Descrição dos Testes

Este documento descreve todos os testes automatizados do projeto CardShop.

Testes Unitários

AuthMiddleware

tests/unit/auth.middleware.test.ts

Função/Recurso testado: `verifyToken`

O que testa: Verifica que o middleware de autenticação rejeita requisições que não possuem o header `Authorization`, retornando a mensagem "Acesso negado. Token não fornecido."

Código HTTP esperado: 401

Resultado esperado: Requisição bloqueada com `success: false`, `next()` não é chamado.

Função/Recurso testado: `verifyToken`

O que testa: Verifica que o middleware rejeita tokens enviados sem o prefixo "Bearer" (ex: `Authorization: Basic some-token`), retornando a mensagem "Formato do token inválido."

Código HTTP esperado: 401

Resultado esperado: Requisição bloqueada com `success: false`, `next()` não é chamado.

Função/Recurso testado: `verifyToken`

O que testa: Verifica que o middleware rejeita quando o header contém `Bearer ` seguido de string vazia.

Código HTTP esperado: 401

Resultado esperado: Requisição bloqueada com `success: false`.

Função/Recurso testado: `verifyToken`

O que testa: Verifica que o middleware captura erro do `jwt.verify` quando o token é malformado ou expirado.

Código HTTP esperado: 401

Resultado esperado: Requisição bloqueada com mensagem "Token inválido ou expirado."

Função/Recurso testado: `verifyToken`

O que testa: Verifica que, ao receber um token JWT válido no header `Authorization: Bearer <token>`, o middleware decodifica o token e injeta `userId` (42) e `userRole` ("CUSTOMER") no objeto da requisição.

Resultado esperado: `next()` é chamado uma vez, `req.userId` igual a 42 e `req.userRole` igual a "CUSTOMER".

Função/Recurso testado: `isAdmin`

O que testa: Verifica que o middleware `isAdmin` permite a passagem de usuários com role "ADMIN".

Resultado esperado: `next()` é chamado uma vez, `res.status` não é chamado.

Função/Recurso testado: `isAdmin`

O que testa: Verifica que o middleware `isAdmin` bloqueia usuários com role "CUSTOMER", retornando a mensagem "Acesso negado. Requer privilégios de Administrador."

Código HTTP esperado: 403

Resultado esperado: Requisição bloqueada com `success: false`, `next()` não é chamado.

RateLimitMiddleware

tests/unit/rateLimit.middleware.test.ts

Função/Recurso testado: `strict limiter (autenticação)`

O que testa: Verifica que o limiter restrito (usado em rotas de autenticação) permite requisições quando o número de chamadas está dentro do limite configurado (3).

Código HTTP esperado: 200

Resultado esperado: Requisição processada com sucesso.

Função/Recurso testado: `strict limiter (autenticação)`

O que testa: Verifica que, após enviar 3 requisições (o limite configurado) para a rota de login, a próxima requisição é bloqueada pelo rate limiter.

Código HTTP esperado: 429

Resultado esperado: Requisição bloqueada por excesso de tentativas.

Função/Recurso testado: `strict limiter (autenticação)`

O que testa: Verifica que a resposta de rate limit contém o corpo no formato padrão de erro da API: `{ error: { message: "Muitas tentativas de autenticação...", status: 429 } }`.

Código HTTP esperado: 429

Resultado esperado: Corpo da resposta segue o formato de erro padrão com mensagem em português.

Função/Recurso testado: `strict limiter (autenticação)`

O que testa: Verifica que a resposta 429 inclui o header `retry-after`, informando ao cliente quando pode tentar novamente.

Código HTTP esperado: 429

Resultado esperado: Header `retry-after` presente na resposta.

Função/Recurso testado: `global limiter`

O que testa: Verifica que o limiter global (aplicado a todas as rotas `/api/`) permite requisições dentro do limite configurado (3).

Código HTTP esperado: 200

Resultado esperado: Requisição processada com sucesso.

Função/Recurso testado: `global limiter`

O que testa: Verifica que, após exceder o limite global de requisições, a API retorna erro 429 com a mensagem "Muitas requisições. Por favor, tente novamente mais tarde."

Código HTTP esperado: 429

Resultado esperado: Requisição bloqueada com corpo de erro no formato padrão.

ValidationMiddleware

tests/unit/validation.middleware.test.ts

Função/Recurso testado: `validateId`

O que testa: Verifica que o middleware rejeita IDs não numéricos (ex: "abc") nos parâmetros da rota, retornando a mensagem "ID inválido".

Código HTTP esperado: 400

Resultado esperado: Requisição bloqueada com `success: false`, `next()` não é chamado.

Função/Recurso testado: `validateId`

O que testa: Verifica que o middleware permite a passagem quando o ID é um número válido (ex: "42").

Resultado esperado: `next()` é chamado uma vez, `res.status` não é chamado.

Função/Recurso testado: `validateBody`

O que testa: Verifica que o middleware rejeita requisições com corpo vazio (`{}`), retornando a mensagem "Corpo da requisição não pode estar vazio".

Código HTTP esperado: 400

Resultado esperado: Requisição bloqueada com `success: false`, `next()` não é chamado.

Função/Recurso testado: `validateRequiredFields`

O que testa: Verifica que, ao exigir os campos ["name", "email", "password", "cpf"] e receber apenas "name", o middleware retorna a lista dos campos faltantes: ["email", "password", "cpf"].

Código HTTP esperado: 400

Resultado esperado: Resposta com `success: false`, `message: "Campos obrigatórios ausentes"` e `missingFields` contendo os campos ausentes.

UserService

tests/unit/user.service.test.ts

Função/Recurso testado: `registerNewUser`

O que testa: Verifica que o serviço cadastra um novo usuário corretamente: verifica que o email não está em uso, faz hash da senha com bcrypt e chama o repositório para criar o usuário.

Resultado esperado: Usuário criado com os dados fornecidos, senha armazenada como hash ("hashed_password").

Função/Recurso testado: `registerNewUser`

O que testa: Verifica que o serviço impede o cadastro de um usuário com email já existente no banco de dados.

Resultado esperado: Erro lançado com mensagem "Erro ao cadastrar usuário: E-mail já registrado."

Função/Recurso testado: `findUserById`

O que testa: Verifica que o serviço rejeita buscas sem userId (valor 0), lançando erro de validação.

Resultado esperado: Erro lançado com mensagem "Erro ao buscar perfil: userId é obrigatório".

Função/Recurso testado: `findUserById`

O que testa: Verifica que erros do repositório propagam corretamente (sem wrapping, pois findUserById retorna sem await).

Resultado esperado: Erro lançado com a mensagem original do repositório ("DB error").

Função/Recurso testado: `findUserById`

O que testa: Verifica que o serviço retorna os dados de um usuário ao buscar por ID, e que o campo `password` não é incluído na resposta.

Resultado esperado: Usuário com `id: 1`, `email: "teste@email.com"`, `name: "Teste"`, sem a propriedade `password`.

Função/Recurso testado: `updateUserProfile`

O que testa: Verifica que o serviço atualiza dados do perfil (nome, email) sem alterar a senha quando ela não é fornecida.

Resultado esperado: Usuário atualizado com novos dados, repositório chamado sem campo password.

Função/Recurso testado: `updateUserProfile`

O que testa: Verifica que, quando uma nova senha é fornecida, o serviço faz hash antes de enviar ao repositório.

Resultado esperado: Repositório recebe senha como hash ("new_hashed_password"), não como texto plano.

Função/Recurso testado: `updateUserProfile`

O que testa: Verifica que o serviço lança erro quando tenta atualizar perfil sem userId (valor 0).

Resultado esperado: Erro lançado com mensagem "Erro ao atualizar perfil: userId é obrigatório".

Função/Recurso testado: `updateUserProfile`

O que testa: Verifica que o serviço lança erro quando tenta atualizar perfil de usuário inexistente.

Resultado esperado: Erro lançado com mensagem "Erro ao atualizar perfil: Usuário não encontrado".

Função/Recurso testado: `deleteUser`

O que testa: Verifica que o serviço lança erro quando tenta deletar sem userId (valor 0).

Resultado esperado: Erro lançado com mensagem "Erro ao deletar usuário: userId é obrigatório".

Função/Recurso testado: `deleteUser`

O que testa: Verifica que o serviço lança erro quando tenta deletar usuário inexistente.

Resultado esperado: Erro lançado com mensagem "Erro ao deletar usuário: Usuário não encontrado".

Função/Recurso testado: `deleteUser`

O que testa: Verifica que o serviço deleta um usuário existente, primeiro verificando sua existência e depois chamando o repositório para remoção.

Resultado esperado: Retorna os dados do usuário deletado com `id: 1` e `email: "teste@email.com"`.

Função/Recurso testado: `authenticateUser`

O que testa: Verifica que o serviço autentica um usuário com email e senha corretos, retornando um token JWT e os dados do usuário sem o campo `password`.

Resultado esperado: Objeto com propriedade `token` (string) e propriedade `user` contendo `email: "teste@email.com"` e sem `password`.

Função/Recurso testado: `authenticateUser`

O que testa: Verifica que o serviço rejeita a autenticação quando a senha fornecida não corresponde ao hash armazenado (bcrypt.compare retorna false).

Resultado esperado: Erro lançado com mensagem "Erro ao autenticar usuário: Credenciais inválidas."

ProductService

tests/unit/product.service.test.ts

Função/Recurso testado: `getAllProducts`

O que testa: Verifica que erros do repositório são propagados corretamente pelo serviço.

Resultado esperado: Erro lançado com mensagem "Erro ao buscar produtos: DB error".

Função/Recurso testado: `getAllProducts`

O que testa: Verifica que o serviço de produtos retorna uma lista de produtos ao aplicar filtros de jogo ("yugioh") e tipo de carta ("MONSTER"), delegando a busca ao repositório.

Resultado esperado: Array com 2 produtos, o primeiro com nome "Dark Magician" e o segundo com nome "Blue-Eyes White Dragon".

Função/Recurso testado: `getProductById`

O que testa: Verifica que o serviço lança erro quando tenta buscar um produto sem ID (string vazia).

Resultado esperado: Erro lançado com mensagem "Erro ao buscar produto: ID do produto é obrigatório".

Função/Recurso testado: `getProductById`

O que testa: Verifica que erros do repositório são propagados pelo serviço ao buscar por ID.

Resultado esperado: Erro lançado com mensagem "Erro ao buscar produto: DB error".

Função/Recurso testado: `getProductById`

O que testa: Verifica que o serviço retorna um produto específico ao buscar por ID, com todos os campos corretos.

Resultado esperado: Produto com `id: 1`, `name: "Dark Magician"` e `game: "yugioh"`.

Função/Recurso testado: `createProduct`

O que testa: Verifica que o serviço cria um novo produto após validar os dados e verificar que o nome não está duplicado.

Resultado esperado: Produto criado com nome "Dark Magician", preço 29.99, game "yugioh".

Função/Recurso testado: `createProduct`

O que testa: Verifica que o serviço impede a criação de produto com nome já existente no banco.

Resultado esperado: Erro lançado com mensagem contendo "Já existe um produto com este nome".

Função/Recurso testado: `createProduct`

O que testa: Verifica que o serviço lança erro ao tentar criar produto com dados incompletos (apenas campo "name").

Resultado esperado: Erro lançado com mensagem contendo "Erro ao criar produto".

Função/Recurso testado: `updateProduct`

O que testa: Verifica que o serviço atualiza os dados de um produto existente com validações de preço e estoque.

Resultado esperado: Produto atualizado com novo preço (39.99).

Função/Recurso testado: `updateProduct`

O que testa: Verifica que o serviço retorna null ao tentar atualizar um produto inexistente.

Resultado esperado: Retorna null.

Função/Recurso testado: `updateProduct`

O que testa: Verifica que o serviço rejeita atualizações com objeto vazio (`{}`).

Resultado esperado: Erro lançado com mensagem contendo "Dados de atualização são obrigatórios".

Função/Recurso testado: `updateProduct`

O que testa: Verifica que o serviço rejeita atualizações com preço negativo.

Resultado esperado: Erro lançado com mensagem contendo "Preço não pode ser negativo".

Função/Recurso testado: `updateProduct`

O que testa: Verifica que o serviço rejeita atualizações com estoque negativo (-1).

Resultado esperado: Erro lançado com mensagem contendo "Estoque não pode ser negativo".

Função/Recurso testado: `updateProduct`

O que testa: Verifica que o serviço rejeita a atualização do tipo de carta quando ele é incompatível com o jogo (ex: "CREATURE" para "yugioh").

Resultado esperado: Erro lançado com mensagem contendo "Tipo de carta inválido para Yugioh".

Função/Recurso testado: `deleteProduct`

O que testa: Verifica que o serviço deleta um produto existente do repositório.

Resultado esperado: Retorna true.

Função/Recurso testado: `deleteProduct`

O que testa: Verifica que o serviço retorna null ao tentar deletar um produto inexistente.

Resultado esperado: Retorna null.

Função/Recurso testado: `deleteProduct`

O que testa: Verifica que, quando o produto está referenciado em pedidos ou carrinhos (erro "Foreign key constraint"), o serviço lança erro adequado.

Resultado esperado: Erro lançado com mensagem contendo "em uso".

Função/Recurso testado: `deleteProduct`

O que testa: Verifica que erros desconhecidos do repositório ao deletar são propagados com prefixo de contexto.

Resultado esperado: Erro lançado com mensagem contendo "Erro ao deletar produto".

Função/Recurso testado: `validateProductData`

O que testa: Verifica que a validação de dados do produto aceita um objeto com todos os campos obrigatórios preenchidos corretamente (name, price, stock, game).

Resultado esperado: Nenhum erro é lançado.

Função/Recurso testado: `validateProductData`

O que testa: Verifica que a validação lança erro quando o campo `price` está ausente no objeto do produto.

Resultado esperado: Erro lançado com mensagem "Campo price é obrigatório".

Função/Recurso testado: `validateProductData`

O que testa: Verifica que a validação rejeita produtos com preço negativo (-5).

Resultado esperado: Erro lançado com mensagem "Preço não pode ser negativo".

Função/Recurso testado: `validateProductData`

O que testa: Verifica que a validação rejeita produtos com estoque negativo (-1).

Resultado esperado: Erro lançado com mensagem "Estoque não pode ser negativo".

Função/Recurso testado: `validateProductData`

O que testa: Verifica que a validação rejeita games fora do enum (ex: "pokemon").

Resultado esperado: Erro lançado com mensagem "Jogo inválido. Opções: mtg, yugioh".

Função/Recurso testado: `validateProductData`

O que testa: Verifica que a validação lança erro quando o campo `name` está ausente no objeto do produto.

Resultado esperado: Erro lançado com mensagem "Campo name é obrigatório".

Função/Recurso testado: `validateProductData`

O que testa: Verifica que a validação lança erro quando o campo `stock` está ausente no objeto do produto.

Resultado esperado: Erro lançado com mensagem "Campo stock é obrigatório".

Função/Recurso testado: `validateProductData`

O que testa: Verifica que a validação lança erro quando o campo `game` está ausente no objeto do produto.

Resultado esperado: Erro lançado com mensagem "Campo game é obrigatório".

Função/Recurso testado: `validateProductData`

O que testa: Verifica que a validação aceita um cardType válido ("MONSTER") quando associado ao jogo correto ("yugioh").

Resultado esperado: Nenhum erro é lançado.

Função/Recurso testado: `validateCardTypeForGame`

O que testa: Verifica que o tipo de carta "MONSTER" é aceito como válido para o jogo "yugioh".

Resultado esperado: Nenhum erro é lançado.

Função/Recurso testado: `validateCardTypeForGame`

O que testa: Verifica que o tipo de carta "SPELL" é aceito como válido para o jogo "yugioh".

Resultado esperado: Nenhum erro é lançado.

Função/Recurso testado: `validateCardTypeForGame`

O que testa: Verifica que o tipo de carta "TRAP" é aceito como válido para o jogo "yugioh".

Resultado esperado: Nenhum erro é lançado.

Função/Recurso testado: `validateCardTypeForGame`

O que testa: Verifica que o tipo de carta "CREATURE" é aceito como válido para o jogo "mtg".

Resultado esperado: Nenhum erro é lançado.

Função/Recurso testado: `validateCardTypeForGame`

O que testa: Verifica que o tipo de carta "INSTANT" é aceito como válido para o jogo "mtg".

Resultado esperado: Nenhum erro é lançado.

Função/Recurso testado: `validateCardTypeForGame`

O que testa: Verifica que o tipo de carta "PLANESWALKER" é aceito como válido para o jogo "mtg".

Resultado esperado: Nenhum erro é lançado.

Função/Recurso testado: `validateCardTypeForGame`

O que testa: Verifica que o tipo de carta "CREATURE" (exclusivo de MTG) é rejeitado quando associado ao jogo "yugioh".

Resultado esperado: Erro lançado com mensagem "Tipo de carta inválido para Yugioh".

Função/Recurso testado: `validateCardTypeForGame`

O que testa: Verifica que o tipo de carta "MONSTER" (exclusivo de Yu-Gi-Oh!) é rejeitado quando associado ao jogo "mtg".

Resultado esperado: Erro lançado com mensagem "Tipo de carta inválido para Magic the Gathering".

CartService

tests/unit/cart.service.test.ts

Função/Recurso testado: `getCart`

O que testa: Verifica que o serviço retorna o carrinho de um usuário existente, incluindo seus itens com quantidade e dados do produto.

Resultado esperado: Carrinho com `id: 1`, `userId: 1`, contendo 1 item com `quantity: 2`.

Função/Recurso testado: `getCart`

O que testa: Verifica que o serviço lança erro quando tenta buscar carrinho sem userId (valor 0).

Resultado esperado: Erro lançado com mensagem "Erro ao buscar carrinho: userId é obrigatório".

Função/Recurso testado: `getCart`

O que testa: Verifica que o serviço lança erro quando o usuário não é encontrado no banco (prisma.user.findUnique retorna null).

Resultado esperado: Erro lançado com mensagem contendo "Usuário não encontrado".

Função/Recurso testado: `addToCart`

O que testa: Verifica que, quando o produto não está no carrinho, o serviço cria ou busca o carrinho e adiciona o item.

Resultado esperado: Carrinho retornado com o novo item adicionado.

Função/Recurso testado: `addToCart`

O que testa: Verifica que, quando o produto já está no carrinho, a quantidade é atualizada (somada) em vez de criar novo item.

Resultado esperado: Item atualizado com quantidade somada (2 + 2 = 4).

Função/Recurso testado: `addToCart`

O que testa: Verifica que o serviço lança erro quando tenta adicionar ao carrinho sem userId (valor 0).

Resultado esperado: Erro lançado com mensagem "Erro ao adicionar ao carrinho: userId é obrigatório".

Função/Recurso testado: `addToCart`

O que testa: Verifica que o serviço lança erro quando tenta adicionar ao carrinho sem productId (valor 0).

Resultado esperado: Erro lançado com mensagem "Erro ao adicionar ao carrinho: productId é obrigatório".

Função/Recurso testado: `addToCart`

O que testa: Verifica que o serviço rejeita adições ao carrinho com quantidade 0 ou negativa.

Resultado esperado: Erro lançado com mensagem "Erro ao adicionar ao carrinho: Quantidade deve ser no mínimo 1".

Função/Recurso testado: `addToCart`

O que testa: Verifica que o serviço lança erro ao tentar adicionar produto inexistente ao carrinho.

Resultado esperado: Erro lançado com mensagem contendo "Produto não encontrado".

Função/Recurso testado: `addToCart`

O que testa: Verifica que o serviço lança erro ao tentar adicionar ao carrinho de um usuário inexistente.

Resultado esperado: Erro lançado com mensagem contendo "Usuário não encontrado".

Função/Recurso testado: `addToCart`

O que testa: Verifica que, ao tentar adicionar unidades de um produto que excederia o estoque disponível (3 no carrinho + 5 solicitado > 5 em estoque), o serviço lança erro.

Resultado esperado: Erro lançado com mensagem contendo "Estoque insuficiente".

Função/Recurso testado: `updateQuantity`

O que testa: Verifica que o serviço atualiza a quantidade de um item no carrinho quando há estoque suficiente.

Resultado esperado: Carrinho retornado com quantidade atualizada para 5.

Função/Recurso testado: `updateQuantity`

O que testa: Verifica que o serviço lança erro quando tenta atualizar quantidade sem userId (valor 0).

Resultado esperado: Erro lançado com mensagem "Erro ao atualizar quantidade: userId é obrigatório".

Função/Recurso testado: `updateQuantity`

O que testa: Verifica que o serviço lança erro quando tenta atualizar quantidade sem productId (valor 0).

Resultado esperado: Erro lançado com mensagem "Erro ao atualizar quantidade: productId é obrigatório".

Função/Recurso testado: `updateQuantity`

O que testa: Verifica que o serviço rejeita atualizações de quantidade com valor 0 ou negativo.

Resultado esperado: Erro lançado com mensagem "Erro ao atualizar quantidade: Quantidade deve ser no mínimo 1".

Função/Recurso testado: `updateQuantity`

O que testa: Verifica que o serviço lança erro quando tenta atualizar item de carrinho inexistente.

Resultado esperado: Erro lançado com mensagem contendo "Carrinho não encontrado".

Função/Recurso testado: `updateQuantity`

O que testa: Verifica que o serviço lança erro ao atualizar item que não existe no carrinho.

Resultado esperado: Erro lançado com mensagem contendo "Item não encontrado no carrinho".

Função/Recurso testado: `updateQuantity`

O que testa: Verifica que o serviço lança erro quando o produto associado ao item do carrinho não é encontrado no banco.

Resultado esperado: Erro lançado com mensagem contendo "Produto não encontrado".

Função/Recurso testado: `updateQuantity`

O que testa: Verifica que o serviço rejeita atualização de quantidade quando excede o estoque do produto (5 solicitado > 3 em estoque).

Resultado esperado: Erro lançado com mensagem contendo "Estoque insuficiente".

Função/Recurso testado: `removeFromCart`

O que testa: Verifica que um item existente é removido do carrinho corretamente, retornando o carrinho atualizado sem itens.

Resultado esperado: Carrinho retornado com array de itens vazio (length 0).

Função/Recurso testado: `removeFromCart`

O que testa: Verifica que o serviço lança erro quando tenta remover item sem userId (valor 0).

Resultado esperado: Erro lançado com mensagem "Erro ao remover item: userId é obrigatório".

Função/Recurso testado: `removeFromCart`

O que testa: Verifica que o serviço lança erro quando tenta remover item sem productId (valor 0).

Resultado esperado: Erro lançado com mensagem "Erro ao remover item: productId é obrigatório".

Função/Recurso testado: `removeFromCart`

O que testa: Verifica que o serviço lança erro ao tentar remover item de carrinho inexistente.
Resultado esperado: Erro lançado com mensagem contendo "Carrinho não encontrado".

Função/Recurso testado: `removeFromCart`

O que testa: Verifica que o serviço lança erro ao tentar remover item que não existe no carrinho.
Resultado esperado: Erro lançado com mensagem contendo "Item não encontrado no carrinho".

Função/Recurso testado: `clearCart`

O que testa: Verifica que todos os itens do carrinho são removidos de uma vez.
Resultado esperado: Carrinho retornado vazio, repositório clearCart chamado.

Função/Recurso testado: `clearCart`

O que testa: Verifica que o serviço lança erro quando tenta limpar carrinho sem userId (valor 0).
Resultado esperado: Erro lançado com mensagem "Erro ao limpar carrinho: userId é obrigatório".

Função/Recurso testado: `clearCart`

O que testa: Verifica que o serviço lança erro ao tentar limpar carrinho inexistente.
Resultado esperado: Erro lançado com mensagem contendo "Carrinho não encontrado".

OrderService

tests/unit/order.service.test.ts

Função/Recurso testado: `getAllOrders`

O que testa: Verifica que o serviço retorna todos os pedidos filtrados por status "PENDING", delegando ao repositório.
Resultado esperado: Array com 2 pedidos, ambos com `status: "PENDING"`.

Função/Recurso testado: `getAllOrders`

O que testa: Verifica que erros do repositório são propagados corretamente pelo serviço.
Resultado esperado: Erro lançado com mensagem "Erro ao buscar pedidos: DB error".

Função/Recurso testado: `getOrderById`

O que testa: Verifica que o serviço retorna um pedido específico com seus itens ao buscar por ID.

Resultado esperado: Pedido com `id: 1`, `status: "PENDING"`, `totalPrice: 150.5` e 1 item na lista.

Função/Recurso testado: `getOrderById`

O que testa: Verifica que o serviço lança erro quando tenta buscar pedido sem orderId (valor 0).

Resultado esperado: Erro lançado com mensagem "Erro ao buscar pedido: orderId é obrigatório".

Função/Recurso testado: `getOrdersByUser`

O que testa: Verifica que o serviço retorna todos os pedidos de um usuário específico.

Resultado esperado: Array de pedidos do usuário.

Função/Recurso testado: `getOrdersByUser`

O que testa: Verifica que o serviço lança erro quando tenta buscar pedidos sem userId (valor 0).

Resultado esperado: Erro lançado com mensagem "Erro ao buscar pedidos: userId é obrigatório".

Função/Recurso testado: `getOrdersByUser`

O que testa: Verifica que o serviço valida a existência do usuário antes de buscar pedidos.

Resultado esperado: Erro lançado com mensagem contendo "Usuário não encontrado".

Função/Recurso testado: `createOrder`

O que testa: Verifica que o serviço cria um pedido completo usando `prisma.$transaction()`: calcula total dos itens, cria o pedido, decrementa estoque e limpa o carrinho.

Resultado esperado: Pedido criado com itens, total correto e carrinho limpo.

Função/Recurso testado: `createOrder`

O que testa: Verifica que o serviço lança erro ao tentar criar pedido sem userId (valor 0).

Resultado esperado: Erro lançado com mensagem "Erro ao criar pedido: userId é obrigatório".

Função/Recurso testado: `createOrder`

O que testa: Verifica que o campo shippingAddress é obrigatório para criação de pedidos (string vazia).

Resultado esperado: Erro lançado com mensagem "Erro ao criar pedido: Endereço de entrega é obrigatório".

Função/Recurso testado: `createOrder`

O que testa: Verifica que o serviço rejeita endereço contendo apenas espaços em branco (" ").

Resultado esperado: Erro lançado com mensagem contendo "Endereço de entrega é obrigatório".

Função/Recurso testado: `createOrder`

O que testa: Verifica que o serviço valida a existência do usuário antes de criar o pedido.

Resultado esperado: Erro lançado com mensagem contendo "Usuário não encontrado".

Função/Recurso testado: `createOrder`

O que testa: Verifica que o serviço impede a criação de pedidos quando o carrinho do usuário está vazio (sem itens).

Resultado esperado: Erro lançado com mensagem "Erro ao criar pedido: Carrinho está vazio".

Função/Recurso testado: `createOrder`

O que testa: Verifica que o serviço lança erro quando o carrinho do usuário não existe (null).

Resultado esperado: Erro lançado com mensagem contendo "Carrinho está vazio".

Função/Recurso testado: `createOrder`

O que testa: Verifica que o serviço valida o estoque de cada item antes de criar o pedido, rejeitando quando quantidade excede estoque (20 solicitado > 5 em estoque).

Resultado esperado: Erro lançado com mensagem contendo "Estoque insuficiente".

Função/Recurso testado: `updateOrderStatus`

O que testa: Verifica que a máquina de estados permite a transição de status de "PENDING" para "PROCESSING" (transição válida).

Resultado esperado: Pedido atualizado com `status: "PROCESSING"`.

Função/Recurso testado: `updateOrderStatus`

O que testa: Verifica que a transição PROCESSING -> SHIPPED é aceita pela máquina de estados.

Resultado esperado: Pedido atualizado com `status: "SHIPPED"`.

Função/Recurso testado: `updateOrderStatus`

O que testa: Verifica que pedidos pendentes podem ser cancelados (transição válida).

Resultado esperado: Pedido atualizado com `status: "CANCELLED"`.

Função/Recurso testado: `updateOrderStatus`

O que testa: Verifica que o serviço lança erro ao tentar atualizar status sem orderId (valor 0).

Resultado esperado: Erro lançado com mensagem "Erro ao atualizar status: orderId é obrigatório".

Função/Recurso testado: `updateOrderStatus`

O que testa: Verifica que o serviço lança erro ao tentar atualizar com status vazio (string vazia).

Resultado esperado: Erro lançado com mensagem "Erro ao atualizar status: status é obrigatório".

Função/Recurso testado: `updateOrderStatus`

O que testa: Verifica que o serviço rejeita status fora do enum (ex: "INVALID").

Resultado esperado: Erro lançado com mensagem contendo "Status inválido".

Função/Recurso testado: `updateOrderStatus`

O que testa: Verifica que o serviço lança erro ao tentar atualizar status de pedido inexistente.

Resultado esperado: Erro lançado com mensagem contendo "Pedido não encontrado".

Função/Recurso testado: `updateOrderStatus`

O que testa: Verifica que a máquina de estados bloqueia a transição de "DELIVERED" para "CANCELLED", pois pedidos entregues não podem ser cancelados.

Resultado esperado: Erro lançado com mensagem "Transição inválida: DELIVERED → CANCELLED. Permitidas: nenhuma".

Função/Recurso testado: `updateOrderStatus`

O que testa: Verifica que a máquina de estados bloqueia a transição de "SHIPPED" para "CANCELLED", pois pedidos enviados só podem avançar para "DELIVERED".

Resultado esperado: Erro lançado com mensagem "Transição inválida: SHIPPED → CANCELLED. Permitidas: DELIVERED".

Função/Recurso testado: `updateOrderStatus`

O que testa: Verifica que o serviço lança erro quando o repositório retorna null ao tentar atualizar o status.

Resultado esperado: Erro lançado com mensagem contendo "Erro ao atualizar pedido".

Função/Recurso testado: `deleteOrder`

O que testa: Verifica que o serviço deleta um pedido existente do repositório.
Resultado esperado: Retorna true indicando sucesso.

Função/Recurso testado: `deleteOrder`

O que testa: Verifica que o serviço retorna false quando tenta deletar pedido inexistente.
Resultado esperado: Retorna false.

Função/Recurso testado: `deleteOrder`

O que testa: Verifica que o serviço lança erro ao tentar deletar pedido sem orderId (valor 0).
Resultado esperado: Erro lançado com mensagem "Erro ao deletar pedido: orderId é obrigatório".

UserRepository

tests/unit/user.repository.test.ts

Função/Recurso testado: `create`

O que testa: Verifica que o repositório chama `prisma.user.create` com os dados corretos e retorna o usuário criado.
Resultado esperado: Usuário criado com id, name, email, cpf e role "CUSTOMER".

Função/Recurso testado: `findByEmail`

O que testa: Verifica que o repositório busca por email e inclui o campo password (necessário para autenticação).
Resultado esperado: Usuário com email e propriedade `password` presente.

Função/Recurso testado: `findByEmail`

O que testa: Verifica que o repositório retorna null quando nenhum usuário é encontrado com o email fornecido.
Resultado esperado: Retorna null.

Função/Recurso testado: `findById`

O que testa: Verifica que o repositório busca por ID e exclui o campo password da resposta (segurança).
Resultado esperado: Usuário com id e email, sem propriedade `password`.

Função/Recurso testado: `findById`

O que testa: Verifica que o repositório retorna null quando nenhum usuário é encontrado com o ID fornecido.
Resultado esperado: Retorna null.

Função/Recurso testado: `update`
O que testa: Verifica que o repositório chama `prisma.user.update` com where/data corretos.
Resultado esperado: Usuário atualizado com novos dados.

Função/Recurso testado: `delete`
O que testa: Verifica que o repositório chama `prisma.user.delete` com o ID correto.
Resultado esperado: Usuário deletado retornado.

ProductRepository

tests/unit/product.repository.test.ts

Função/Recurso testado: `findAll`
O que testa: Verifica que o repositório retorna todos os produtos quando nenhum filtro é aplicado.
Resultado esperado: Array com 2 produtos.

Função/Recurso testado: `findAll`
O que testa: Verifica que o repositório aplica o filtro `where: { game }` corretamente no Prisma.
Resultado esperado: Prisma chamado com filtro de game "yugioh", retorna 1 produto.

Função/Recurso testado: `findAll`
O que testa: Verifica que o repositório aplica o filtro `where: { cardType }` corretamente no Prisma.
Resultado esperado: Prisma chamado com filtro de cardType "MONSTER", retorna 1 produto.

Função/Recurso testado: `findById`
O que testa: Verifica que o repositório converte ID string para inteiro antes de consultar o Prisma.
Resultado esperado: Produto com id correto, Prisma chamado com `{ where: { id: 1 } }`.

Função/Recurso testado: `findById`
O que testa: Verifica que o repositório retorna null quando nenhum produto é encontrado com o ID fornecido.
Resultado esperado: Retorna null.

Função/Recurso testado: `findByName`

O que testa: Verifica que o repositório busca produto pelo nome usando `prisma.product.findFirst`.

Resultado esperado: Produto com `name: "Dark Magician"`.

Função/Recurso testado: `findByName`

O que testa: Verifica que o repositório retorna null quando nenhum produto é encontrado com o nome fornecido.

Resultado esperado: Retorna null.

Função/Recurso testado: `create`

O que testa: Verifica que o repositório chama `prisma.product.create` com todos os campos (name, price, stock, game, cardType).

Resultado esperado: Produto criado com os dados fornecidos.

Função/Recurso testado: `update`

O que testa: Verifica que o repositório chama `prisma.product.update` com where/data corretos.

Resultado esperado: Produto atualizado com novo preço (39.99).

Função/Recurso testado: `update`

O que testa: Verifica que o repositório retorna null ao tentar atualizar um produto inexistente.

Resultado esperado: Retorna null.

Função/Recurso testado: `delete`

O que testa: Verifica que o repositório chama `prisma.product.delete` com o ID correto.

Resultado esperado: `prisma.product.delete` chamado uma vez.

CartRepository

tests/unit/cart.repository.test.ts

Função/Recurso testado: `findByUserId`

O que testa: Verifica que o repositório busca o carrinho com include de itens e dados do produto associado.

Resultado esperado: Carrinho com userId e items incluindo dados do produto.

Função/Recurso testado: `findByUserId`

O que testa: Verifica que o repositório retorna null quando nenhum carrinho é encontrado para o userId.

Resultado esperado: Retorna null.

Função/Recurso testado: `findOrCreateByUserId`

O que testa: Verifica que, quando o carrinho já existe, o repositório retorna sem criar novo.

Resultado esperado: Carrinho existente retornado.

Função/Recurso testado: `findOrCreateByUserId`

O que testa: Verifica que, quando não existe carrinho, o repositório cria um novo e retorna.

Resultado esperado: Novo carrinho criado com userId correto.

Função/Recurso testado: `findCartItem`

O que testa: Verifica que o repositório busca item pelo composite key (cartId + productId) usando `prisma.cartItem.findUnique`.

Resultado esperado: Item com `quantity: 2`.

Função/Recurso testado: `findCartItem`

O que testa: Verifica que o repositório retorna null quando nenhum item é encontrado com o composite key.

Resultado esperado: Retorna null.

Função/Recurso testado: `addItem`

O que testa: Verifica que o repositório chama `prisma.cartItem.create` com cartId, productId e quantity.

Resultado esperado: Item criado no carrinho com `quantity: 3`.

Função/Recurso testado: `updateItemQuantity`

O que testa: Verifica que o repositório chama `prisma.cartItem.update` para atualizar a quantidade de um item existente.

Resultado esperado: Item atualizado com `quantity: 5`.

Função/Recurso testado: `removeItem`

O que testa: Verifica que o repositório remove o item pelo composite key (cartId + productId) e retorna true.

Resultado esperado: Retorna true.

Função/Recurso testado: `removeItem`

O que testa: Verifica que o repositório retorna false (sem propagar erro) quando a remoção falha (item não encontrado).

Resultado esperado: Retorna false.

Função/Recurso testado: `clearCart`

O que testa: Verifica que o repositório remove todos os itens do carrinho via `deleteMany`.

Resultado esperado: Retorna true.

Função/Recurso testado: `clearCart`

O que testa: Verifica que o repositório retorna false (sem propagar erro) quando a limpeza do carrinho falha.

Resultado esperado: Retorna false.

OrderRepository

tests/unit/order.repository.test.ts

Função/Recurso testado: `findAll`

O que testa: Verifica que o repositório retorna todos os pedidos com include de itens e dados do produto.

Resultado esperado: Array com 2 pedidos incluindo items.

Função/Recurso testado: `findAll`

O que testa: Verifica que o repositório aplica o filtro de status ao buscar pedidos.

Resultado esperado: Array com 1 pedido filtrado por status "PENDING".

Função/Recurso testado: `findAll`

O que testa: Verifica que o repositório aplica o filtro de userId ao buscar pedidos.

Resultado esperado: Array com 1 pedido filtrado por userId 1.

Função/Recurso testado: `findById`

O que testa: Verifica que o repositório busca pedido com items e product incluídos.

Resultado esperado: Pedido com id, userId, status e items.

Função/Recurso testado: `findById`

O que testa: Verifica que o repositório retorna null quando nenhum pedido é encontrado com o ID fornecido.

Resultado esperado: Retorna null.

Função/Recurso testado: `findByUserId`

O que testa: Verifica que o repositório filtra pedidos por userId com include de items.

Resultado esperado: Array com 2 pedidos do usuário específico.

Função/Recurso testado: `findByUserId`

O que testa: Verifica que o repositório retorna array vazio quando o usuário não possui pedidos.

Resultado esperado: Array vazio (length 0).

Função/Recurso testado: `create`

O que testa: Verifica que o repositório usa `prisma.order.create` com nested `items: { create: [...] }` para criar pedido e itens atomicamente.

Resultado esperado: Pedido criado com itens associados.

Função/Recurso testado: `updateStatus`

O que testa: Verifica que o repositório chama `prisma.order.update` para alterar o status.

Resultado esperado: Pedido com status atualizado para "PROCESSING".

Função/Recurso testado: `updateStatus`

O que testa: Verifica que o repositório retorna null ao tentar atualizar status de pedido inexistente.

Resultado esperado: Retorna null.

Função/Recurso testado: `delete`

O que testa: Verifica que o repositório chama `prisma.order.delete` com o ID correto.

Resultado esperado: `prisma.order.delete` chamado uma vez.

Testes de Integração

Rotas da API

tests/integration/routes.test.ts

Rota testada: `GET /api/products`

O que testa: Verifica que a rota GET /api/products retorna a lista de produtos corretamente, passando por todas as camadas (rota -> controller -> service -> repositório stubado).

Código HTTP esperado: 200

Resultado esperado: Resposta com `success: true` e `data` sendo um array com 1 produto.

Rota testada: `GET /api/products/:id`

O que testa: Verifica que a rota GET /api/products/1 retorna um produto existente com todos os campos.

Código HTTP esperado: 200

Resultado esperado: Resposta com `success: true`, `data.id` igual a 1, `data.name` igual a "Dark Magician".

Rota testada: `GET /api/products/:id`

O que testa: Verifica que a rota GET /api/products/999 retorna erro 404 quando o produto não é encontrado no banco de dados.

Código HTTP esperado: 404

Resultado esperado: Resposta com `success: false` e `message: "Produto não encontrado"`.

Rota testada: `POST /api/products`

O que testa: Verifica que um usuário ADMIN consegue criar um novo produto enviando todos os campos obrigatórios.

Código HTTP esperado: 201

Resultado esperado: Resposta com `success: true` e dados do produto criado.

Rota testada: `POST /api/products`

O que testa: Verifica que a rota bloqueia usuários com role CUSTOMER de criar produtos (rota admin-only).

Código HTTP esperado: 403

Resultado esperado: Acesso negado.

Rota testada: `POST /api/products`

O que testa: Verifica que a rota bloqueia requisições sem token de autenticação.

Código HTTP esperado: 401

Resultado esperado: Acesso negado.

Rota testada: `PUT /api/products/:id`

O que testa: Verifica que um admin consegue atualizar os dados de um produto existente.

Código HTTP esperado: 200

Resultado esperado: Resposta com `success: true`.

Rota testada: `PUT /api/products/:id`

O que testa: Verifica que a rota retorna 404 ao tentar atualizar produto inexistente.

Código HTTP esperado: 404

Resultado esperado: Produto não encontrado.

Rota testada: `PUT /api/products/:id`

O que testa: Verifica que a rota bloqueia usuários CUSTOMER de atualizar produtos.

Código HTTP esperado: 403

Resultado esperado: Acesso negado.

Rota testada: `DELETE /api/products/:id`

O que testa: Verifica que um admin consegue deletar um produto existente.

Código HTTP esperado: 200

Resultado esperado: Resposta com `success: true`.

Rota testada: `DELETE /api/products/:id`

O que testa: Verifica que a rota retorna 404 ao tentar deletar produto inexistente.

Código HTTP esperado: 404

Resultado esperado: Produto não encontrado.

Rota testada: `DELETE /api/products/:id`

O que testa: Verifica que deletar um produto referenciado em pedidos/carrinhos retorna erro de conflito (foreign key).

Código HTTP esperado: 409

Resultado esperado: Resposta com mensagem indicando que produto está sendo referenciado.

Rota testada: `DELETE /api/products/:id`

O que testa: Verifica que a rota bloqueia usuários CUSTOMER de deletar produtos.

Código HTTP esperado: 403

Resultado esperado: Acesso negado.

Rota testada: `POST /api/users/register`

O que testa: Verifica que a rota POST /api/users/register cria um novo usuário quando todos os campos obrigatórios (name, email, password, cpf) são enviados corretamente.

Código HTTP esperado: 201

Resultado esperado: Resposta com `success: true` e `data.email` igual a "teste@email.com".

Rota testada: `POST /api/users/register`

O que testa: Verifica que a rota POST /api/users/register rejeita a requisição quando apenas o campo "name" é enviado, sem email, password e cpf.

Código HTTP esperado: 400

Resultado esperado: Resposta com `success: false` e `missingFields` contendo ["email", "password", "cpf"].

Rota testada: `POST /api/users/register`

O que testa: Verifica que a rota retorna conflito ao tentar cadastrar email duplicado.

Código HTTP esperado: 409

Resultado esperado: Resposta com `success: false`.

Rota testada: `POST /api/users/login`

O que testa: Verifica que a rota POST /api/users/login retorna um token JWT e dados do usuário ao fornecer credenciais válidas.

Código HTTP esperado: 200

Resultado esperado: Resposta com `success: true`, `data.token` e `data.user.email`.

Rota testada: `POST /api/users/login`

O que testa: Verifica que a rota POST /api/users/login retorna erro de autenticação quando as credenciais fornecidas são inválidas.

Código HTTP esperado: 401

Resultado esperado: Resposta com `success: false`.

Rota testada: `POST /api/users/login`

O que testa: Verifica que a rota rejeita a requisição quando o campo "password" não é enviado.

Código HTTP esperado: 400

Resultado esperado: Resposta com `success: false` e `missingFields` contendo "password".

Rota testada: `GET /api/users/profile`
O que testa: Verifica que a rota retorna os dados do perfil do usuário logado usando o token JWT.
Código HTTP esperado: 200
Resultado esperado: Resposta com `success: true` e `data.email` igual a "teste@email.com".

Rota testada: `GET /api/users/profile`
O que testa: Verifica que acessar o perfil sem token de autenticação é bloqueado.
Código HTTP esperado: 401
Resultado esperado: Acesso negado.

Rota testada: `GET /api/users/profile`
O que testa: Verifica que a rota retorna 404 quando o serviço retorna null para o userId do token.
Código HTTP esperado: 404
Resultado esperado: Resposta com `success: false`.

Rota testada: `GET /api/users/profile`
O que testa: Verifica que erros não tratados no serviço resultam em resposta 500.
Código HTTP esperado: 500
Resultado esperado: Resposta com `success: false`.

Rota testada: `PATCH /api/users/profile`
O que testa: Verifica que a rota permite atualizar dados do perfil (nome) com token válido.
Código HTTP esperado: 200
Resultado esperado: Resposta com `success: true` e `data.name` igual a "Novo Nome".

Rota testada: `PATCH /api/users/profile`
O que testa: Verifica que atualizar o perfil sem token de autenticação é bloqueado.
Código HTTP esperado: 401
Resultado esperado: Acesso negado.

Rota testada: `PATCH /api/users/profile`
O que testa: Verifica que a rota rejeita requisições de atualização com corpo vazio (`{}`).
Código HTTP esperado: 400
Resultado esperado: Resposta com `success: false`.

Rota testada: `PATCH /api/users/profile`
O que testa: Verifica que a rota retorna conflito ao tentar atualizar para um email já em uso.
Código HTTP esperado: 409
Resultado esperado: Resposta com `success: false`.

Rota testada: `POST /api/users/logout`
O que testa: Verifica que a rota de logout retorna sucesso quando o usuário está autenticado.
Código HTTP esperado: 200
Resultado esperado: Resposta com `success: true` e `message: "Logout realizado com sucesso"`.

Rota testada: `POST /api/users/logout`
O que testa: Verifica que a rota de logout bloqueia requisições sem token de autenticação.
Código HTTP esperado: 401
Resultado esperado: Acesso negado.

Rota testada: `DELETE /api/users/:id`
O que testa: Verifica que um admin pode deletar qualquer usuário pelo ID.
Código HTTP esperado: 200
Resultado esperado: Resposta com `success: true`.

Rota testada: `DELETE /api/users/:id`
O que testa: Verifica que a rota rejeita IDs não numéricos (ex: "abc").
Código HTTP esperado: 400
Resultado esperado: Resposta com `success: false`.

Rota testada: `DELETE /api/users/:id`
O que testa: Verifica que a rota bloqueia usuários não-admin de deletar outros usuários.
Código HTTP esperado: 403
Resultado esperado: Acesso negado.

Rota testada: `DELETE /api/users/:id`
O que testa: Verifica que a rota retorna 404 ao tentar deletar um usuário inexistente.
Código HTTP esperado: 404
Resultado esperado: Resposta com `success: false`.

Rota testada: `DELETE /api/users/:id`

O que testa: Verifica que a rota bloqueia requisições sem token de autenticação.

Código HTTP esperado: 401

Resultado esperado: Acesso negado.

Rota testada: `GET /api/cart`

O que testa: Verifica que a rota retorna o carrinho do usuário autenticado com seus itens.

Código HTTP esperado: 200

Resultado esperado: Resposta com `success: true` e dados do carrinho.

Rota testada: `GET /api/cart`

O que testa: Verifica que acessar o carrinho sem token é bloqueado.

Código HTTP esperado: 401

Resultado esperado: Acesso negado.

Rota testada: `GET /api/cart`

O que testa: Verifica que um CUSTOMER não pode acessar o carrinho de outro usuário via query parameter `userId`.

Código HTTP esperado: 403

Resultado esperado: Resposta com `success: false`.

Rota testada: `POST /api/cart/items`

O que testa: Verifica que a rota adiciona um item ao carrinho do usuário autenticado.

Código HTTP esperado: 200

Resultado esperado: Resposta com `success: true` e carrinho atualizado.

Rota testada: `POST /api/cart/items`

O que testa: Verifica que a rota rejeita requisições sem o campo productId.

Código HTTP esperado: 400

Resultado esperado: Resposta com `success: false` e `message: "productId é obrigatório"`.

Rota testada: `POST /api/cart/items`

O que testa: Verifica que adicionar item ao carrinho sem token é bloqueado.

Código HTTP esperado: 401

Resultado esperado: Acesso negado.

Rota testada: `PUT /api/cart/items/:productId`
O que testa: Verifica que a rota atualiza a quantidade de um item no carrinho.
Código HTTP esperado: 200
Resultado esperado: Resposta com `success: true` e carrinho atualizado.

Rota testada: `PUT /api/cart/items/:productId`
O que testa: Verifica que a rota rejeita productId não numérico (ex: "abc").
Código HTTP esperado: 400
Resultado esperado: Resposta com `message: "productId inválido"`.

Rota testada: `PUT /api/cart/items/:productId`
O que testa: Verifica que a rota rejeita quantidade igual a 0.
Código HTTP esperado: 400
Resultado esperado: Resposta com `message: "Quantidade deve ser no mínimo 1"`.

Rota testada: `PUT /api/cart/items/:productId`
O que testa: Verifica que atualizar quantidade sem token é bloqueado.
Código HTTP esperado: 401
Resultado esperado: Acesso negado.

Rota testada: `DELETE /api/cart/items/:productId`
O que testa: Verifica que a rota remove um item específico do carrinho.
Código HTTP esperado: 200
Resultado esperado: Resposta com `success: true` e carrinho atualizado.

Rota testada: `DELETE /api/cart/items/:productId`
O que testa: Verifica que a rota rejeita productId não numérico (ex: "abc").
Código HTTP esperado: 400
Resultado esperado: Resposta com `message: "productId inválido"`.

Rota testada: `DELETE /api/cart/items/:productId`
O que testa: Verifica que remover item sem token é bloqueado.
Código HTTP esperado: 401
Resultado esperado: Acesso negado.

Rota testada: `DELETE /api/cart`

O que testa: Verifica que a rota remove todos os itens do carrinho do usuário.

Código HTTP esperado: 200

Resultado esperado: Resposta com `success: true`.

Rota testada: `DELETE /api/cart`

O que testa: Verifica que limpar o carrinho sem token é bloqueado.

Código HTTP esperado: 401

Resultado esperado: Acesso negado.

Rota testada: `GET /api/orders`

O que testa: Verifica que a rota retorna a lista de pedidos do usuário autenticado.

Código HTTP esperado: 200

Resultado esperado: Resposta com `success: true` e `data` sendo um array.

Rota testada: `GET /api/orders`

O que testa: Verifica que listar pedidos sem token é bloqueado.

Código HTTP esperado: 401

Resultado esperado: Acesso negado.

Rota testada: `POST /api/orders`

O que testa: Verifica que a rota cria um pedido com endereço de entrega, processando o carrinho do usuário.

Código HTTP esperado: 201

Resultado esperado: Resposta com `success: true` e dados do pedido criado.

Rota testada: `POST /api/orders`

O que testa: Verifica que a rota rejeita criação de pedido sem endereço de entrega.

Código HTTP esperado: 400

Resultado esperado: Resposta com `message: "Endereço de entrega é obrigatório"`.

Rota testada: `POST /api/orders`

O que testa: Verifica que criar pedido sem token é bloqueado.

Código HTTP esperado: 401

Resultado esperado: Acesso negado.

Rota testada: `GET /api/orders/:id`

O que testa: Verifica que a rota retorna detalhes de um pedido pertencente ao usuário autenticado.
Código HTTP esperado: 200
Resultado esperado: Resposta com `success: true` e dados do pedido.

Rota testada: `GET /api/orders/:id`

O que testa: Verifica que a rota rejeita IDs não numéricos (ex: "abc").
Código HTTP esperado: 400
Resultado esperado: Resposta com `message: "ID do pedido inválido"`.

Rota testada: `GET /api/orders/:id`

O que testa: Verifica que a rota retorna 404 quando o pedido não é encontrado.
Código HTTP esperado: 404
Resultado esperado: Pedido não encontrado.

Rota testada: `GET /api/orders/:id`

O que testa: Verifica que um CUSTOMER não pode acessar pedidos de outros usuários (isolamento de dados).
Código HTTP esperado: 403
Resultado esperado: Acesso negado.

Rota testada: `GET /api/orders/:id`

O que testa: Verifica que buscar pedido por ID sem token é bloqueado.
Código HTTP esperado: 401
Resultado esperado: Acesso negado.

Rota testada: `GET /api/orders/all`

O que testa: Verifica que a rota /all retorna todos os pedidos do sistema para usuários admin.
Código HTTP esperado: 200
Resultado esperado: Resposta com `success: true` e array de todos os pedidos.

Rota testada: `GET /api/orders/all`

O que testa: Verifica que a rota /all é restrita a administradores.
Código HTTP esperado: 403
Resultado esperado: Acesso negado.

Rota testada: `GET /api/orders/all`

O que testa: Verifica que acessar /all sem token é bloqueado.

Código HTTP esperado: 401

Resultado esperado: Acesso negado.

Rota testada: `PATCH /api/orders/:id/status`

O que testa: Verifica que um admin pode atualizar o status de um pedido (ex: PENDING -> PROCESSING).

Código HTTP esperado: 200

Resultado esperado: Resposta com `success: true` e pedido com novo status.

Rota testada: `PATCH /api/orders/:id/status`

O que testa: Verifica que a rota rejeita IDs não numéricos (ex: "abc").

Código HTTP esperado: 400

Resultado esperado: ID inválido.

Rota testada: `PATCH /api/orders/:id/status`

O que testa: Verifica que a rota rejeita requisições sem o campo status no corpo.

Código HTTP esperado: 400

Resultado esperado: Resposta com `message: "status é obrigatório"`.

Rota testada: `PATCH /api/orders/:id/status`

O que testa: Verifica que a rota bloqueia usuários CUSTOMER de atualizar status de pedidos.

Código HTTP esperado: 403

Resultado esperado: Acesso negado.

Rota testada: `DELETE /api/orders/:id`

O que testa: Verifica que um admin pode deletar um pedido pelo ID.

Código HTTP esperado: 200

Resultado esperado: Resposta com `success: true`.

Rota testada: `DELETE /api/orders/:id`

O que testa: Verifica que a rota rejeita IDs não numéricos (ex: "abc").

Código HTTP esperado: 400

Resultado esperado: ID inválido.

Rota testada: `DELETE /api/orders/:id`

O que testa: Verifica que deletar um pedido inexistente retorna erro 404.

Código HTTP esperado: 404

Resultado esperado: Resposta com `success: false`.

Rota testada: `DELETE /api/orders/:id`

O que testa: Verifica que a rota bloqueia usuários CUSTOMER de deletar pedidos.

Código HTTP esperado: 403

Resultado esperado: Acesso negado.

Rota testada: `DELETE /api/orders/:id`

O que testa: Verifica que deletar pedido sem token é bloqueado.

Código HTTP esperado: 401

Resultado esperado: Acesso negado.

Rota testada: `Rota inexistente`

O que testa: Verifica que o handler global de 404 do Express captura requisições para rotas inexistentes.

Código HTTP esperado: 404

Resultado esperado: Resposta com mensagem "Rota não encontrada".

Rota testada: `GET /api-docs.json`

O que testa: Verifica que a rota /api-docs.json retorna a especificação OpenAPI/Swagger completa da API.

Código HTTP esperado: 200

Resultado esperado: JSON com propriedade `openapi`.

Rota testada: `Error handler global`

O que testa: Verifica que o error handler global do Express captura erros não tratados nos controllers e retorna 500.

Código HTTP esperado: 500

Resultado esperado: Resposta com `error.message` presente.
