CENTRO ESTADUAL DE EDUCAÇÃO TECNOLÓGICA PAULA SOUZA

FATEC PRAIA GRANDE

Desenvolvimento de Software Multiplataforma

Anthony Fernandes do Vale Passos

Camile Benucci Costa

Diego Pereira da Silva

Guilherme Pereira Cardoso de Andrade

Pedro Ricardo da Silva Tavares

BENUCCI ARTESANATO

Praia Grande – SP

2025

Anthony Fernandes do Vale Passos

Camile Benucci Costa

Diego Pereira da Silva

Guilherme Pereira Cardoso de Andrade

Pedro Ricardo da Silva Tavares

BENUCCI ARTESANATO

Projeto de Curricularização da Extensão apresentado ao Curso Superior de Tecnologia em Desenvolvimento de Software Multiplataforma da Fatec de Praia Grande, orientado pelo Prof. Alessandro Ferreira Paz Lima, como requisito para aprovação da disciplina de Laboratório de Desenvolvimento Mobile.

Praia Grande – SP

2025

SUMARIO

1. INTRODUÇÃO

1.1. Objetivo do Documento

Este documento tem como objetivo apresentar a documentação técnica do Benucci Artesanato, um marketplace especializado em produtos artesanais. Ele descreve a visão geral do sistema, seu funcionamento, tecnologias utilizadas, arquitetura adotada e principais funcionalidades. Além disso, estabelece a base de entendimento para desenvolvedores, stakeholders e qualquer pessoa envolvida no ciclo de vida do software.

1.2. Visão Geral do Sistema

O Benucci Artesanato é uma plataforma digital que conecta artesãos e compradores, oferecendo uma experiência de compra personalizada, segura e escalável.
A aplicação utiliza uma arquitetura integrada composta por:

- **Frontend** que consome a API central do sistema;
- **API RESTful em Spring Boot**, responsável por coordenar todas as operações;
- **Banco de dados PostgreSQL**, garantindo armazenamento robusto e confiável;

O backend fornece endpoints estratégicos para autenticação, gestão de usuários, catálogo de produtos, pedidos e recomendações. Recursos como autenticação JWT, criptografia com BCrypt e validação de permissões reforçam a segurança do ambiente.

1.3. Público-Alvo

Este documento é destinado a:

- Desenvolvedores envolvidos no desenvolvimento, manutenção ou expansão do sistema;
- Professores e avaliadores acadêmicos do projeto;

1.4. Escopo do Projeto

- O projeto engloba o desenvolvimento completo de uma **solução** para o **Benucci Artesanato**, abordando:
- Implementação do **Backend em Spring Boot**, com integração a serviços de banco de dados (PostgreSQL).
- Desenvolvimento da **API RESTful** com *endpoints* para Autenticação, Produtos, Pedidos, Usuários e Categorias.
- Criação do **App Mobile (Frontend)** em React Native para consumo da API.
- Implementação de mecanismos de segurança essenciais (JWT e BCrypt).
- Garantia de escalabilidade e performance nas camadas de dados e aplicação.

1.5. Contexto do projeto

Cliente:

Priscila (Benucci Artes Artesanato)

Problemática:

• Dificuldade em divulgar produtos para além da loja física.
• Clientes não sabem quais itens estão disponíveis em tempo real.
A loja não consegue prever quais produtos terão mais saída em certas épocas (ex.: festas juninas, Natal, Dia das Mães).

Solução:

Desenvolvimento de um aplicativo mobile em React Native. A plataforma integrará um catálogo em tempo real, solucionando a falta de visibilidade dos produtos. Uma API inteligente classificará os itens por categoria, tema e mais vendidos. O app expandirá as vendas para além da loja física, oferecendo uma experiência de compra digital prática e eficiente.

2. REQUISITOS DO SISTEMA

- **2.1. Requisitos Funcionais**
- **Cadastro e login de usuários**
    - Usuário pode criar conta com email, senha, nome, telefone e endereço.
    - Usuário pode logar usando email e senha.
    - Usuário recebe token JWT ao logar.
- **Consulta de produtos**
    - Listar todos os produtos disponíveis.
    - Filtrar produtos por categoria, preço, mais vendidos ou promoções.
    - Consultar **detalhes do produto**: descrição, fotos, preço, estoque e avaliações.
- **Avaliação de produtos**
    - Usuário pode avaliar produtos comprados (nota e comentário).
    - Média de avaliações é exibida na página do produto.
- **Recomendação de produtos**
    - Sistema recomenda produtos similares quando usuário acessa um produto específico.
    - Recomendações baseadas em histórico de compras, categorias e produtos comprados juntos.
- **Gestão de pedidos**
    - Usuário adiciona produtos ao carrinho e finaliza pedido.
    - Usuário escolhe forma de pagamento (Mercado Pago).
    - Usuário escolhe entrega ou retirada na loja.
    - Usuário acompanha status do pedido (em preparação, enviado, entregue).
- **Gestão de produtos (Admin)**
    - Admin pode adicionar, atualizar ou remover produtos.
    - Admin pode criar e gerenciar categorias de produtos.
    - Admin pode gerenciar estoque, preços e promoções.

- **Pagamentos e integração financeira**
    - Integração com Mercado Pago para pagamento online.
    - Confirmação automática do pagamento para liberar pedido.
- **Entrega do produto**
    - Usuário pode informar endereço para entrega.
    - Sistema calcula tempo estimado e fornece rastreamento do pedido.

- **2.2. Requisitos Não Funcionais**
- **Performance:** consultas de produtos e carrinho < 2 segundos.
- **Segurança:** autenticação JWT, senhas criptografadas, rotas protegidas.
- **Escalabilidade:** banco e API suportam aumento de usuários, produtos e pedidos.
- **Disponibilidade:** sistema online 99% do tempo.
- **Portabilidade:** app compatível Android e iOS.
- **Usabilidade:** interface intuitiva, fluxo simples de compra, visual moderno.

- **2.3. Regras de Negócio**
- Usuário só pode criar pedidos se estiver logado.
- Produtos sem estoque não podem ser adicionados ao carrinho.
- Admin é o único autorizado a criar, atualizar ou remover produtos/categorias.
- Pedido só é considerado finalizado após confirmação de pagamento.
- Avaliação de produto só pode ser feita por usuário que comprou o produto.
- Produtos recomendados consideram histórico de compras, categorias e produtos similares.

- **2.4. Casos de Uso**
- **Caso de Uso 1 – Cadastro de Usuário**
- **Ator:** Usuário
- **Ação:** Preenche formulário → envia → recebe confirmação
- **Resultado:** Conta criada, usuário pode logar
- **Caso de Uso 2 – Login de Usuário**
- **Ator:** Usuário
- **Ação:** Envia email e senha → recebe token JWT

- **Resultado:** Usuário autenticado

- **Caso de Uso 3 – Consulta de Produtos**

- **Ator:** Usuário
- **Ação:** Busca produtos → filtra → visualiza detalhes, descrição, imagens e avaliações
- **Resultado:** Usuário vê informações completas do produto

- **Caso de Uso 4 – Avaliação de Produto**

- **Ator:** Usuário
- **Ação:** Avalia produto comprado → envia nota e comentário
- **Resultado:** Avaliação registrada, média de avaliações atualizada

- **Caso de Uso 5 – Criação de Pedido e Pagamento**

- **Ator:** Usuário
- **Ação:** Adiciona produtos ao carrinho → escolhe entrega ou retirada → realiza pagamento
- **Resultado:** Pedido registrado e confirmado, status atualizado

- **Caso de Uso 6 – Recomendação de Produtos Similares**

- **Ator:** Usuário
- **Ação:** Clica em produto → visualiza produtos similares
- **Resultado:** Sugestões exibidas com base em categorias e histórico

- **Caso de Uso 7 – Gestão de Produtos (Admin)**

- **Ator:** Admin
- **Ação:** Adiciona, atualiza ou remove produtos/categorias, gerencia estoque e promoções
- **Resultado:** Alterações refletidas no catálogo do app

3. ARQUITETURA DO SOFTWARE

3.1. Visão Geral da Arquitetura

O sistema **BenucciArtesanato** adota uma arquitetura **Cliente-Servidor (Cliente-API)**, organizada em três camadas lógicas principais:

1. **Aplicativo Mobile (React Native):** Responsável pela interface com o usuário, navegação, visualização do catálogo, autenticação e fluxo de pedidos.
2. **Backend (API REST em Spring Boot):** Atua como o núcleo do sistema, processando requisições, aplicando regras de negócio, garantindo segurança e orquestrando o acesso a dados e cache.
3. **Banco de Dados + Cache:**
    a. **PostgreSQL:** Armazena informações persistentes (usuários, produtos, pedidos, etc.) em um modelo relacional.

A comunicação entre o cliente (Mobile) e o servidor (Backend) é realizada via **HTTP/HTTPS**, utilizando **JSON** como formato padrão de dados.

3.2. Arquitetura do Backend (API)

O *backend* é desenvolvido em Java com **Spring Boot** e segue os princípios de **Arquitetura em Camadas** e design **RESTful**.

Camadas Principais

A organização do código é clara e separada por responsabilidades:

- **Controller Layer:**
    - Recebe requisições HTTP (ex: POST /users, GET /products/{id}).
    - Mapeia a URL para o método correto e retorna a resposta HTTP.
    - *Exemplos:* AuthController, ProductController, OrderController.
- **Service Layer:**
    - Contém a **lógica de negócio** principal do sistema.
    - Valida regras, orquestra operações complexas e gerencia transações.

- **Repository Layer:**
    - Realiza operações de persistência no banco de dados, utilizando o **Spring Data JPA** e o **Hibernate**.
    - *Exemplos:* UserRepository, ProductRepository, OrderRepository.

DTO Layer (Data Transfer Objects)

- Responsável por transportar dados entre as camadas do sistema sem expor diretamente as entidades do banco de dados.

Camada de Segurança:

A segurança é gerenciada pelo **Spring Security**, implementando:

- **Autenticação JWT:** Geração e validação de **JSON Web Tokens**.
- **Filtro de Autorização JWT:** Intercepta cada requisição protegida para validar o token presente no *header* Authorization: Bearer <token>.
- **BCrypt:** Utilizado para **criptografia** segura das senhas dos usuários no banco de dados.

Fluxo de Requisições

1. O cliente envia uma requisição (ex: POST /auth/login).
2. A camada de **Segurança** intercepta e, se o token for válido, permite a passagem.
3. O **Controller** mapeia e recebe a requisição.
4. O **Service** executa a regra de negócio.
5. O **Repository** acessa o banco de dados **PostgreSQL**.
6. A resposta é serializada em JSON e retorna ao cliente Mobile.

3.3. Arquitetura do Mobile (React Native)

O aplicativo mobile é construído com **React Native**, focando em modularidade e uma comunicação eficiente com a API.

Componente	Detalhe

Navegação	Utiliza **React Navigation**, organizando o fluxo via Stack Navigator (para login/cadastro) e Bottom Tabs (para a área interna principal).
Comunicação API	O **Axios** é usado como cliente HTTP, configurado com *interceptors* para injetar o **token JWT** automaticamente no *header* de todas as requisições protegidas.
Gerenciamento de Estado	Utiliza a **Context API** do React para gerenciar o estado global da aplicação (ex: dados do usuário logado, carrinho de compras). O **AsyncStorage** é usado para persistir dados locais, como o token JWT.
Estrutura	O código é dividido em componentes reutilizáveis, telas (*Screens*) e uma camada de serviços dedicada à comunicação com a API.

3.4. Integração com o Algoritmo de Machine Learning

A integração entre o backend em **Spring Boot** e o módulo de **Machine Learning** desenvolvido em **Python (Scikit-learn)** é responsável por fornecer funcionalidades inteligentes ao aplicativo Benucci Artes, como **recomendações de produtos.**

Essa integração foi projetada de forma modular, garantindo que o modelo de ML possa ser atualizado, refeito ou expandido sem necessidade de alterar a API principal ou o aplicativo mobile.

Arquitetura de Integração

A arquitetura segue o padrão **API-to-API**, onde:

- O backend em **Spring Boot** expõe rotas como:
  - GET /recomendacoes/{usuarioId}
  - GET /recomendacoes/populares
- O módulo de Machine Learning é executado separadamente em um serviço Python, que pode funcionar como:
  - um microserviço Flask/FastAPI,
  - um job offline que gera arquivos .pkl ou .json,
  - ou um servidor interno apenas para previsão.

A comunicação entre backend e ML é realizada por **HTTP** ou por **troca de arquivos de modelo** (dependendo da implantação final).

Fluxo de Comunicação

O app envia uma requisição para o backend pedindo recomendações:

- GET /recomendacoes/{productId}
- **O backend aciona o módulo de Machine Learning**, enviando o ID do usuário ou o tipo de recomendação solicitado.
- **O modelo de ML retorna IDs de produtos recomendados**, usando uma lógica simples baseada em semelhança (produtos com o mesmo ID ou IDs próximos).
- **O backend consulta o banco de dados** para obter os detalhes completos desses produtos.
- **O backend retorna a lista final ao app**, que exibe os produtos recomendados na interface.

3.5. Banco de Dados e Estrutura de Dados

O banco de dados principal é o **PostgreSQL**, utilizado devido à sua robustez e capacidade de transações.

3.6. Tecnologias Utilizadas

Categoria	Tecnologia	Detalhe/Versão
Backend	**Java**	Versão 17 ou superior.
Backend	**Spring Boot (java)**	Framework principal.
Backend	**Spring Data JPA**	Interface para acesso a dados.
Backend	**Spring Security & JWT**	Autenticação e Autorização.
Frontend	**React Native**	Framework principal de desenvolvimento mobile.
Frontend	**React Navigation**	Gerenciamento de navegação no app.
Banco de Dados	**PostgreSQL**	Banco de dados relacional persistente.
Build Tool	**Maven**	Gerenciamento de dependências e build do Backend.

Documentação	**Swagger (OpenAPI)**	Geração automática da documentação da API.
Machine Learning	**Python + Scikit-learn**	Algoritmos de recomendação e previsão de demanda.
Machine Learning	**Pandas/Numpy**	Tratamento e análise de dados.
APIs Externas	**Mercado Pago API**	API para processar e realizar pagamentos.

4. MODELAGEM DO SISTEMA

Esta seção detalha a arquitetura lógica e de dados do sistema, utilizando a notação UML (Unified Modeling Language) para processos e a Modelagem Entidade-Relacionamento para o banco de dados.

4.1. Diagramas UML

4.1.1. Diagrama de Casos de Uso

Os Casos de Uso (UCs) definem as funcionalidades primárias do sistema, agrupadas por perfil de usuário.

Ator	Casos de Uso (UCs)
Usuário Não Autenticado	**UC01 – Cadastrar Usuário** **UC02 – Realizar Login**
Usuário Autenticado	**UC04 – Listar Produtos** **UC07 – Ver Detalhes do Produto**
Comprador	**UC05 – Criar Pedido**
Admin	**UC03 – Cadastrar Produto** **UC09 – Editar Produto** **UC10 – Remover Produto** **UC06 – Atualizar Status do Pedido** **UC14 – Gerenciar Categorias** (criar, editar, remover) **UC15 – Gerenciar Estoque e Informações do Produto**
Sistema API	**UC11 – Validar Token JWT** **UC12 – Aplicar Regras de Negócio** **UC13 – Consultar Banco de Dados** **UC16 – Integrar com Sistema de Recomendação (ML)**

4.1.2. Diagrama de Classes (Padronização Consistente)

O modelo de classes reflete as principais entidades de negócio. A nomenclatura de classes e atributos é **padronizada para o inglês** (padrão do código Java/Spring Boot).

Classe (Entidade)	Atributos Principais	Relacionamentos Chave
User	- id: Long, - name: String, - email: String, - password: String, - type: Enum	1 -- N **Product** (Artesão), 1 -- N

	(COMPRADOR / ARTESAO), - createdAt: DateTime, - updatedAt: DateTime	**Order** (Comprador), 1 -- N **Rating**
Category	- id: Long, - name: String, - slug: String, - description: String, - createdAt: DateTime, - updatedAt: DateTime	1 -- N **Product**, 1 -- N **Subcategory**
Subcategory	- id: Long, - categoryId: FK, - name: String, - slug: String	N -- M **Theme**
Product	- id: Long, - name: String, - description: String, - price: Decimal, - images: List<String>, - categoryId: FK, - userId: FK (Artesão)	N -- M **Rating**, N -- M **OrderItem**
Order	- id: Long, - status: Enum, - total: Decimal, - userId: FK (Comprador), - createdAt: DateTime	1 -- N **OrderItem**, 0..1 -- 1 **Payment**
OrderItem	- id: Long, - quantity: Integer, - unitPrice: Decimal, - orderId: FK, - productId: FK	N -- 1 **Order**, N -- 1 **Product**
Rating (Avaliação)	- id: Long, - score: Integer, - comment: String, - userId: FK, - productId: FK	N -- 1 **User**, N -- 1 **Product**
Theme	- id: Long, - name: String, - description: String	

4.1.3. Diagramas de Sequência

Detalham a interação temporal entre as camadas para os processos críticos do sistema.

- **Sequência Login:** User → App Mobile → **POST /auth/login** → API (Valida BCrypt) → Banco → Retorna **JWT** para o App.
- **Sequência Cadastrar Produto (Admin):** Admin → App Mobile → **POST /products** → API (Valida Permissões) → Banco (Insere Produto) → Retorna Confirmação.
- **Sequência Criar Pedido (Comprador):** Usuário → App → **POST /orders** → API (Valida Disponibilidade) → Banco (Cria Pedido e Itens) → Retorna Confirmação do Pedido.

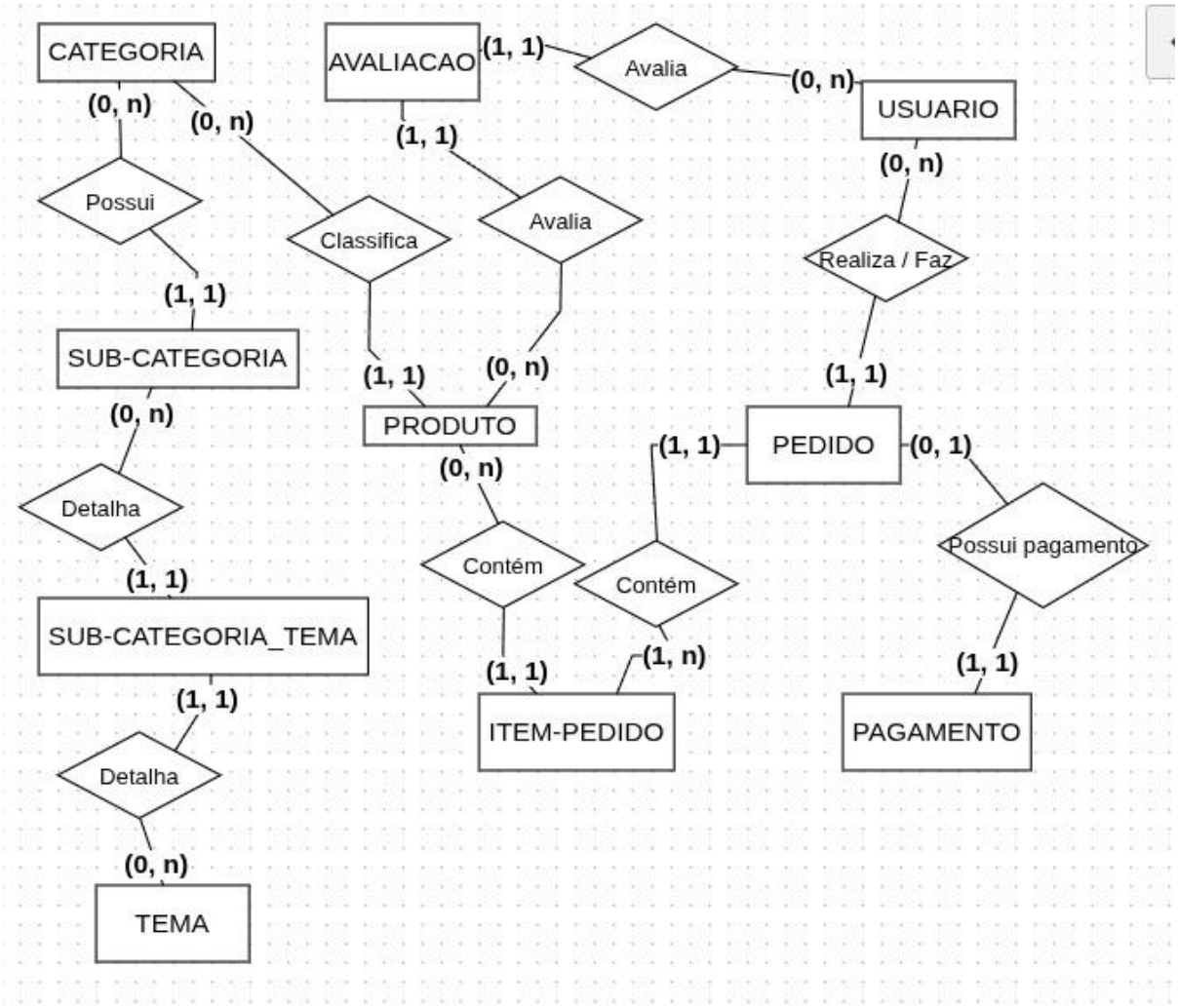
4.1.4. Diagramas de Atividade

Descrevem o fluxo de controle lógico das atividades principais (Login, Criação de Produto, Pedido).

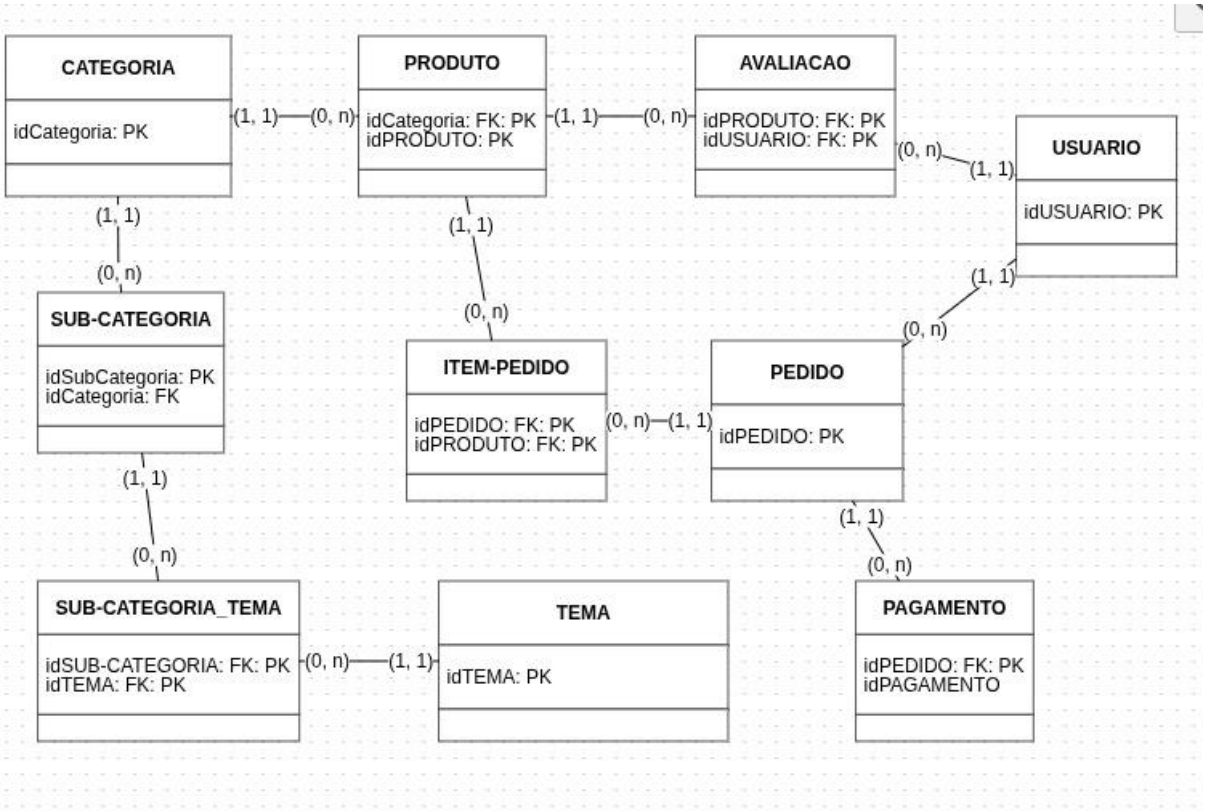
4.2. Estrutura do Banco de Dados

O modelo físico do banco de dados (PostgreSQL) foi desenvolvido com base no Diagrama Entidade-Relacionamento e utiliza o padrão *snake_case* para nomes de colunas (created_at).

4.2.1. Diagrama Conceitual



4.2.2. Diagrama Relacional



O diagrama relacional detalha as tabelas e as chaves primárias e estrangeiras.

4.3. Modelagem do Algoritmo de Machine Learning

A modelagem do algoritmo de Machine Learning para o Benucci Artesanato foi elaborada com o objetivo de **recomendar produtos**. O modelo utiliza dados reais da plataforma, como histórico de compras, categorias mais acessadas e preferências implícitas do usuário.

4.3.1. Objetivo do Modelo

O modelo tem dois principais objetivos:

1. Recomendações Personalizadas

Gerar sugestões de produtos relevantes com base em:

- Categorias mais buscadas
- Temas mais visualizados
- Histórico de pedidos
- Itens avaliados pelo usuário

4.3.2. Tipo de Modelo Utilizado

Para atender às necessidades do sistema, foi planejada uma abordagem híbrida envolvendo:

A. Content-Based Filtering

Utiliza características do produto:

- Categoria → Ex.: "Crochê", "Bordado", "Natal"
- Tema → Ex.: "Infantil", "Religioso", "Flores"
- Subcategoria → Ex.: “Chaveiros”

Bibliotecas:

- **Pandas**
- **Numpy**
- **Scikit-learn** (TF-IDF, Similaridade do Cosseno)

Modelos possíveis:

- KNN
- Matrizes fatoradas (SVD)

4.3.3. Variáveis Consideradas

Tipo	Variável	Descrição
Categoria	categoria	Representa o grupo principal ao qual o produto pertence. Ex.: *“Crochê”, “Bordado”, “Natal”*. É a primeira camada de filtragem usada para identificar produtos semelhantes.
Subcategoria	subcategoria	Detalhamento mais específico dentro da categoria. Ex.: *“Chaveiros”, “Tapetes”, “Acessórios de Crochê”*. Ajuda o modelo a refinar recomendações de produtos realmente próximos.
Tema	tema	Indica temas ou estilos associados ao produto. Ex.: *“Infantil”, “Religioso”, “Flores”, “Datas Festivas”*. Utilizado especialmente para recomendações sazonais.

4.3.4. Pré-Processamento dos Dados

Antes do treinamento, todos os dados passam por:

1. Limpeza

- Remoção de itens sem categoria
- Normalização de nomes (“Croche”, “Crochê”, etc.)
- Remoção de duplicidades

2. Tratamento de Dados Categóricos

- Transformação para vetores usando:
    - **One-Hot Encoding**
    - **TF-IDF para textos**

3. Construção da Matriz Usuário × Produto

Usada para calcular similaridade entre preferências.

4. Normalização

- Escalonamento de valores numéricos, quando necessário

4.3.5. Treinamento do Modelo

O modelo é treinado com base no histórico de uso da plataforma.

Passos principais:

1. Extração de amostras de interação usuário-produto
2. Geração da matriz de similaridade
3. Cálculo da relevância
4. Treinamento do modelo de recomendação
5. Validação com métricas como:
    a. **Precisão@K**
    b. **Recall@K**
    c. **RMSE (se houver nota/avaliação)**

Ferramentas:

- **Scikit-learn**
- **Pandas**
- **Numpy**
- **Joblib** (para salvar o modelo treinado)

4.3.6. Output (Saída do Modelo)

A. Recomendações Personalizadas

Formato de saída:

Só retorna o ID dos produtos.

4.3.7. Integração com o Backend

O backend (Spring Boot) acessa o modelo por meio de:

- **API Flask/FastAPI** (modelo servindo predições em tempo real)
  ou
- **Carregamento de arquivo .pkl** com o modelo treinado
  ou
- **Script Python executado sob demanda**

Rotas integradas:

- GET /recomendacoes/{usuarioId}
- GET /recomendacoes/populares

4.3.8. Atualização Contínua do Modelo

O modelo pode ser atualizado conforme:

- Novo histórico de vendas
- Novas categorias e produtos adicionados
- Mudanças no comportamento do usuário

Planejamento de atualização:

- Semanal (para plataforma com alto fluxo)
- Mensal (adequado ao projeto atual)

5. DESENVOLVIMENTO DO BACKEND

O *backend* do **BenucciArtes** foi desenvolvido utilizando o *framework* **Spring Boot** (Java), estruturado em camadas para garantir organização, segurança e fácil manutenção. Ele fornece todos os serviços essenciais ao aplicativo mobile, utilizando **JWT** para segurança, **BCrypt** para criptografia de senhas e **JPA/Hibernate** para acesso ao banco de dados PostgreSQL.

5.1. Estrutura do Projeto

A estrutura segue as boas práticas do Spring Boot, organizada por pacotes de responsabilidade (controller, service, repository), refletindo a arquitetura em camadas e o escopo de entidades do sistema (Produtos, Pedidos, Categorias, etc.):

src/main/java/br/edu/fatecpg/BenucciArtesanato
│
├── config/
│ └── Configurações de segurança (WebSecurityConfig, filtros Jwt)│
├── controller/
│ └── Endpoints REST (AuthController, UserController, ProductController, OrderController, etc.).
│
├── model/
│ └── Entidades JPA e DTOs (User, Product, Order, Category, etc.).
│
├── repository/
│ └── Interfaces Spring Data JPA para acesso ao banco (UserRepository, ProductRepository, etc.).
│
└── service/
└── Regras de negócio e lógica (AuthService, UserService, ProductService, OrderService).

5.2. Endpoints e Métodos HTTP

A API RESTful é estruturada em torno de diversos controladores, com nomenclatura de rotas e métodos padronizada em inglês.

Controlador	Endpoint	Método	Descrição
Auth	/auth/register	POST	Cria um novo usuário no sistema.
Auth	/auth/login	POST	Autentica o usuário e retorna o token JWT.
User	/users/{id}	GET/PUT/DELETE	Gerenciamento de dados de usuário por ID.
Product	/products	GET/POST	Lista produtos (paginação) e cria novo produto.
Order	/orders	POST	Cria um novo pedido.
Order	/orders/status/{id}	PUT	Atualiza status do pedido (restrito).
Category	/categories/{id}	GET/PUT/DELETE	Gerenciamento de Categoria por ID.
SubCategory	/categories/{id}/subcategories	GET/POST	Lista e cria Subcategorias associadas.
Theme	/themes	GET/POST	Lista e cria Temas.
Payment	/api/payments/preference	POST	Cria preferência de pagamento externo.
Web Hook	/webhook/mercadopago	POST	Recebe notificações de status de pagamento.

5.3. Autenticação e Segurança (JWT)

O sistema utiliza **JWT (JSON Web Token)** como mecanismo principal de autenticação para proteger todas as rotas de acesso a dados.

Fluxo de Autenticação:

1. O cliente envia email e password para o *endpoint* /auth/login.

2. O *backend* valida a senha, criptografada usando **BCrypt**, e encontra os últimos dígitos criptografados.
3. Um token JWT é gerado.
4. O token é enviado ao cliente e deve ser incluído em todas as requisições protegidas subsequentes no *header*: Authorization: Bearer <token>.

Características da Segurança:

- **Assinatura:** O token é assinado usando o algoritmo **HS512**.
- **Filtro:** O **JwtFilter** do Spring Security garante que apenas usuários com tokens válidos acessem rotas protegidas.

5.4. Tratamento de Erros e Logs

O *backend* implementa um tratamento robusto de erros para garantir clareza nas respostas da API e facilitar a depuração.

Tratamento de Erros Centralizado

O tratamento de exceções é **centralizado** usando a anotação **@RestControllerAdvice** do Spring. Isso garante que todos os erros sejam capturados e retornados ao cliente em um formato JSON padronizado, mantendo a consistência.

Exemplos de erros tratados:

- **401 Unauthorized**: Token expirado, ausente ou inválido.
- **404 Not Found**: Recurso não encontrado (ex: produto com ID inexistente).
- **400 Bad Request**: Erros de validação (campos ausentes ou formatação incorreta).

Padrão de Resposta de Erro:

JSON

```
{
 "error": "Recurso não encontrado",
 "status": 404,
```

"timestamp": "2025-11-23T02:40:00"
}

Logs

Os logs são gerados pelo Spring, utilizando o **Logback** como *framework* padrão. O nível de *logging* é configurado para controlar a verbosidade, sendo essenciais para:

- Registrar falhas de autenticação e exceções (ERROR).
- Monitorar o tempo de execução de serviços e acessos aos *endpoints* (INFO/DEBUG).

6. DESENVOLVIMENTO DO APP MOBILE

O aplicativo mobile do **Benucci Artes** foi desenvolvido em **React Native**, seguindo uma arquitetura modular que facilita manutenção, escalabilidade e integração direta com a API REST em Spring Boot. O app é responsável por autenticação, navegação entre telas, e exibição do catálogo de produtos, gerenciando a experiência do usuário de forma fluida.

6.1. Estrutura do Projeto Mobile

A estrutura geral do projeto segue uma organização limpa, baseada em responsabilidades e alinhada com as boas práticas de desenvolvimento React Native:

```
/src
│
├── api/                -> Configuração da instância Axios e interceptors
│     └── api.js
│
├── components/         -> Componentes reutilizáveis (Botões, Cards, Inputs, etc.)
├── contexts/           -> Contextos para gerenciamento de estados globais (Ex: AuthContext)
├── hooks/              -> Hooks personalizados (Ex: useAuth)
├── navigation/         -> Controle de rotas (Stack e Tab Navigators)
├── screens/            -> Telas principais do app (Login, Home, ProductDetails, etc.)
├── services/           -> Funções de acesso à API por módulo (AuthService, ProductService)
├── styles/             -> Estilização global, temas e paleta de cores
│
└── App.js              -> Arquivo raiz do aplicativo
```

Essa organização garante a **separação de preocupações** (lógica de UI, lógica de API, e estado global) para maior manutenibilidade.

6.2. Fluxo de Navegação

O fluxo de navegação utiliza a biblioteca **React Navigation**, organizando as rotas em *stacks* para garantir transições suaves e segurança nas rotas protegidas.

O estado do **AuthContext** determina qual *stack* de navegação é exibida:

- **AuthStack**: Contém rotas de acesso público (/login, /register).
- **AppStack**: Contém rotas protegidas que requerem o token JWT (/home, /products, /profile).

O AuthContext é o *gatekeeper* (porteiro) do aplicativo. Se o **token** for válido, o usuário é redirecionado para o AppStack; caso contrário, permanece no AuthStack.

6.3. Comunicação com a API

A comunicação é gerenciada pelo cliente HTTP **Axios**, configurado no arquivo api/api.js.

Uso do JWT (Token)

Após o login bem-sucedido, o token JWT retornado pelo backend é armazenado no storage (Ex: AsyncStorage) e imediatamente injetado nos headers de todas as requisições futuras usando um interceptor ou a configuração padrão do Axios:

JavaScript
api.defaults.headers.common["Authorization"] = `Bearer ${token}`;

Serviços Implementados

Os acessos à API são encapsulados no diretório /services, garantindo que as telas do app contenham apenas a lógica de apresentação.

Serviço	Endpoint (método)	Função
AuthService	/auth/login (POST)	Realiza login.
AuthService	/auth/register (POST)	Cadastra novo usuário.
UserService	/users/{id} (GET)	Busca dados do usuário.
ProductService	/products (GET)	**Lista produtos**.

ProductService	/products/{id} (GET)	**Detalhes** de um produto.

6.4. Gerenciamento de Estados

O estado global do aplicativo é gerenciado utilizando as ferramentas nativas do React: **Context API** e **Hooks**.

- **AuthContext (Exemplo):** Gerencia os dados cruciais para a sessão:
  - user: Dados do usuário logado.
  - token: Token JWT.
  - login( ), logout( ), register( ): Funções que orquestram a comunicação com o **AuthService**.

Este modelo de gerenciamento de estado evita a dependência de bibliotecas externas complexas (como Redux), mantendo o *bundle size* reduzido e a arquitetura mais simples e eficiente.

6.5. Interface do Usuário e Design System

O design do aplicativo segue uma estética artesanal e aconchegante, alinhada à identidade visual do projeto, implementada através de um **Design System** coeso.

Padrões Adotados:

- **Paleta de Cores:** Utilização de tons claros e suaves (bege, branco, verde suave) para criar um ambiente acolhedor.
- **Acessibilidade:** Fontes legíveis e botões padronizados com alto contraste para feedback visual claro.
- **Layout:** Componentes com cantos arredondados e **Cards** destacando as imagens dos produtos.

Componentes Reutilizáveis:

A criação de uma biblioteca de componentes (ButtonPrimary, InputField, ProductCard, CategoryChip) garante a uniformidade visual e acelera o desenvolvimento de novas telas. Os estilos são centralizados nos arquivos **/styles/colors.js** e **/styles/global.js**, facilitando mudanças de *layout* futuras.

7. IMPLEMENTAÇÃO DO ALGORITMO DE MACHINE LEARNING

A plataforma Benucci Artes incorpora um módulo de Machine Learning com o objetivo de oferecer recomendações inteligentes e auxiliar a loja a prever tendências de venda. O modelo será desenvolvido em Python utilizando **Scikit-learn**, **Pandas** e **NumPy**, e integrado à API em Spring Boot por meio de endpoints específicos.

7.1. Objetivo e Funcionalidade do Modelo

O objetivo principal do algoritmo é:

1. Recomendações Personalizadas

Gerar recomendações de produtos para cada usuário com base em:

- Categorias mais acessadas
- Histórico de pedidos
- Similaridade entre produtos (categoria, subcategoria e tema)
- Popularidade geral de produtos

Recomendações Baseadas no Produto Visualizado
Sempre que o usuário **clica em um produto**, o app chama o endpoint de recomendação passando o **ID do produto**.

- A API retorna produtos que possuem:
    - **A mesma categoria**
    - **A mesma subcategoria**
    - **Temas relacionados**
    - **Características semelhantes** no banco de dados
- Esse tipo de recomendação é rápido e não exige tokenização complexa, funcionando mesmo com pouco dado histórico.

7.2. Coleta e Pré-Processamento de Dados

Os dados utilizados pelo modelo são coletados diretamente do backend, contemplando:

Fontes de Dados

- **Tabela de pedidos** (produtos)
- **Tabela de itens do pedido** (produtos)
- **Categorias e produtos**

- **Acessos/visualizações de produtos** (opcional)
- **Histórico do usuário**

Etapas de Pré-Processamento

1. **Limpeza dos dados**
    a. Remoção de registros inconsistentes
    b. Normalização de strings (categorias, nomes)
    c. Conversão de datas para formato padrão
2. **Transformação**
    a. Criação da matriz usuário × produto (para recomendações)
    b. Normalização de contagem de compras / acessos
    c. Extração de features (mês, época do ano, categoria dominante)
3. **Feature Engineering**
    a. Pesos extras para produtos mais recentes
    b. Pesos para produtos com alto índice de venda
    c. Codificação de categorias (One-Hot Encoding)
4. **Divisão dos dados**
    a. 70% treino
    b. 30% validação

O pipeline é desenvolvido em **Python**, permitindo fácil reexecução sempre que novos dados são adicionados.

7.3. Deployment e Integração com a API

Após o treinamento, o modelo é exportado e disponibilizado para uso pela API.

Forma de Deploy

- O modelo é salvo no formato **.pkl** (Pickle)
- Servido por um microserviço Python (Flask/FastAPI)
  OU integrado diretamente ao backend Java via chamadas HTTP

Integração com o Backend (Spring Boot)

A API expõe endpoints para recomendações:

GET /recomendacoes/{usuarioId}
GET /recomendacoes/populares

Fluxo da Requisição

1. App → API Java
   /recomendacoes/{productId}
2. Microserviço Python
   Carrega o modelo e calcula recomendações
3. Python → Java
   Retorna lista de IDs de produtos
4. Java → App
   Resposta final com produtos formatados

Vantagens

- Deploy independente do backend Java
- Atualização do modelo sem reiniciar a API
- Menor acoplamento entre tecnologias

7.4. Monitoramento e Atualização do Modelo

Para manter o sistema de recomendações eficiente, é necessário:

Monitoramento Contínuo

- Analisar logs de requisições do endpoint de recomendações
- Verificar volume de acertos × rejeições
- Monitorar sazonalidade real versus prevista
- Acompanhar feedback do usuário (opcional)

Atualização Periódica do Modelo

- Reprocessar dados semanal ou mensalmente
- Treinar modelo com novos pedidos
- Reavaliar métricas e comparar versões
- Atualizar o arquivo .pkl e redeploy automático

Versionamento do Modelo

Cada versão deve possuir:

- Identificador único (ex.: model_v3.2.pkl)
- Log de alterações
- Métricas associadas
- Registro de datasets utilizados

8. IMPLANTAÇÃO E MANUTENÇÃO

A implantação e manutenção do sistema Benucci Artes envolvem uma infraestrutura integrada entre o **backend (API Spring Boot)**, o **banco de dados PostgreSQL**, o **aplicativo mobile em React Native** e serviços de monitoramento para garantir disponibilidade, segurança e evolução contínua.

8.1. Estratégia de Deploy (CI/CD)

Deploy Automatizado Usando Render

- O projeto utiliza o **Render** como plataforma principal de deploy, que oferece **CI/CD integrado de forma nativa**. Isso elimina a necessidade de ferramentas externas e permite que todo o fluxo de publicação seja automático.

Como funciona o CI/CD no Render

- O Render monitora continuamente o repositório configurado (GitHub). Sempre que um commit é enviado para o branch conectado — normalmente o **main** — ele executa automaticamente o fluxo:

Fluxo de CI/CD no Render

1. CI – Integração Contínua (Automática pelo Render)

Assim que ocorre um push no repositório:

- O Render verifica se houve mudanças no código.
- Faz o **clone do repositório** atualizado.
- Compila automaticamente a aplicação backend em **Spring Boot**.
- Resolve e instala todas as dependências definidas no pom.xml.
- Executa o build do projeto.

Caso algo falhe (erro de build, dependência quebrada), o deploy é cancelado e o log fica disponível na plataforma.

2. CD – Entrega Contínua (Automática pelo Render)

- Se a etapa anterior for concluída com sucesso, o Render inicia o deploy:

- Executa novamente o build da aplicação.
- Gera o artefato final (JAR).
- Inicia uma nova instância da API com o código atualizado.
- Substitui a versão antiga pela nova sem interrupções significativas.
    - Todo o processo é **100% automático**, incluindo:
    - ✓ Atualização da API
      ✓ Reinício do servidor
      ✓ Reconfiguração das variáveis de ambiente
      ✓ Registros de logs acessíveis via painel

Essa estratégia reduz erros humanos, acelera entregas e aumenta a confiabilidade do sistema.

8.2. Infraestrutura e Hospedagem (Cloud, Servidores, Banco de Dados)

A arquitetura de implantação é composta por três camadas:

1. API Backend (Spring Boot)

Opções viáveis de hospedagem:

- **Railway** (fácil e gratuito)
- **Render (Usamos ele)**
- **AWS EC2**
- **DigitalOcean Droplets**
- **Heroku (plano pago)**

Requisitos mínimos:

- Ter internet.

2. Banco de Dados

O banco PostgreSQL pode ser hospedado em:

- Render PostgreSQL (usamos ele pra colocar a aplicação da api)
- Supabase (Ele hospeda nosso postgre online)

Configurações importantes:

- SSL habilitado
- Rotação de credenciais
- Backups automáticos diários

8.3. Monitoramento e Logs

Para garantir estabilidade e detectar problemas rapidamente, o sistema prevê monitoramento tanto no backend quanto no mobile:

Backend:

Possíveis ferramentas:

- **Spring Boot Actuator** → métricas e endpoints de saúde
- **Railway/Render Logs** → logs em tempo real
- **Sentry** → captura de exceções e alertas

Logs importantes:

- Tentativas de login
- Erros de autenticação JWT
- Falhas de integração com o banco
- Exceções inesperadas (500)

9. DOCUMENTAÇÃO TÉCNICA E DE USUÁRIO

Esta seção reúne todo o material de suporte necessário para desenvolvedores, usuários finais e avaliadores técnicos. Inclui um guia técnico para manutenção e evolução do sistema, bem como instruções claras de uso do aplicativo.

9.1. Guia do Desenvolvedor

O Guia do Desenvolvedor tem como objetivo fornecer todas as informações necessárias para instalação, execução, manutenção e expansão do backend e do aplicativo mobile.

Requisitos de Ambiente

- **Backend**
  - Java 17
  - Maven
  - PostgreSQL 14+
  - IDE recomendada: IntelliJ IDEA / Eclipse / VS Code
- **Mobile**
  - Node.js 18+
  - React Native CLI
  - Android Studio (SDK + Emulador)
  - IDE recomendada: VS Code.

EM ESPERA SERÁ ENVIADO EM BREVE!!!!

9.2. Manual do Usuário

O Manual do Usuário descreve as principais ações que o cliente e o artesão podem realizar no aplicativo.

Abertura do Aplicativo

Ao abrir o app, o usuário encontra:

- Tela de login
- Botão de registrar nova conta
- Interface simples e intuitiva

1. Criar Conta

O usuário deve informar:

- Nome
- Email
- Senha
- Telefone
- Endereço

Após enviar, o app confirma o cadastro e redireciona para o login.

2. Fazer Login

O usuário informa email e senha.
Se autenticado, a API retorna um **token JWT**, e o app libera o acesso.

3. Catálogo de Produtos

Funcionalidades:

- Ver todos os produtos disponíveis
- Filtrar por categoria (ex.: artesanato, decoração, datas especiais)
- Selecionar produto para ver detalhes

4. Ver Detalhes do Produto

A tela exibe:

- Fotos
- Nome
- Descrição
- Preço
- Disponibilidade

5. Área do Usuário

O usuário pode:

- Editar dados pessoais

6. Área do Admin

O Admin pode:

- Cadastrar produtos
- Atualizar produtos
- Remover produtos
- Acompanhar pedidos
- Alterar status do pedido (criado → enviado → concluído)

9.3 Documentação da API

Base URL: https://benucci-artesanato.onrender.com

Autenticação: JWT Bearer Token (necessário para rotas protegidas)

Swagger: https://benucci-artesanato.onrender.com/swagger-ui/index.html#

1. Auth controller:

Responsável por autentificação e registros dos usuários no sistema. O usuário só poder fazer requisições se tiver logado e com o token jwt de autentificação.

Registrar Usuário:

• Método: POST

• Rota: /auth/register

• Descrição: Cria um novo usuário no sistema. Pega a senha digitada e criptografa ela usando o bcrypt depois salva a senha encriptografada no banco.

Body (JSON):

{

"name": "string",

"email": "string",

"password": "string",

"phoneNumber": "string",

"address": "string",

"cpf": "string",

"role": "string"

}

Login

• Método: POST

• Rota: /auth/login

• Descrição: Autentica o usuário e retorna um token JWT.

• Body (JSON):

{

"email": "guilherme.a232ds@gmail.com",

"password": "senha"

}

User Controller:

Obter Usuário:

• Método: GET

• Rota: /users/{id}

• Descrição: Retorna os dados de um usuário específico.

• Authorization: Bearer Token

{

"name": "Mario Sérgio ",

"email": "m@email.com",

"password":
"$2a$10$abvsZp38dJxE7K4s4YFwjOEGHYXE.JkULSLj1sX/6EcrI5qxfyH4G",

"phoneNumber": "13988120276",

"address": "{"Endereco blabs das d"},

"role": "ROLE_USER",

"cpf": "99236555047"

}

Atualizar Usuário:

• Método: PUT

• Rota: /users/{id}

• Descrição: Atualiza dados do usuário.

• Body (JSON):

{

"name": "Mario Sérgio ",

"email": "m@email.com",

"password":
"$2a$10$abvsZp38dJxE7K4s4YFwjOEGHYXE.JkULSLj1sX/6EcrI5qxfyH4G",

"phoneNumber": "13988120276",

"address": "{"Endereco blabs das d"},

"role": "ROLE_USER",

"cpf": "99236555047"

}

Deletar Usuário:

• Método: DELETE

• Rota: /users/{id}

• Descrição: Remove o usuário do sistema.

Listar Usuarios:

• Método: GET

• Rota: /users

• Descrição: Lista todos os usuarios.

• Body (JSON):

Obter usuario pelo email:

• Método: GET

• Rota: /users/email/{email}

• Descrição: Lista os dados de um usuario com base no email.

Theme Controller

Atualizar tema:

• Método: PUT

• Rota: /themes/{id}

• Descrição: Atualizar um tema com base no id.

Deletar tema:

• Método: DELETE

• Rota: /themes/{id}

• Descrição: Remove.

Listar todos os temas:

• Método: GET

• Rota: /themes

• Descrição: Listar todos.

Criar Tema:

• Método: POST

• Rota: /themes

• Descrição: Cria.

Subacategory-theme-controller

Listar Ids de temas associados a uma subcategoria:

• Método: GET

• Rota: /subcategory-themes/{subcategoryId}

• Descrição: Lista todos os temas associados a uma subcategoria.

Listar Ids de temas associados a uma subcategoria:

• Método: PUT

• Rota: /subcategory-themes/{subcategoryId}

• Descrição: Atualiza temas associados a uma subcategoria.

Associar temas a uma subcategoria

• Método: POST

• Rota: /subcategory-themes

• Descrição: Associar temas a uma subcategoria.

{ "categoryId": 0, "subcategoryId": 0, "themeIds": [0] }

Remove um tema espesifico de uma subcategoria

• Método: DELETE

• Rota: /subcategory-themes/{subcategoryId}/{themeId}

• Descrição: Remove um tema especifico de uma subcategoria, precisa passar o id da subcategoria e do tema.

Product Controller:

• Método: GET

• Rota: /produtcts/{id}

• Descrição: retornar todos os dados do produto com base no id.


```
"id": 0,
"name": "string",
"description": "string",
"price": 0,
"stock": 0,
"imageUrls": [
 "string"
],
"mainImageUrl": "string",
"createdAt": "2025-11-29T18:13:15.393Z",
"updatedAt": "2025-11-29T18:13:15.393Z",
"subcategoryId": 0,
"subcategoryName": "string",
"categoryId": 0,
"categoryName": "string",
"themeIds": [
 0
],
"themeNames": [
 "string"
]
}
```


Atualizar produto:

• Método: PUT

• Rota: /produtcts/{id}

• Descrição: Atualiza um produto

Remover produto:

• Método: DELETE

• Rota: /produtcts/{id}

• Descrição: delete.

Criar produto:

• Método: POST

• Rota: /produtcts

• Descrição: envia o json do produto, seleciona uma ou mais imagens, a primeira selecionada vai ser a imagem principal, as imagens serão salvas em outro banco Storage que armazena somente as imagens, e, o link dessas imagens será salvo no banco principal.

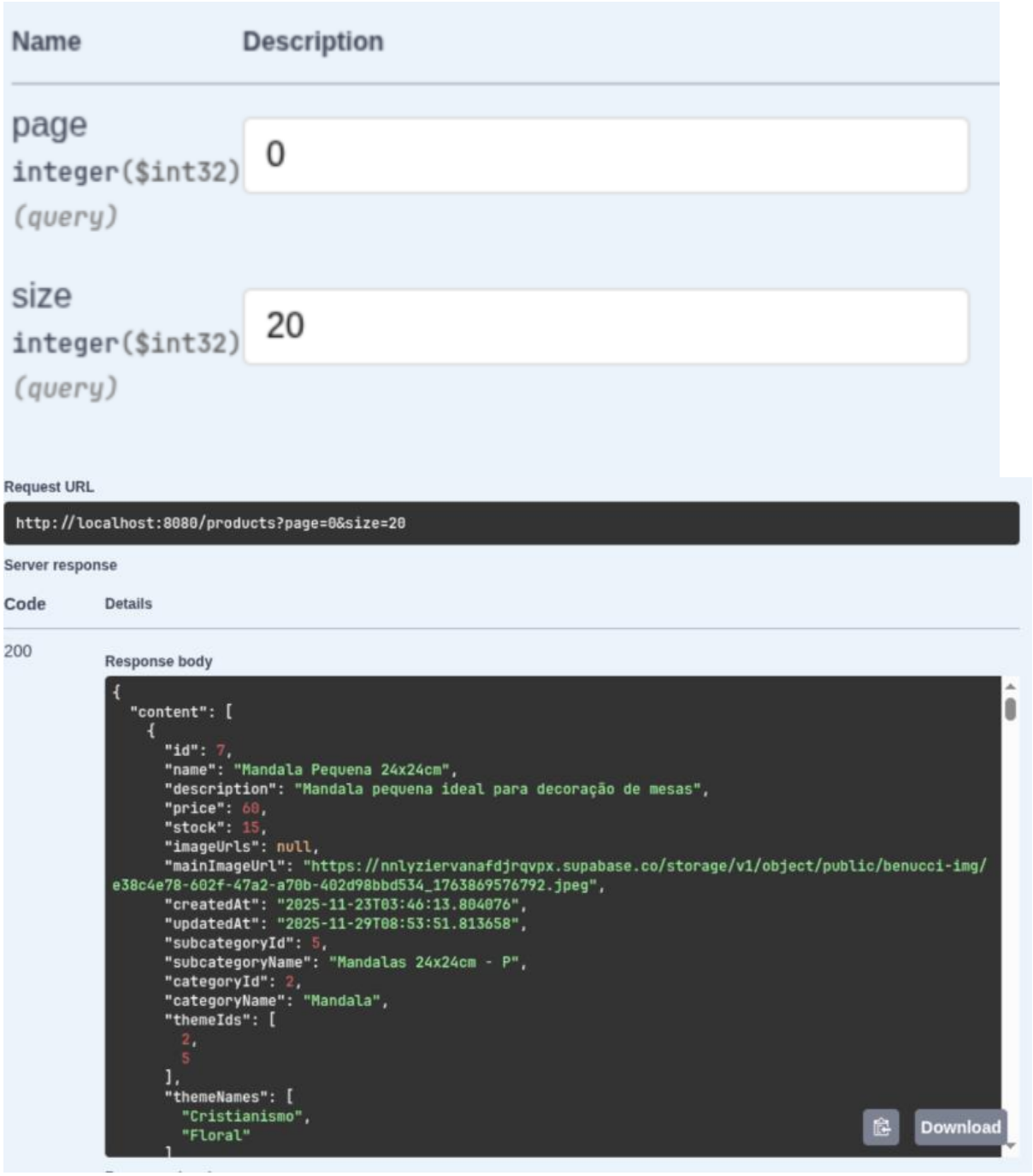
```
 "id": 13,
 "name": "Miniatura Religiosa de Santo",
 "description": "Miniatura artesanal com representação religiosa",
 "price": 80,
 "stock": 0,
 "imageUrls": [
  "https://nnlyziervanafdjrqvpx.supabase.co/storage/v1/object/public/benucci-img/1546c148-805d-40a5-
801c-2b453e1047e8_1763870042459.jpeg",
  "https://nnlyziervanafdjrqvpx.supabase.co/storage/v1/object/public/benucci-img/9a6a745c-da0e-4c35-
b743-1ec479897b82_1763870041189.jpeg"
 ],
 "mainImageUrl": "https://nnlyziervanafdjrqvpx.supabase.co/storage/v1/object/public/benucci-
img/1546c148-805d-40a5-801c-2b453e1047e8_1763870042459.jpeg",
 "createdAt": "2025-11-23T03:54:00.121239",
 "updatedAt": "2025-11-25T04:55:04.570977",
 "subcategoryId": 8,
 "subcategoryName": "Miniatura Religiosa",
 "categoryId": 3,
 "categoryName": "Miniaturas",
 "themeIds": [
  1,
  2,
  3,
  4,
  5
 ],
 "themeNames": [
  "Umbanda",
  "Cristianismo",
  "Santo",
  "Natureza",
  "Floral"
 ]
}
```


Listar produtos com paginação:

• Método: GET

• Rota: /products?page=0&size=20

• Descrição: Retorna uma lista paginada de produtos cadastrados. Separa uma grande quantidade de dados em pedaços e só mostra o mesmo. A menos que mude de página.



Order-controller

Atualiza status do pedido:

• Método: PUT

• Rota: /order/status/{id}

• Descrição: Permite alterar o status de um pedido existente

Order-controller

Atualiza status do pedido:

• Método: PUT

• Rota: /order/status/{id}

• Descrição: Permite alterar o status de um pedido existente

3. Pedido (/orders)

Criar Pedido

• Método: POST

• Rota: /orders

• Descrição: Cria um pedido para um usuário específico com produtos e quantidade.

• Body (JSON):

{

"userId": 2,

"deliveryType": "delivery",

"deliveryAddress": "Rua das Flores, 123",

"items": [

{"productId": 1, "quantity": 2},

{"productId": 2, "quantity": 1}

]

}

Relacionamentos: Pedido está ligado a um usuário (userId) e produtos (items).

Listar Pedidos de um Usuário:

• Método: GET

• Rota: /orders/user/{userId}

• Descrição: Lista todos os pedidos feitos por um usuário.

Detalhes de um Pedido

• Método: GET

• Rota: /orders/{orderId}

• Descrição: Retorna os detalhes completos do pedido.

Atualizar Status de Pedido:

• Método: PUT

• Rota: /orders/status/{orderId}

• Descrição: Atualiza o status do pedido (preparing, shipped, delivered, etc.).

• Body (JSON):

{

"status": "preparing"

}

Criar um novo pedido:

• Método: POST

• Rota: /orders

• Descrição:

• Body (JSON):

```
"userId": 0,
"deliveryType": "string",
"deliveryAddress": "string",
"paymentMethod": "string",
"items": [
 {
  "productId": 0,
  "quantity": 0
 }
]
}
```

Consultar pedido por id:

• Método: POST

• Rota: /orders/{id}

• Descrição: Retorna detalhes de um pedido específico pelo seu ID

```
"id": 0,
"orderDate": "2025-11-29T18:32:30.337Z",
"totalAmount": 0,
"deliveryType": "string",
"deliveryAddress": "string",
"status": "string",
"items": [
 {
  "productId": 0,
  "productName": "string",
  "quantity": 0,
  "unitPrice": 0
 }
]
}
```


Listar pedidos de um usuario:

• Método: GET

• Rota: /orders/users/{userId}

• Descrição: Retorna detalhes de um pedido específico pelo id do usuario


```
{
 "id": 0,
 "orderDate": "2025-11-29T18:33:19.742Z",
 "totalAmount": 0,
 "deliveryType": "string",
 "deliveryAddress": "string",
 "status": "string",
 "items": [
  {
   "productId": 0,
   "productName": "string",
   "quantity": 0,
   "unitPrice": 0
  }
 ]
}
]
```


sub-category-controller

PUT:

/categories

atualizar Subcategoria

DELETE:

/categories

Excluir Subcategoria

GET:

/categories

Listar Subcategorias de uma Categoria

POST:

/categories

Criar Subcategoria

4. Pagamento (/api/payments)

Criar Preferência de Pagamento

• Método: POST

• Rota: /api/payments/preference

• Descrição: Cria uma preferência de pagamento, podendo escolher o método (ex: Mercado Pago).

• Body (JSON):

```
"userId": 0,
"deliveryType": "string",
"deliveryAddress": "string",
"paymentMethod": "string",
"items": [
 {
  "productId": 0,
  "quantity": 0
 }
]
```

5. Categoria (/categories)

Listar Categorias:

• Método: GET

• Rota: /categories

• Descrição: Lista todas as categorias de produtos.

Criar Categoria:

• Método: POST

• Rota: /categories

• Descrição: Adiciona uma nova categoria.

• Body (JSON):

{

"name": "Bijuterias",

"description": "Acessórios e bijuterias artesanais"

}

9.4. Perguntas Frequentes (FAQ)

1. Por que não consigo logar?

- Email ou senha incorretos
- Usuário não registrado
- Token expirado (necessário login novamente)

2. O app não carrega os produtos. O que fazer?

- Verificar conexão com internet
- Checar se a API está ativa na porta configurada
- Verificar se o IP da API está corretamente configurado no mobile

3. Não consigo cadastrar produto, por quê?

Apenas usuários do tipo admin têm permissão.

4. O pedido não muda de status.

Apenas o admin responsável pelo produto pode alterar o status.

5. Os dados são seguros?

Sim.
A API utiliza:

- JWT para autenticação
- BCrypt para armazenar senhas
- Filtros de autorização no backend

10. GERENCIAMENTO DO PROJETO

O gerenciamento do projeto **Benucci Artes** foi conduzido utilizando uma abordagem **ágil**, priorizando a flexibilidade, entregas incrementais e *feedback* contínuo.

10.1. Metodologia Utilizada (Scrum Adaptado)

A metodologia de desenvolvimento foi **inspirada no *Scrum***, mas adaptada à rotina acadêmica e ao tamanho da equipe.

Ritual/Prática	Frequência	Objetivo
Sprints	Semanal	Definição clara de tarefas e objetivos a serem cumpridos no ciclo.
Daily Stand-up	Diária (ou Semi-Diária)	Reuniões rápidas de alinhamento para identificar o que foi feito, o que será feito e quais são os impedimentos (bloqueadores).
Backlog Priorizado	Contínuo	Lista de funcionalidades e tarefas organizadas por valor de negócio e complexidade.
Sprint Review	Ao final da Sprint	Demonstração das funcionalidades concluídas ao avaliador, garantindo o alinhamento e o *feedback*.
Retrospectiva	Variável	Análise do ciclo para identificar pontos fortes, gargalos (*bottlenecks*) e oportunidades de melhoria.

Essa metodologia garantiu a entrega de valor de forma contínua, permitindo ajustes rápidos no escopo do projeto.

10.2. Cronograma e Marcos do Projeto

O cronograma foi organizado em fases sequenciais, com duração de **10 semanas**, garantindo que as entregas fossem graduais e fáceis de acompanhar.

Fase	Período (Exemplo Semanas)	Marcos de Entrega / Objetivos
1 — Planejamento	Semana 1 – 2	Definição do escopo, modelagem de dados (DER/UML) e *setup* inicial do ambiente.
2 — Backend Base	Semana 3 – 4	Configuração da API, Autenticação JWT e criação dos *endpoints* CRUD de User e Product.
3 — App Mobile Base	Semana 4 – 6	Estrutura inicial do React Native, Telas de Login/Cadastro e **Integração inicial** com a API.
4 — Funcionalidades	Semana 6 – 8	Implementação do fluxo completo de Pedidos, Gerenciamento de Estado Global (Context API/Zustand), UI/UX.
5 — Refino e QA	Semana 8 – 9	**Testes manuais** de fluxo de ponta a ponta, Correção de *bugs* e melhorias de performance (Logs e Caching).
6 — Entrega Final	Semana 10	**Revisão final** do código, Documentação Técnica e preparação da apresentação.

10.3. Ferramentas de Gerenciamento de Projeto

Ferramenta	Propósito Principal	Detalhes Técnicos
GitHub Projects	Gerenciamento de Tarefas (Kanban)	Board com colunas: *To Do*, *In Progress*, *Code Review* e *Done*. **Issues** vinculadas a *commits* e *Pull Requests*.
Git & GitHub	Versionamento e Colaboração	Uso de **Feature Branches** (branches por funcionalidade), e **Pull Requests (PRs)** para revisão de código antes do *merge*.

Postman / Swagger	Teste e Documentação de API	Planejamento e testes dos contratos dos *endpoints* e registro dos fluxos (autenticação, pedidos).
Notion / Google Docs	Documentação	Criação e edição colaborativa do documento técnico, *checklists* de entregáveis e anotações.
WhatsApp / Discord	Comunicação Rápida	Alinhamentos diários e comunicação de bloqueios (*impediments*).

10.4. Gestão de Riscos

Os principais riscos identificados foram monitorados e mitigados através de práticas ágeis:

Risco Identificado	Nível	Estratégia de Mitigação
Atrasos no Desenvolvimento	Alto	**Sprints curtas** (semanais) e tarefas menores, com *dailies* para reavaliação constante da carga de trabalho.
Problemas de Integração	Médio	**Definição de Contrato** de API prévia e testes contínuos no Postman (Contrato *First*).
Falhas de Segurança	Médio	Implementação de **JWT/BCrypt** baseada em padrões de mercado, e uso de filtros de autorização do Spring Security.
Conflitos de Código (Merge)	Baixo/Médio	Exigência de **Branches por Funcionalidade** e revisão de código via **Pull Requests**.
Inconsistência no Design	Baixo	Definição de um **Design System** simples (cores, fontes e componentes) para uso em todo o app.