

1. Contexto do Projeto

O sistema backend implementa uma API para processar pagamentos via PIX utilizando o Mercado Pago como intermediário. A estrutura está organizada em:

- **Modelos (models):** para representar entidades no banco, como Usuário (`user.js`) e Pagamento (`pagamento.js`).
 - **Serviços (services):** onde fica a lógica de negócio, incluindo integração com Mercado Pago (`pagamentoService.js`).
 - **Controladores (controllers):** que recebem as requisições HTTP e chamam os serviços (`pagamentoController.js`).
 - **Testes automatizados (tests):** para garantir que a API está funcionando conforme o esperado (`pagamento.test.js`).
-

2. Dependências e Ambiente para o Teste

Para rodar os testes com Jest e Supertest, foram necessárias as seguintes dependências instaladas via `npm`:

- **jest:** framework para testes unitários e integração.
- **supertest:** biblioteca para simular requisições HTTP ao servidor Express.
- **express:** framework web para criar rotas HTTP na API.
- **uuid:** para gerar IDs únicos (usado no serviço de pagamento).
- **sequelize:** ORM para manipulação do banco de dados (mockado nos testes).
- **mercadopago** (configurado em `config/mercadopago.js`): SDK oficial para integração com Mercado Pago (mockado nos testes).

O comando para instalar as dependências básicas para teste seria algo assim:

```
npm install --save-dev jest supertest
```

```
npm install express uuid sequelize mercadopago
```

No `package.json`:

```
"scripts": {  
  "test": "jest"  
}
```

3. Objetivo do Teste

O teste visa garantir que a rota **POST** `/api/pagamento`:

- Aceita uma requisição com valor de pagamento via header.
 - Cria um pagamento com o usuário autenticado.
 - Retorna corretamente o link do boleto PIX (`ticketUrl`) e a imagem em base64 do QR code.
-

4. Análise Detalhada do Código do Teste

4.1 Configuração do servidor Express para teste

```
const app = express();  
app.use(express.json());  
app.post("/api/pagamento", mockAuthMiddleware, pagamentoController.criarPagamento);
```

- Cria uma instância do Express.
- Usa um middleware de autenticação falso `mockAuthMiddleware` para simular usuário logado.
- Registra a rota de criação de pagamento.

4.2 Middleware de autenticação fake

```
const mockAuthMiddleware = (req, res, next) => {  
  req.usuario = { id: "user-123", email: "teste@exemplo.com" };  
  next();  
};
```

Injeta um objeto `usuario` no `req`, simulando usuário autenticado, sem passar pela autenticação real.

4.3 Mocks das dependências

Para isolar o teste e evitar dependências externas:

```
jest.mock("../models/user.js", () => ({  
  findByPk: jest.fn().mockResolvedValue({ id: "user-123", email: "teste@exemplo.com" }),  
}));
```

```
jest.mock("../models/pagamento.js", () => ({  
  create: jest.fn().mockResolvedValue({  
    usuarioid: "user-123",  
    valor: 150,  
    status: "approved",  
    referenciaExterna: "fake-id",  
    metodoPagamento: "pix",  
    paymentIdMP: "fake-payment-id",  
    qrCode: "fake-qr-code",  
    qrCodeBase64: "fake-base64",  
    ticketUrl: "https://fake-pix-url.com/",  
  }),  
}));
```

```
jest.mock("../services/pagamentoService.js", () => ({  
  criarPagamento: jest.fn().mockResolvedValue({  
    ticketUrl: "https://fake-pix-url.com/",  
    qrCodeBase64: "fake-base64",  
  }),  
  getPagamento: jest.fn(),  
}));
```

- **Model User:** simula o retorno da busca de usuário por ID, retornando um usuário fixo.

- **Model Pagamento:** simula a criação de pagamento no banco, retornando dados fake.
- **Service Pagamento:** simula a criação de pagamento com Mercado Pago, retornando ticket URL e QR Code base64 fixos.

4.4 Execução do teste

```
describe("Pagamento", () => {  
  it("deve criar um pagamento com sucesso", async () => {  
    const response = await request(app)  
      .post("/api/pagamento")  
      .set("value", "150");  
  
    expect(response.status).toBe(201);  
    expect(response.body.ticketUrl).toContain("https://fake-pix-url.com/");  
    expect(response.body.qrCodeBase64).toBe("fake-base64");  
  });  
});
```

- Envia um POST para `/api/pagamento` com header `value = 150`.
- Espera status HTTP 201 (criado).
- Espera o corpo da resposta JSON conter:
 - `ticketUrl` com link para o boleto PIX (URL fake)
 - `qrCodeBase64` com imagem em base64 (fake)

5. Fluxo Interno Simulado Durante o Teste

1. O request chega no controller `criarPagamento`.

2. O controller lê o `usuarioId` do `req.usuario` (mockado).
 3. O controller lê o valor do header `value`.
 4. Chama `pagamentoService.criarPagamento(usuarioId, valor)` (mockado).
 5. O service retorna os dados fake do pagamento com `ticketUrl` e `qrCodeBase64`.
 6. O controller responde HTTP 201 com esses dados.
 7. O teste valida a resposta conforme esperado.
-

6. Resultados e Respostas Esperadas

- **Status HTTP:** 201 Created
- **Corpo JSON:**

json
CopiarEditar

```
{  
  "ticketUrl": "https://fake-pix-url.com/",  
  "qrCodeBase64": "fake-base64"  
}
```

Este formato corresponde ao que seria retornado na vida real, contendo o link para pagamento e o QR Code para o cliente realizar o PIX.

7. Arquivos Envolvidos

- **models/user.js:** definição do modelo Usuário (mockado no teste).
 - **models/pagamento.js:** modelo de pagamento (mockado).
 - **services/pagamentoService.js:** lógica que cria pagamentos e chama Mercado Pago (mockada).
 - **controllers/pagamentoController.js:** controller que recebe as requisições e usa o service.
 - **tests/pagamento.test.js:** teste automatizado que simula toda a operação.
-

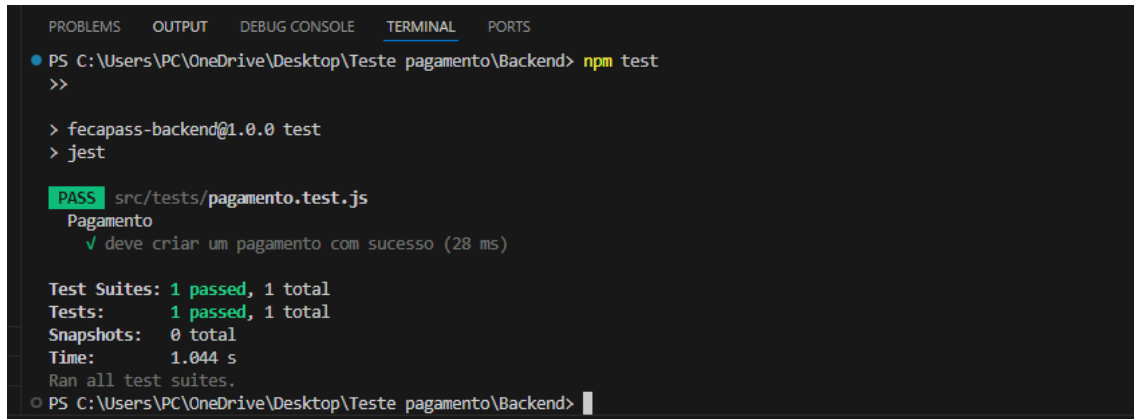
8. Como Rodar o Teste

1. Certifique-se de ter o Node.js e npm instalados.
2. No terminal, no diretório do projeto, instale as dependências com:

npm install

Execute os testes com: npm test

O Jest executará o arquivo de teste, exibindo se passou ou falhou.



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
● PS C:\Users\PC\OneDrive\Desktop\Teste pagamento\Backend> npm test
>>

> fecapass-backend@1.0.0 test
> jest

PASS src/tests/pagamento.test.js
  Pagamento
    ✓ deve criar um pagamento com sucesso (28 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.044 s
Ran all test suites.
○ PS C:\Users\PC\OneDrive\Desktop\Teste pagamento\Backend>
```

9. Por que usar Mocks?

- **Isolamento:** evitar dependência do banco ou do Mercado Pago em testes.
- **Velocidade:** testes rápidos sem acesso a serviços externos.
- **Confiabilidade:** resultados previsíveis, facilitando debug.