeto para desenvolvimento de software. Entender as relações entre routes, controllers, services, repositories e models exigiu demais de mim, especialmente quando estávamos testando as rotas no Postman para ver se estava tudo rodando direito. Aí vinham os erros na execução e parar para compreender cada um foi exatamente como diz o brocado popular ***“No pain, no gain”*** (sem dor, sem ganho).

A alteração dos Repositories para SQL puro foi irritante. Acho que foi a pior parte.

Mas fica muitos aprendizados, especialmente das tecnologias utilizadas tais como, Node.js, Sequelize, Conteiners, Docker, MySQL, questões do próprio código e a aplicação da arquitetura MVC. Além disso, o esforço de trabalhar em grupo, com perfis e níveis de conhecimento distintos que trazem bons desafios.

Enfim, resta-nos agora a impermanência da vida, um futuro a se descortinar diante de todos nós, com a coragem e a fé de que podemos conquistar nossos sonhos e a vida que almejamos, seja profissional ou pessoal.

Gratidão ao professor Adailton por acreditar em nós e deixar em nossa história uma tão expressiva e importante marca na nossa trajetória acadêmica. (Dinaldo)

**Bibliografia**

**ALURA.** Node.js: o que é, características, vantagens e quando usar. Alura, 2023. Disponível em: <https://www.alura.com.br/artigos/node-js-definicao-caracteristicas-vantagens-usos>. Acesso em: 04 jun. 2025.

**COULOURIS, George; DOLLIMORE, Jean; KINDBERG, Tim; et al.** Sistemas distribuídos. [S.l.]: Bookman, [s.d.]. Disponível em: <https://app.minhabiblioteca.com.br/reader/books/9788582600542>. Acesso em: 04 jun. 2025.

**IBM.** Docker: o que é, como funciona e por que usar? IBM Think. Disponível em: <https://www.ibm.com/br-pt/think/topics/docker>. Acesso em: 09 jun. 2025.

**IBM.** O que são contêineres? IBM. Disponível em: <https://www.ibm.com/br-pt/topics/containers>. Acesso em: 09 jun. 2025.

**MICROSOFT.** Introdução aos contêineres e Docker. Microsoft Learn. Disponível em: <https://learn.microsoft.com/pt-br/dotnet/architecture/microservices/container-docker-introduction/>. Acesso em: 09 jun. 2025.

**MICROSOFT.** O que é um banco de dados relacional? Microsoft Azure. Disponível em: <https://azure.microsoft.com/pt-br/resources/cloud-computing-dictionary/what-is-a-relational-database>. Acesso em: 04 jun. 2025.

**MONTEIRO, Eduarda Rodrigues; MENGATO JUNIOR, Ronaldo C.; LIMA, Bruno Santos de; et al.** Sistemas distribuídos. [S.l.]: SAGAH, 2020. Disponível em: <https://app.minhabiblioteca.com.br/reader/books/9786556901978>. Acesso em: 04 jun. 2025.

**ORACLE.** O que é um banco de dados relacional? Oracle. Disponível em: <https://www.oracle.com/br/database/what-is-a-relational-database/>. Acesso em: 04 jun. 2025.

**PRESSMAN, Roger S.; MAXIM, Bruce R.** Engenharia de software: uma abordagem profissional. 8. ed. Porto Alegre: AMGH, 2016.

**SOMMERVILLE, Ian.** Engenharia de software. 8. ed. São Paulo: Pearson Addison Wesley, 2007.

**ZENKER, Aline Maciel; SANTOS, Jailson Costa dos; COUTO, Júlia M. Colleoni; et al.** Arquitetura de sistemas. [S.l.]: SAGAH, 2019. Disponível em: <https://app.minhabiblioteca.com.br/reader/books/9788595029767>. Acesso em: 04 jun. 2025.