UNIVERSIDADE FEDERAL DA FRONTEIRA SUL
PROGRAMAÇÃO II
DOCENTE: GIANCARLO DONDONI SALTON

CAMPUS DE CHAPECÓ

CURSO DE CIÊNCIA DA COMPUTAÇÃO / VESPERTINO

TRABALHO INTEGRADOR

Elaboração:

Dieneffer Pereira Lima (discente)

Matrícula: 20240012308

dieneffer.lima@estudante.edu.uffs.br

CHAPECÓ, NOVEMBRO, 2025

1. Introdução

Este projeto tem como objetivo principal a prática e integração de conhecimentos adquiridos nos componentes curriculares de Programação II, Banco de Dados e Engenharia de Software. O trabalho consiste no desenvolvimento de uma aplicação web completa, para atender às necessidades da **Borracharia Pereira Lima**, localizada em Bom Jesus, RS. O sistema busca substituir os atuais registros manuais em cadernos por uma solução digital que oferece um gerenciamento simples e eficiente de serviços, materiais consumidos, e pagamentos recebidos. O escopo inclui funcionalidades obrigatórias como um sistema robusto de **autenticação e autorização** com dois perfis , a implementação de operações **CRUD** (Create, Read, Update, Delete) em entidades como Materiais, Serviços e Despesas , e a exibição de dados analíticos em um **Dashboard**. Um requisito funcional central é a **Gestão de Serviços (Vendas)** e o **Controle de Notas Fiscais (Contas a Receber)**, este último essencial para formalizar transações de pagamento a prazo , garantindo maior segurança e eficiência no controle administrativo da empresa.

2. Lógica Utilizada

No frontend, a aplicação é estruturada como uma Single Page Application. O arquivo App.jsx é o ponto central dessa camada. Nele é definido um estado chamado telaAtual, que funciona como um "roteador manual": em vez de utilizar uma biblioteca externa de rotas, o sistema controla qual tela deve ser exibida com base nesse estado. Logo na inicialização do App, a aplicação verifica se existe um token salvo no localStorage; se existir, o sistema assume que o usuário já está autenticado e começa na tela "Inicial". Se não houver token, a primeira tela exibida é a de "Login". Essa decisão já mostra a integração entre autenticação e navegação: a presença do token é usada como critério para permitir acesso à área administrativa.

Ainda no App.jsx foi criado um conjunto de funções simples de navegação, como irParaLogin, irParaInicial, irParaEstoque, irParaCadastroMateriais, irParaCadastroServicos, irParaCadastroDespesas, irParaCaixa, irParaCadastroNotaFiscal e irParaRelatorioFinanceiro. Cada uma dessas funções apenas altera o valor de telaAtual para uma string específica, e o componente principal usa um switch para decidir qual página renderizar com base nessa string. Essa abordagem, embora simples, deixa a lógica de navegação bem explícita e fácil de entender: quando o usuário clica em um botão em alguma tela, a página chama uma dessas funções recebidas via props, e o App passa a renderizar o componente correspondente.

Outro ponto importante em App.jsx é o estado vendaPrazoId. Ele guarda o identificador de uma venda realizada a prazo, para ser reutilizado posteriormente na tela de CadastroNotaFiscal. A função irParaCadastroNotaFiscal recebe o id da venda como parâmetro; se esse id existir, ele é armazenado tanto no estado vendaPrazoId quanto, opcionalmente, no localStorage em um objeto chamado ultimaVenda. Isso mostra uma preocupação em preservar o contexto entre telas: a venda gerada no caixa é ligada diretamente à nota fiscal, evitando que o usuário precise procurar manualmente essa informação. A função handleLogout também está centralizada no App e é responsável por limpar o token, o usuário e a ultimaVenda do localStorage, voltando a aplicação para a tela de Login. Essa organização confere ao App o papel de orquestrar o fluxo geral do sistema, mantendo as demais páginas mais focadas em suas funcionalidades específicas.

A TelaInicial.jsx é a porta de entrada da “Área Administrativa” após o login. Nela é feita a leitura do usuário autenticado a partir do localStorage, usando um useEffect para buscar os dados salvos sob a chave “usuario”. Se não houver usuário, a lógica chama handleLogout, garantindo que ninguém consiga acessar a área restrita sem estar corretamente autenticado. Essa verificação reforça a ligação entre o frontend e o mecanismo de autenticação do backend. Uma vez que o usuário é carregado, a tela exibe suas informações básicas, como tipo de usuário e nome ou e-mail, e apresenta uma grade de botões que funcionam como atalhos para as principais funcionalidades: controle de estoque, caixa, cadastro de materiais, despesas, serviços e, mais recentemente, a geração de relatório financeiro. Cada botão simplesmente dispara uma função de navegação recebida via props, por exemplo irParaEstoque, irParaCadastroServicos ou irParaRelatorioFinanceiro. Desse modo, a TelaInicial concentra a visão geral do sistema, enquanto a responsabilidade de trocar de página permanece com o componente App, mantendo baixo acoplamento entre a interface e a lógica de roteamento.

A tela de RelatórioFinanceiro.jsx é um exemplo claro da integração entre frontend e backend com foco em uma funcionalidade analítica. Esse componente define estados locais para loading e error usando useState, que são utilizados para controlar o feedback visual durante a geração do relatório. Quando o usuário clica no botão “Gerar Relatório de Vendas BRUTO”, é chamada a função handleGenerateReport. Essa função é assíncrona e segue um fluxo bem definido: primeiro ela marca loading como verdadeiro e limpa qualquer mensagem de erro anterior; em seguida monta o endpoint da API, que é um caminho fixo para relatórios de vendas brutas, no formato “/api/relatorios/vendas/bruto”, usando a constante API_URL como base. Com axios.get, o frontend envia uma requisição GET ao backend, aguarda a resposta e, se estiver tudo certo, extrai do corpo da resposta o valor total, exibindo-o ao usuário em um alerta no formato monetário. Caso algum erro ocorra — seja por falha de conexão, seja por problema interno na API — a função captura a exceção, registra o erro no console para fins de depuração e atualiza o estado error com uma mensagem amigável para o usuário. Ao final do processo, independentemente de sucesso ou falha, loading volta a ser falso, e a tela pode exibir mensagens como “Gerando Relatório Bruto...” ou uma mensagem de erro condizente. A tela também possui um botão de “Voltar à Área Administrativa”, que apenas chama a função irParaInicial recebida via props, retornando o usuário à TelaInicial. Essa estrutura demonstra a aplicação prática do consumo de uma API REST a partir do frontend, com tratamento de estados de carregamento e erro, como pedido nas instruções do trabalho.

No backend, o arquivo server.js é o ponto de partida da API. Ele inicia carregando as variáveis de ambiente com dotenv, o que permite separar, em um arquivo .env, informações sensíveis como nome do banco de dados, usuário, senha, host e a chave secreta para geração de tokens JWT. No exemplo fornecido, são definidas variáveis como DB_NAME, DB_USER, DB_PASS, DB_HOST e JWT_SECRET, utilizadas na configuração do Sequelize e da autenticação. Em seguida, server.js cria uma instância do Express, define a porta do servidor e aplica middlewares globais importantes: cors, para permitir que o frontend — normalmente rodando em outra porta — consiga acessar a API, e express.json, para que o servidor consiga interpretar corretamente requisições com corpo em JSON. O arquivo também inicializa o Passport com a estratégia JWT previamente configurada em config/passport.js, garantindo que rotas protegidas possam validar o token de autenticação enviado pelo cliente.

Depois disso, são registradas as rotas principais do sistema. O agrupamento das rotas segue uma lógica semântica: “/api/auth” centraliza tudo que diz respeito a autenticação e cadastro de

usuários, enquanto “/api/materiais”, “/api/estoque”, “/api/servicos”, “/api/despesas” e “/api/venda” estão relacionados às entidades de domínio da aplicação. A rota “/api/notas-fiscais” trata da emissão e consulta das notas. A integração com os relatórios financeiros é feita por meio da rota “/api/relatorios”, que é exatamente a base utilizada na tela de RelatorioFinanceiro.jsx. Quando o frontend chama “/api/relatorios/vendas/bruto”, o Express direciona a requisição para o router de relatórios, que é responsável por consultar o banco de dados, somar as vendas e devolver ao cliente o valor total no formato adequado. Esse caminho demonstra claramente a sequência: a ação do usuário no botão dispara uma função JavaScript no frontend, que chama um endpoint específico no backend, que por sua vez acessa o banco via Sequelize, realiza um cálculo e devolve o resultado para ser exibido na interface.

Ainda em server.js, a função startServer organiza o fluxo de inicialização da aplicação backend. Ela tenta autenticar a conexão com o banco por meio de sequelize.authenticate. Caso a conexão seja bem-sucedida, são sincronizados os modelos com o banco (sequelize.sync), garantindo a criação ou atualização das tabelas de acordo com as definições dos models. Só depois disso o servidor começa a escutar na porta configurada. Caso algum erro aconteça ao conectar-se ao banco ou ao subir o servidor, a mensagem de erro é exibida no console, o que facilita muito o processo de depuração e evidencia a preocupação em tratar falhas de forma clara.

Em termos de integração conceitual, o conjunto desses arquivos mostra bem a lógica de ponta a ponta do sistema. O usuário acessa o frontend, faz login e tem seu token salvo localmente. A partir da TelaInicial, ele navega para módulos específicos, como controle de estoque, caixa, cadastros ou relatórios. Cada ação importante — seja cadastrar um material, lançar uma venda ou gerar um relatório de vendas brutas — corresponde a uma requisição enviada ao backend, onde a API valida dados, aplica regras de negócio, conversa com o banco de dados e devolve a resposta. Do ponto de vista de Programação II, isso reforça conceitos como manipulação de estado no frontend, uso de funções assíncronas e consumo de APIs