

Sumário

1. Introdução	4
1.1 Objetivo da Auditoria	4
1.2 Escopo Avaliado	4
1.3 Ferramentas Utilizadas	5
2. Auditoria das Métricas de Qualidade	6
2.1 Métricas de Código	7
2.2 Métricas de Desempenho	7
2.3 Métricas de Usabilidade	8
2.4 Métricas de Segurança	9
3. Análise dos Resultados	10
3.1 Funcionalidade e Regras de Negócio	10
3.2 Segurança	10
3.3 Desempenho e Teste de Carga	11
3.4 Usabilidade e Feedback ao Usuário	11
3.5 Complexidade Ciclométrica	12
3.6 Linhas de Código por Método	13
3.7 Síntese dos Resultados	13
4. Ações Corretivas e Melhorias Propostas	14
4.1 Melhorias na Usabilidade	14
4.2 Correções em Mensagens de Erro	14
4.3 Validações	14
4.4 Código e Arquitetura	14
4.5 Funcionalidade e Regras de Negócio	15
5. Conclusão	16
6. Evidências da Auditoria de Qualidade	17
6.1 M-01 – Complexidade Ciclométrica	17
6.2 M-03 – Linhas de Código por Método	22
6.3 M-04 – Tempo de Resposta	23
6.4 M-05 – Número de Reservas (Teste de Carga)	24
6.5 M-06, M-07 e M-08 – Usabilidade com Usuário	28
6.6 M-09 – Número de Vulnerabilidades Encontradas	29

6.7	M-10 – Tempo Médio para Correção de Falhas	34
6.8	M-11 – Execução de Auditorias de Segurança	34
6.9	Testes Extras	34
6.9.1	Login com Dados Válidos	34
6.9.2	Login com Email Não Existente.....	35
6.9.3	Login com Senha Incorreta.....	36
6.9.4	Acesso à Rota Protegida com Token Válido	37
6.9.5	Acesso à Rota Protegida sem Token	37
6.9.6	CRUD de Clientes	38
6.9.7	Cadastro de Usuários com Dados Inválidos	41
6.9.8	CRUD de Reservas	42
6.9.9	Verificação de Status de Reservas	46
6.9.10	Criar Reserva Fora do Expediente	46
6.9.11	Criar Reservas Duplicadas.....	47
6.9.12	Cadastro com CPF já existente	48

1. Introdução

1.1 Objetivo da Auditoria

Esta auditoria de qualidade tem como objetivo verificar se o sistema de reservas de mesas para um restaurante atende aos critérios de qualidade definidos previamente em seu planejamento, avaliando a efetividade e a concordância com as metas estabelecidas para as métricas de código, desempenho, segurança e usabilidade, descrevendo também possíveis mudanças feitas no planejamento e escopo. Através de uma análise crítica dos resultados dos testes, pretende-se identificar os pontos fortes, as fragilidades e as oportunidades de melhoria no sistema.

1.2 Escopo Avaliado

A auditoria foi realizada sobre os principais módulos do sistema, incluindo:

- **Módulo de Autenticação:** responsável pelo login de clientes e administradores com segurança via JWT.
- **Módulo de Cadastro de Clientes:** funcionalidades de criação, edição e exclusão de perfis.
- **Módulo de Reservas:** criação, edição, confirmação e cancelamento de reservas.
- **Módulo Administrativo:** visualização de reservas ativas, controle de disponibilidade e confirmação de agendamentos.

As funcionalidades analisadas estão diretamente ligadas aos principais Requisitos Funcionais e Requisitos Não Funcionais, com foco especial em desempenho sob carga, clareza nas mensagens de erro, segurança da autenticação e usabilidade da interface.

Atualmente, o front-end encontra-se finalizado, porém com possibilidade de receber alguns ajustes e/ou pequenas melhorias no futuro.

Do lado do back-end, o banco de dados está estruturado e operacional, e os principais CRUDs de usuários e reservas foram implementados e testados via Postman.

A conexão entre front-end e back-end está em andamento, com alguns fluxos já funcionando parcialmente, enquanto outros ainda estão sendo integrados ou corrigidos.

Dessa forma, a auditoria abrange os módulos prontos e funcionais no momento, concentrando-se nas rotas da API, validações, mensagens de erro e as telas já desenvolvidas da interface.

1.3 Ferramentas Utilizadas

Inicialmente, foi planejado o uso de uma variedade de ferramentas para apoiar a garantia de qualidade do sistema, no entanto, ao longo do desenvolvimento, algumas adaptações foram necessárias devido a mudanças do escopo, limitações técnicas e uma maior praticidade.

Como resultado, optou-se por uma abordagem mais enxuta e funcional, priorizando ferramentas acessíveis, leves e integráveis ao fluxo de desenvolvimento atual, centrado em Node.js com Express, autenticação JWT, e o banco de dados MySQL.

Nesse contexto, o Postman foi adotado como principal ferramenta para validação da maioria dos critérios de qualidade. O Postman se mostrou extremamente versátil não apenas para a realização de testes manuais e simulações de requisições, mas também para verificar tempo de resposta, analisar o comportamento da API em diferentes fluxos, validação dos tokens, etc.

Além disso, bibliotecas como bcrypt e jsonwebtoken foram incorporadas no backend para garantir boas práticas de segurança, incluindo criptografia de senhas e geração e validação de tokens de autenticação.

Assim, mesmo com a substituição de ferramentas mais robustas por alternativas práticas e funcionais, foi possível manter um padrão de validação e confiabilidade dos testes. Essa abordagem garantiu a continuidade do projeto sem comprometer a análise dos requisitos de qualidade, priorizando os testes e a sua validação, além de proporcionar um sistema funcional.

2. Auditoria das Métricas de Qualidade

Esta seção apresenta a auditoria das métricas de qualidade estabelecidas para o sistema, com base nas diretrizes definidas no planejamento de qualidade. A avaliação foi em geral realizada por meio de testes manuais na API utilizando o Postman, além de medições diretas no código-fonte e de observações praticas com usuarios para aspectos relacionados à usabilidade.

A tabela a seguir resume os resultados obtidos para cada métrica, acompanhados das respectivas ferramentas utilizadas, evidências reunidas, status atual e eventuais ações corretivas propostas.

	Métrica	Objetivo	Resultado	Ferramenta	Evidência	Status	Correções
M-01	Complexidade Ciclomática	<15	Sempre abaixo de 15	VS Code (extensão CodeMetrics)	Evidencia M-01	OK	—
M-02	Cobertura de Código	≥80%	Não avaliado	Não aplicada	—	Pendente	—
M-03	Linhas de Código por Método	≤50	Maior valor: 24 linhas	VS Code (análise manual)	Evidencia M-03	OK	—
M-04	Tempo de Resposta	<10secs	Todos inferiores a 800ms	Postman	Evidencia M-04	OK	—
M-05	Número de Reservas	≥30	>30 reservas cadastradas	Postman	Evidencia M-05	OK	—
M-06	Taxa de Sucesso nas Tarefas	90%	Valor atingido	Testes com usuários	Evidencia M-06	OK	—
M-07	Tempo Médio para Completar uma Tarefa	<3min	Média de ~40 segundos por tarefa	Testes com usuários	Evidencia M-07	OK	—
M-08	Número de Erros de Usuário	Mínimo	Observados problemas com ícones	Testes com usuários	Evidencia M-08	NOK	Melhorar ícones
M-09	Número de Vulnerabilidades Encontradas	Zero na entrega final	Nenhuma vulnerabilidade encontrada	Testes manuais (login, tokens)	Evidencia M-09	OK	—
M-10	Tempo Médio para Correção de Falhas	48hrs para bugs críticos	Falhou	Histórico de correções	Evidencia M-10	NOK	Corrigir erros identificados
M-11	Execução de Auditorias de Segurança	Mensalmente	1ª auditoria concluída	Revisão de código	Evidencia M-11	OK	—

2.1 Métricas de Código

As métricas de código têm como objetivo garantir que a base do sistema seja clara, bem estruturada e de fácil manutenção, tanto da parte do frontend quanto do backend. Elas promovem boas práticas de programação e facilitam a detecção de problemas estruturais mais rapidamente.

As diretrizes definidas para o projeto foram:

- **Complexidade Ciclométrica:** deve ser mantida abaixo de quinze.
- **Cobertura de Código:** pelo menos 80% do código deve ser coberto por testes unitários e de integração.
- **Linhas de Código por Método:** métodos devem conter, no máximo, 50 linhas cada, para manter legibilidade

Essas métricas foram consideradas durante o desenvolvimento do planejamento de qualidade do sistema, sendo utilizadas como diretrizes de qualidade de código. No entanto, até o momento da auditoria, não foi possível testá-las complementemente com ferramentas automatizadas como o SonarQube, devido a dificuldades e limitações técnicas.

Por esse motivo:

- A complexidade ciclométrica foi avaliada com o apoio da extensão CodeMetrics no Visual Studio Code.
- A métrica de linhas de código por método foi avaliada manualmente por meio da análise direta do código-fonte.
- A cobertura de código ainda não foi analisada, porém será considerada em algum outro momento, com o apoio de uma ferramenta específica.

2.2 Métricas de Desempenho

As métricas de desempenho têm como objetivo avaliar a capacidade do sistema de responder adequadamente sob diferentes níveis de carga e garantir uma experiência fluida para o usuário, mesmo em momentos de maior uso.

Foram definidos os seguintes critérios:

- **Tempo de Resposta:** cada requisição deve ser respondida em menos de 10 segundos.
- **Número de Reservas:** o sistema deve ser capaz de lidar com pelo menos 30 reservas simultâneas sem perda perceptível de desempenho.

Como o backend já está funcional e testável via Postman, essas métricas foram avaliadas por meio de simulações manuais com diferentes usuarios. A verificação do tempo de resposta por requisição também foi feita diretamente no Postman, permitindo validar o comportamento do sistema em condições próximas do uso real, mesmo com a interface ainda em fase de integração total.

Os testes foram realizados utilizando o Postman, com observações do tempo de resposta em diferentes endpoints. Os tempos médios obtidos ficaram todos abaixo de um segundo, demonstrando um bom desempenho.

A simulação de carga foi feita com requisições repetidas para testar o endpoint de criação de reservas, e não houve registros de perda de desempenho perceptível.

2.3 Métricas de Usabilidade

As métricas de usabilidade avaliam a facilidade de uso do sistema, a clareza da interface e a eficiência com que os usuários conseguem realizar as ações proporcionadas pelo sistema. O objetivo é garantir que o sistema seja intuitivo, reduzindo erros e proporcionando uma melhor experiência aos usuarios.

Foram definidos os seguintes critérios:

- **Taxa de Sucesso nas Tarefas:** pelo menos 90% das tarefas devem ser concluídas pelos usuarios sem grandes dificuldades.
- **Tempo Médio para Completar uma Tarefa:** cada tarefa deve ser realizada em menos de três minutos.
- **Número de Erros de Usuário:** deve ser mínimo, indicando uma boa usabilidade e compreensão da interface.

Como a interface já está disponível, ainda que sujeita a ajustes, foi possível aplicar esses testes de forma pratica com usuarios para simular interações simples, como realizar uma reserva, editar cadastro ou cancelar uma reserva, entre outras. Com base nessas simulações, foi possível observar o tempo que cada usuário levou para realizar cada ação, os erros cometidos por eles e o sucesso nas ações realizadas, assim tendo sido possível identificar pontos de melhoria e pontos fortes.

Durante os testes, os usuários concluíram as tarefas com sucesso, e o tempo médio por tarefa permaneceu dentro do limite anteriormente previsto. Foi observado um pequeno número de erros causados por uma leve confusão nos ícones, com uma ênfase no de cadastro de reservas, sugerindo a necessidade de alguns pequenos ajustes visuais desses elementos.

2.4 Métricas de Segurança

As métricas de segurança têm como foco garantir que os dados dos usuários e o funcionamento do sistema estejam protegidos contra acessos indevidos e falhas críticas. Elas também servem como base para ações corretivas e preventivas em caso de vulnerabilidades.

Foram definidos os seguintes critérios:

- **Número de Vulnerabilidades Encontradas:** o sistema deve estar sem falhas de segurança conhecidas na sua entrega final.
- **Tempo Médio para Correção de Falhas:** bugs críticos devem ser corrigidos em até 48 horas após a sua identificação.
- **Execução de Auditorias de Segurança:** devem ser realizadas mensalmente para garantir a manutenção contínua da segurança.

Considerando que o backend já possui autenticação com JWT e segue algumas práticas básicas de proteção, foi possível realizar algumas análises de segurança, como testes manuais de login, verificação de exposição de dados e tentativa de ações não autorizadas.

Foram realizados testes manuais básicos de segurança na API, incluindo tentativas de manipulação de tokens e acessos não autorizados a rotas protegidas. O sistema respondeu adequadamente a essas tentativas, bloqueando ações indevidas, indicando uma boa base de segurança básica.

3. Análise dos Resultados

Os testes realizados com o auxílio do Postman permitiram avaliar a qualidade do sistema de reservas em diferentes aspectos: funcionalidade, segurança, desempenho e usabilidade. Foram realizados testes manuais simulando ações de usuários, além de ataques simples para validar a qualidade da API. A seguir, são apresentados os principais resultados obtidos.

3.1 Funcionalidade e Regras de Negócio

As funcionalidades básicas do sistema (cadastro de usuários, login com autenticação JWT, criação e gestão de reservas) foram validadas por meio de operações CRUD. Os testes confirmaram que:

- O cadastro e edição de usuários funcionam corretamente com dados válidos.
- O sistema aceita dados inválidos (como CPF ou e-mail mal formatados), o que representa uma falha nas validações de entrada. Esse problema foi identificado, porém ainda não foi corrigido.
- A funcionalidade de login com geração de token JWT está operando corretamente com usuários válidos, e retorna mensagens apropriadas para eventuais erros de autenticação.
- As rotas protegidas estão corretamente restritas, e exigem um token válido para liberar o acesso.
- As reservas podem ser criadas, editadas, canceladas e listadas com sucesso.
- Algumas regras de negócio importantes não estão sendo aplicadas corretamente:
 - Criação de reservas fora do expediente estão sendo permitidas.
 - Reservas duplicadas para o mesmo cliente, na mesma data e mesmo horário estão sendo aceitas.

Assim, podemos concluir que o sistema implementa corretamente o CRUD e o fluxo geral, mas ainda faltam validações de dados e regras de negócio mais eficientes.

3.2 Segurança

Foram realizados alguns testes focados na segurança da API, com os seguintes resultados:

- A autenticação via JWT funciona corretamente, e o sistema bloqueia o acesso não autorizado às rotas protegidas.
- Tokens inválidos são recusados corretamente.
- Ao tentar realizar injeções de SQL nos campos de login (ex.: ' OR '1'='1'), o sistema retorna mensagens de erro como “Senha inválida” e “Email não encontrado”, indicando que não houve brechas de segurança do sistema nesse quesito.
- Tentativas de acesso a rotas de administrador sem autenticação resultam em erro.
- Ao tentar cadastrar um CPF já existente, o sistema responde com um erro 500, indicando que há controle, porém não retorna uma mensagem de erro mais específica e explicativa para o usuário.

Pode-se concluir, então, que o sistema apresenta boas práticas de segurança quanto à autenticação e injeção, mas precisa melhorar na questão de validação de dados de entrada e tratamento de erros.

3.3 Desempenho e Teste de Carga

Foi feito um teste de carga leve, simulando a criação de 30 reservas em sequência, e foram levantados os seguintes pontos:

- O tempo de resposta de todas as requisições ficou abaixo de 800ms, mesmo com diversas operações sendo feitas uma após a outra.
- A API continuou funcionando sem travamentos ou perda de performance significativa.
- As operações de edição e exclusão funcionaram corretamente mesmo com uma base de dados maior.
- A listagem GET /reserva retornou todos os registros corretamente e sem lentidão perceptível.

Percebe-se que o sistema é capaz de lidar com uma quantidade significativa de dados sem perda perceptível de desempenho, cumprindo com os requisitos esperados para o sistema.

3.4 Usabilidade e Feedback ao Usuário

Durante os testes, também foi observado como o sistema se comunica com o usuário:

- As mensagens de sucesso e erro são claras e ajudam o usuário a entender o que está acontecendo.
- Em alguns casos, como ao tentar cadastrar um CPF que já está cadastrado com outros dados no sistema, a mensagem retornada ainda é genérica.
- O sistema retorna mensagens específicas em casos como login com email inexistente ou senha errada.

Além desses testes técnicos, também foram realizados testes de usabilidade com usuários reais, onde foi possível perceber que:

- O sistema é fácil de usar e a API retorna informações compreensíveis.
- O tempo de execução das tarefas é consideravelmente bom.
- A maior dificuldade identificada foi a identificação de ícones, o que pode ser melhorado com algumas melhorias simples no frontend.

Conclui-se que o sistema apresenta uma boa usabilidade geral, mas precisa melhorar na comunicação em erros inesperados, além de reforçar validações no frontend e backend.

3.5 Complexidade Ciclomática

Foi realizada uma análise preliminar da complexidade ciclomática do código-fonte com o auxílio da extensão CodeMetrics no Visual Studio Code. Essa métrica indica a quantidade de caminhos independentes existentes dentro de funções e métodos, servindo como referência para avaliar a legibilidade, manutenibilidade e testabilidade do código.

Durante a auditoria, todas as funções analisadas apresentaram valores inferiores ao limite de quinze, o maior número identificado tendo sido de onze. Isto sugere que o código esteja bem estruturado, com baixo acoplamento lógico e fluxo de controle adequado.

Cabe ressaltar que essa análise foi realizada com uma ferramenta leve e local, devido a algumas limitações encontradas, porém, pretende-se ser realizados testes complementares com o SonarQube ou outra ferramenta similar, a fim de obter uma análise mais completa da qualidade do código.

Ainda assim, os resultados atuais oferecem uma indicação inicial de que a complexidade do sistema está sob controle.

3.6 Linhas de Código por Método

Esta métrica foi avaliada com o objetivo de compreender a distribuição e o tamanho dos blocos funcionais do sistema. Essa análise ajuda a identificar métodos potencialmente complexos ou sobrecarregados, e que podem comprometer a legibilidade e manutenibilidade do código.

Como não foi possível utilizar uma ferramenta automatizada para essa métrica devido alguns problemas enfrentados, a coleta dos dados foi realizada manualmente por meio de leitura e análise direta dos arquivos-fonte no Visual Studio Code. Apesar de mais trabalhosa, essa abordagem permitiu obter uma perspectiva geral da estrutura dos métodos.

De modo geral, os métodos avaliados apresentam uma quantidade de linhas compatível com boas práticas de desenvolvimento, sendo a maioria composta por blocos pequenos e objetivos. O maior método identificado possui vinte e quatro linhas totais, apenas três delas sendo linhas vazias ou preenchidas por comentários.

3.7 Síntese dos Resultados

Critério	Resultado
Funcionalidade	CRUD funcionando corretamente. Regras de negócio ainda estão incompletas.
Segurança	Autenticação bem desenvolvida. Proteção contra injeção. Pontos de melhoria no tratamento de erros e validação de dados.
Desempenho	Boa performance com mais de 30 reservas. Respostas rápidas e sem travamentos.
Usabilidade	Interface clara. Mensagens compreensíveis, mas com erros genéricos.
Complexidade Ciclomática	Todas as funções com valores abaixo de quinze. Código bem estruturado. Pretende-se utilizar SonarQube para análise mais aprofundada.
Linhas de Código por Método	Nenhum método ultrapassou 24 linhas totais. Estrutura geral concisa, favorecendo legibilidade e manutenibilidade.

4. Ações Corretivas e Melhorias Propostas

Com base nas análises realizadas nos testes manuais e com usuários reais, foram identificadas oportunidades de melhoria que visam aumentar a qualidade, usabilidade e desempenho do sistema. A seguir, estão algumas das principais ações corretivas e melhorias propostas:

4.1 Melhorias na Usabilidade

- Aprimorar a identificação de ícones.
- Adicionar mensagens de erro mais específicos e explicativos.

4.2 Correções em Mensagens de Erro

- Substituir as mensagens genéricas por mensagens mais específicas, como no caso de tentativa de cadastro com CPF já existente, para ajudar o usuário a entender exatamente o motivo do erro ter acontecido e como ele pode ser corrigido.
- Unificar o padrão visual das mensagens, mantendo clareza, design consistente e linguagem simples.

4.3 Validações

- Implementar validações em tempo real nos formulários (ex.: CPF inválido, e-mail mal formatado), tanto no frontend quanto no backend.
- Exibir mensagens instantâneas ao usuário ao detectar campos incorretos, evitando envio desnecessário do formulário.

4.4 Código e Arquitetura

- Aplicar uma ferramenta automatizada como o SonarQube para realizar testes mais aprofundados como a complexidade ciclomática.
- Manter possíveis métodos ou funções futuras abaixo de 50 linhas.

4.5 Funcionalidade e Regras de Negócio

- Finalizar as regras de negócio que ainda se encontram pendentes, como validar cadastro de reservas apenas no horário de funcionamento do restaurante, bloqueio de cadastro de reservas duplicadas, entre outros.

5. Conclusão

A auditoria de qualidade do sistema de reservas de mesa de um restaurante aqui apresentada permitiu uma ampla análise de aspectos cruciais para o bom funcionamento do sistema, como desempenho, segurança, usabilidade e organização do código. A partir de uma combinação de testes como simulações manuais e interações com usuários, foi possível identificar tanto os pontos fortes quanto as áreas em que ainda são necessários alguns aprimoramentos e mudanças.

Os resultados obtidos foram, em sua maioria, positivos. Foram identificados boa performance mesmo havendo uma carga moderada, além de tempos de resposta adequados, segurança básica bem implementada por meio de autenticação JWT e proteção contra injeções, tudo isso aliado a uma interface funcional e de maneira geral clara para os usuários. A estrutura do código também se mostrou coesa e de fácil manutenção, com métodos curtos e bem-organizados.

Entretanto, algumas fragilidades foram detectadas, como algumas regras de negócio incompletas, mensagens de erro genéricas, leves confusões com ícones da interface e falta de validações mais completas em algumas rotas. Com esses pontos tendo sido identificados e documentados, foi possível a estruturação de um conjunto de ações corretivas e melhorias propostas, que visam aperfeiçoar a qualidade geral do sistema.

Conclui-se, portanto, que o sistema está em uma etapa estável o suficiente para um funcionamento básico, porém ainda requer alguns ajustes tanto em alguns pontos técnicos quanto na experiência do usuário. A adoção das melhorias propostas será fundamental para garantir, nos próximos passos do desenvolvimento, principalmente uma segurança firme e uma usabilidade de forma consistente.

6. Evidências da Auditoria de Qualidade

Abaixo estão as evidências detalhadas dos testes realizados para validação das métricas descritas na seção 2.

6.1 M-01 – Complexidade Ciclométrica

Ferramenta: CodeMetrics (extensão para Visual Studio Code).

Cenário: Foi realizada uma análise manual da complexidade ciclométrica de funções e métodos relevantes do sistema, com o apoio da extensão. A verificação foi feita diretamente no Visual Studio Code, observando os valores atribuídos automaticamente pela ferramenta.

Resultado esperado: Todas as funções devem manter a complexidade ciclométrica inferior a quinze.

Resultado obtido: Todas as funções analisadas respeitaram o limite estabelecido, o maior valor identificado tendo sido de onze.

Evidências:

controllers > JS authController.js > ...

```
1  const jwt = require('jsonwebtoken'); 53.9k (gzipped: 16.1k)
2  const bcrypt = require('bcrypt'); 7.3k (gzipped: 2.5k)
3  const pool = require('../config/db');
4  const config = require('../config/config');
5
Complexity is 7 It's time to do something...
6  exports.login = async (req, res) => {
7    const { email, senha } = req.body;
8
9    try {
10     const [rows] = await pool.query('SELECT * FROM Usuario WHERE email_Usuario = ?', [email]);
11
12     if (rows.length === 0) {
13       return res.status(401).json({ message: 'Email não encontrado' });
14     }
15
16     const usuario = rows[0];
17     const senhaValida = await bcrypt.compare(senha, usuario.senha_Usuario);
18
19     if (!senhaValida) {
20       return res.status(401).json({ message: 'Senha inválida' });
21     }
22
23     const token = jwt.sign({ id: usuario.id }, config.jwtSecret, { expiresIn: '1h' });
24
25     res.json({ message: 'Login realizado com sucesso', token });
26   } catch (error) {
27     console.error('Erro no login:', error);
28     res.status(500).json({ message: 'Erro no servidor' });
29   }
30 };
```

```
JS reservaRoutes.js JS clienteController.js X
controllers > JS clienteController.js > [?] clienteController
1  const Usuario = require('../models/usuarioModel');
2  const Cliente = require('../models/clienteModel');
3
4  const clienteController = {
    Complexity is 9 It's time to do something...
5    criar: async (req, res, next) => {
6      try {
7        const { nome, email, telefone, cpf, senha } = req.body;
8
9        // Verificar se todos os dados obrigatórios foram preenchidos
10       if (!nome || !email || !telefone || !cpf || !senha) {
11         return res.status(400).json({ message: 'Todos os campos são obrigatórios' });
12       }
13
14       // Criar o usuário (não precisamos de token aqui)
15       const idUsuario = await Usuario.criar({ nome, email, telefone, cpf, senha });
16
17       // Criar o cliente com base no id do usuário
18       const idCliente = await Cliente.criar(idUsuario);
19
20       // Retornar a resposta de sucesso
21       res.status(201).json({ idCliente, message: 'Cliente criado com sucesso' });
22     } catch (error) {
23       next(error);
24     }
25   },
26
27   Complexity is 3 Everything is cool!
28   listarTodos: async (req, res, next) => {
29     try {
30       const clientes = await Cliente.listarTodos();
31       res.json(clientes);
32     } catch (error) {
33       next(error);
34     }
35   },
36
37   Complexity is 3 Everything is cool!
38   excluir: async (req, res, next) => {
39     try {
40       const { id } = req.params;
41       await Cliente.excluir(id);
42       res.json({ message: 'Cliente excluído com sucesso' });
43     } catch (error) {
44       next(error);
45     }
46   },
47   module.exports = clienteController;
```

- Maior valor identificado (11):

```
JS reservaRoutes.js JS usuarioController.js X
controllers > JS usuarioController.js > ...
1  const db = require('../config/db');
2  const bcrypt = require('bcrypt'); 7.3k (gzipped: 2.5k)
3
4  // Criar novo usuário
5  exports.criarUsuario = async (req, res) => {
6    const { nome_usuario, cpf_usuario, email_usuario, telefone_usuario, senha_usuario } = req.body;
7    if (!nome_usuario || !cpf_usuario || !email_usuario || !senha_usuario) {
8      return res.status(400).json({ message: 'Todos os campos são obrigatórios!' });
9    }
10
11    try {
12      // Verificar se já existe um usuário com o mesmo e-mail
13      const [existingUser] = await db.execute('SELECT * FROM Usuario WHERE email_usuario = ?', [email_usuario]);
14      if (existingUser.length > 0) {
15        return res.status(400).json({ message: 'E-mail já está em uso' });
16      }
17
18      const hashedPassword = await bcrypt.hash(senha_usuario, 10);
19      const [result] = await db.execute(
20        `INSERT INTO Usuario (nome_usuario, cpf_usuario, email_usuario, telefone_usuario, senha_usuario)
21        VALUES (?, ?, ?, ?, ?)`,
22        [nome_usuario, cpf_usuario, email_usuario, telefone_usuario || null, hashedPassword]
23      );
24      res.status(201).json({ message: 'Usuário cadastrado com sucesso', userId: result.insertId });
25    } catch (err) {
26      console.error('Erro ao criar usuário:', err);
27      res.status(500).json({ error: 'Erro ao criar usuário', details: err.message });
28    }
29  };
```

- Menores valores identificados (3):

```
JS reservaRoutes.js • JS reservaController.js X
controllers > JS reservaController.js > [⌕] reservaController > atualizar
1  const Reserva = require('../models/reservaModel');
2
3  const reservaController = {
4      Complexity is 3 Everything is cool!
5      criar: async (req, res, next) => {
6          try {
7              const { numeropessoas, data, clienteId } = req.body;
8              const idReserva = await Reserva.criar({ numeropessoas, data, clienteId });
9              res.status(201).json({ idReserva, message: 'Reserva criada com sucesso' });
10             } catch (error) {
11                 next(error);
12             }
13         },
14
15         Complexity is 3 Everything is cool!
16         listarTodas: async (req, res, next) => {
17             try {
18                 const reservas = await Reserva.listarTodas();
19                 res.json(reservas);
20             } catch (error) {
21                 next(error);
22             }
23         },
24
25         Complexity is 3 Everything is cool!
26         atualizar: async (req, res, next) => {
27             try {
28                 const { id } = req.params;
29                 const { numeropessoas, data, status } = req.body;
30                 await Reserva.atualizar(id, { numeropessoas, data, status });
31                 res.json({ message: 'Reserva atualizada com sucesso' });
32             } catch (error) {
33                 next(error);
34             }
35         },
36
37         Complexity is 3 Everything is cool!
38         excluir: async (req, res, next) => {
39             try {
40                 const { id } = req.params;
41                 await Reserva.excluir(id);
42                 res.json({ message: 'Reserva excluída com sucesso' });
43             } catch (error) {
44                 next(error);
45             }
46         }
47     };
48
49     module.exports = reservaController;
```

Observações:

- Nenhuma função atingiu ou ultrapassou o limite de quinze.
- O maior valor identificado foi de onze.

- O menor valor identificado foi de três.
- Eventualmente, a análise será reexecutada com uma ferramenta mais abrangente, provavelmente o SonarQube, para aprofundar a verificação de qualidade.

6.2 M-03 – Linhas de Código por Método

Ferramenta: Análise manual pelo Visual Studio Code.

Cenário: Cada função ou método foi avaliado individualmente, com contagem manual do número de linhas totais, incluindo e excluindo linhas vazias e comentários.

Resultado esperado: Métodos devem ter no máximo 50 linhas para manter legibilidade.

Resultado obtido: Todos os métodos analisados se mantiveram abaixo desse limite, a maior função identificada possuindo 24 linhas totais.

Evidências:

Arquivo	Função/Método	Linhas de Código	Linhas Vazias/de comentário	Diferença
authController.js	exports.login	24	6	18
clienteController.js	criar	20	8	12
clienteController.js	listarTodos	7	0	7
clienteController.js	excluir	8	0	8
reservaController.js	criar	8	0	8
reservaController.js	listarTodas	7	0	7
reservaController.js	atualizar	9	0	9
reservaController.js	excluir	8	0	8
usuarioController.js	exports.criarUsuario	24	3	21
usuarioController.js	exports.listarUsuarios	10	0	10
usuarioController.js	exports.buscarUsuarioPorId	13	0	13
usuarioController.js	exports.atualizarUsuario	23	4	19
usuarioController.js	exports.deletarUsuario	14	2	12
authMiddleware.js	authMiddleware	20	8	12
errorMiddleware.js	module.exports	7	0	7
clienteModel.js	criar	6	0	6
clienteModel.js	listarTodos	7	0	7
clienteModel.js	excluir	2	0	2
reservaModel.js	criar	6	0	6
reservaModel.js	listarTodas	8	0	8
reservaModel.js	atualizar	5	0	5
reservaModel.js	excluir	2	0	2

usuarioModel.js	criar	6	0	6
usuarioModel.js	listarTodos	3	0	3
usuarioModel.js	atualizar	5	0	5
usuarioModel.js	excluir	2	0	2

6.3 M-04 – Tempo de Resposta

Ferramenta: Postman.

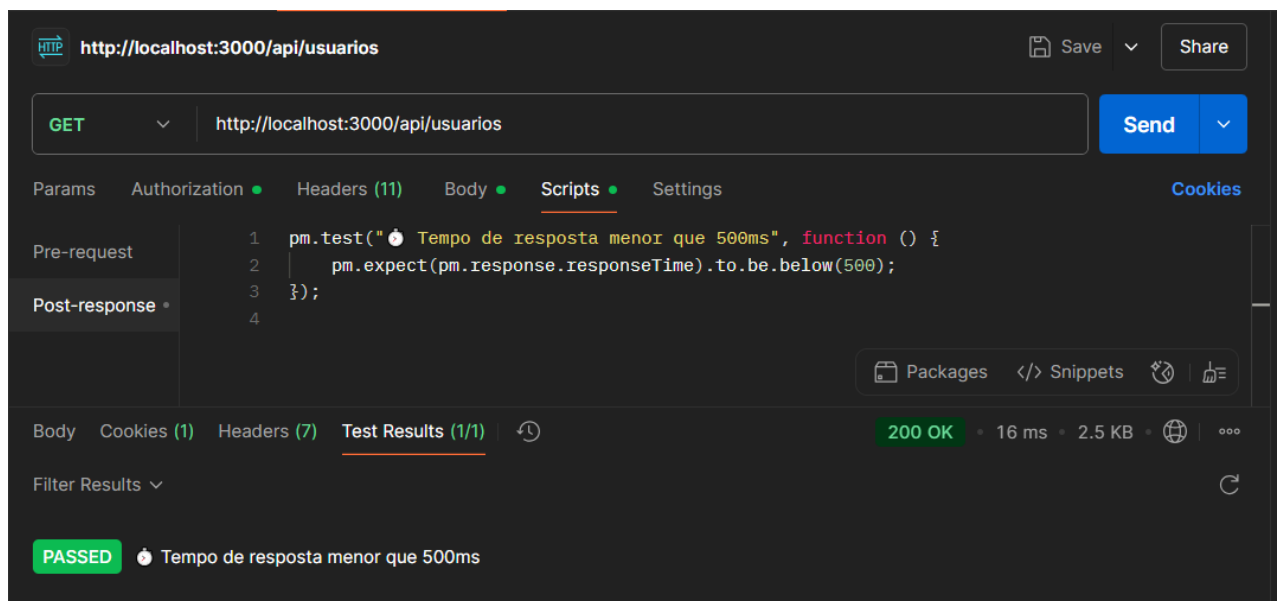
Cenário: Requisição **GET /reservas**.

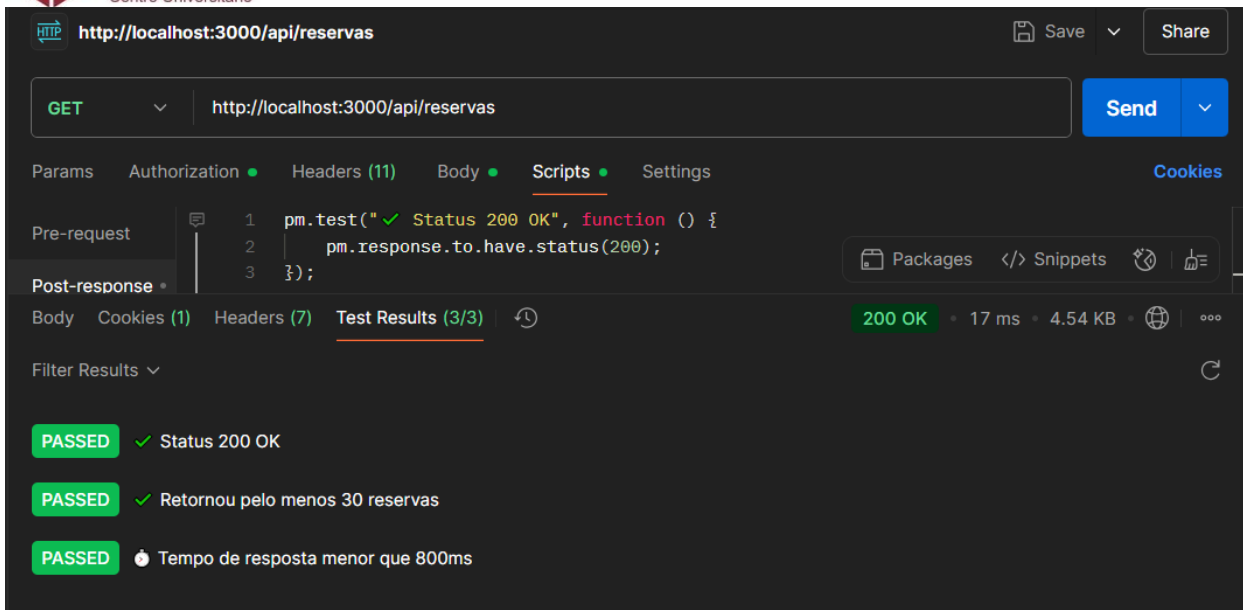
Resultado esperado: Inferior a 10 segundos por requisição.

Resultado obtido: Todos os resultados obtidos foram inferiores a 800ms.

Evidência:

- Todas as requisições tiveram um tempo de resposta de:





HTTP **http://localhost:3000/api/reservas** Save Share

GET **http://localhost:3000/api/reservas** Send

Params Authorization Headers (11) Body Scripts Settings Cookies

Pre-request

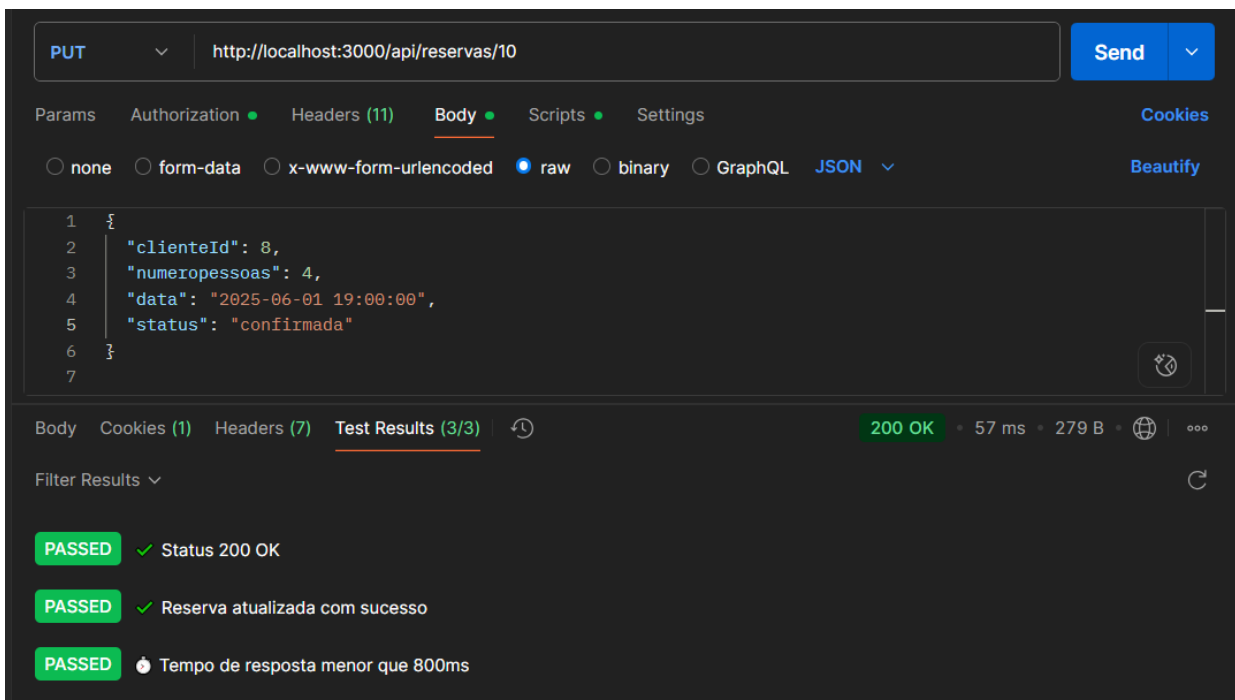
```
1 pm.test("✓ Status 200 OK", function () {  
2   pm.response.to.have.status(200);  
3 });
```

Post-response

Body Cookies (1) Headers (7) Test Results (3/3) 200 OK • 17 ms • 4.54 KB

Filter Results

- PASSED ✓ Status 200 OK
- PASSED ✓ Retornou pelo menos 30 reservas
- PASSED ⌚ Tempo de resposta menor que 800ms



PUT **http://localhost:3000/api/reservas/10** Send

Params Authorization Headers (11) Body Scripts Settings Cookies

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL JSON Beautify

```
1 {  
2   "clienteId": 8,  
3   "numeroPessoas": 4,  
4   "data": "2025-06-01 19:00:00",  
5   "status": "confirmada"  
6 }  
7
```

Body Cookies (1) Headers (7) Test Results (3/3) 200 OK • 57 ms • 279 B

Filter Results

- PASSED ✓ Status 200 OK
- PASSED ✓ Reserva atualizada com sucesso
- PASSED ⌚ Tempo de resposta menor que 800ms

6.4 M-05 – Número de Reservas (Teste de Carga)

Ferramenta: Postman.

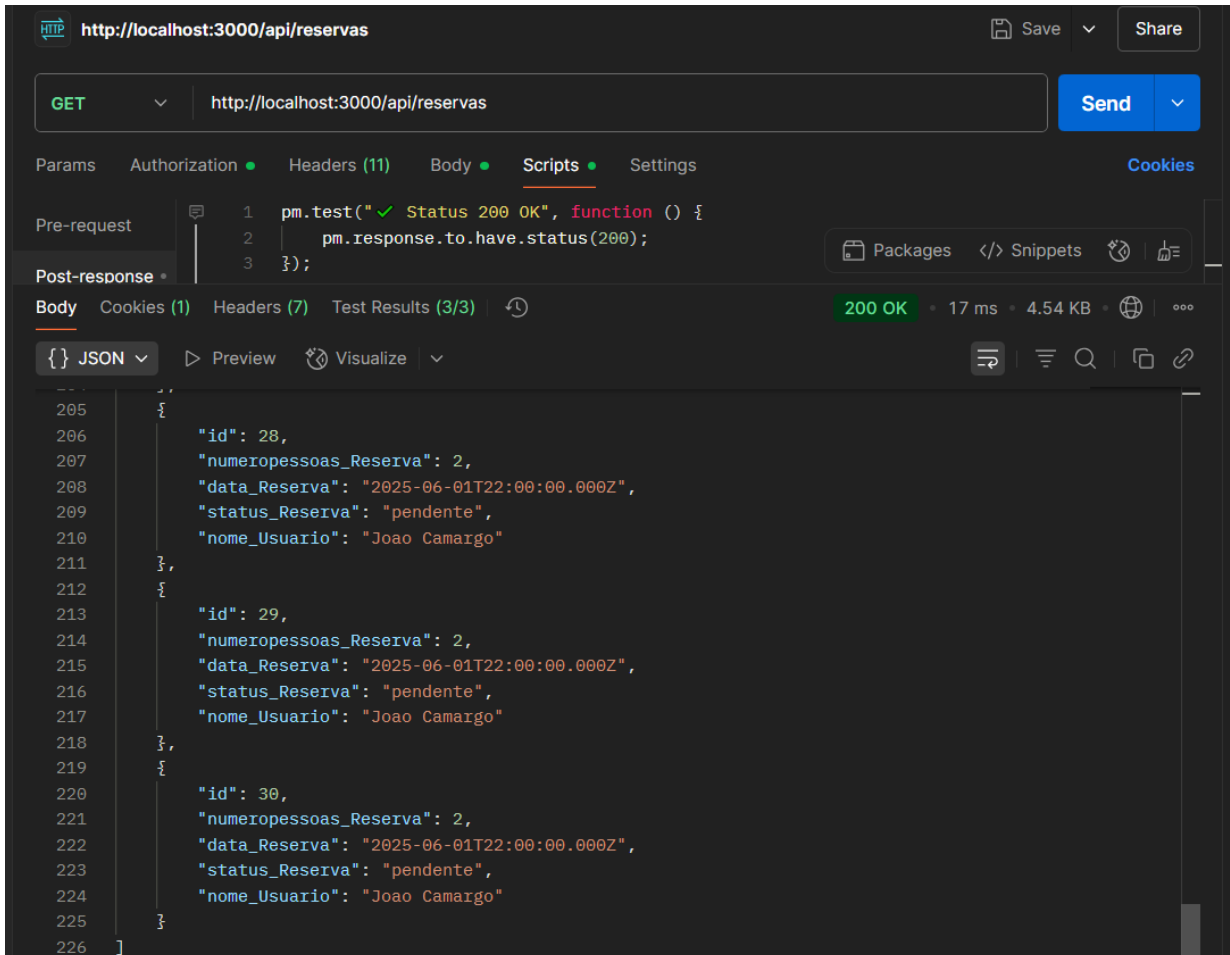
Cenário: Envio de 30 requisições **POST /reservas** com dados válidos em sequência.

Resultado esperado: No mínimo 30 reservas simultâneas.

Resultado: Mais de 30 reservas registradas.

Evidências:

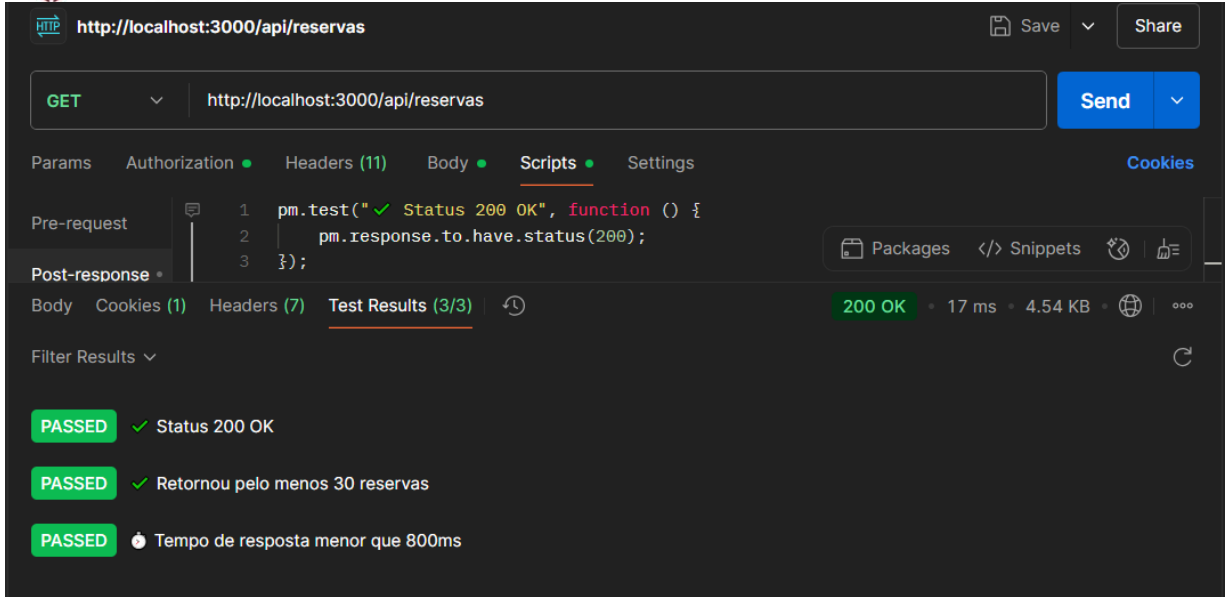
- **GET /reserva** para verificar se retorna todas corretamente.



The screenshot shows a REST client interface with the following details:

- URL:** `http://localhost:3000/api/reservas`
- Method:** `GET`
- Status:** `200 OK` (17 ms, 4.54 KB)
- Response Body (JSON):**

```
[{"id": 28, "numeropessoas_Reserva": 2, "data_Reserva": "2025-06-01T22:00:00.000Z", "status_Reserva": "pendente", "nome_usuario": "Joao Camargo"}, {"id": 29, "numeropessoas_Reserva": 2, "data_Reserva": "2025-06-01T22:00:00.000Z", "status_Reserva": "pendente", "nome_usuario": "Joao Camargo"}, {"id": 30, "numeropessoas_Reserva": 2, "data_Reserva": "2025-06-01T22:00:00.000Z", "status_Reserva": "pendente", "nome_usuario": "Joao Camargo"}]
```

HTTP **http://localhost:3000/api/reservas** Save Share

GET **http://localhost:3000/api/reservas** Send

Params Authorization Headers (11) Body Scripts Settings Cookies

Pre-request

```
1 pm.test("✓ Status 200 OK", function () {  
2   pm.response.to.have.status(200);  
3 });
```

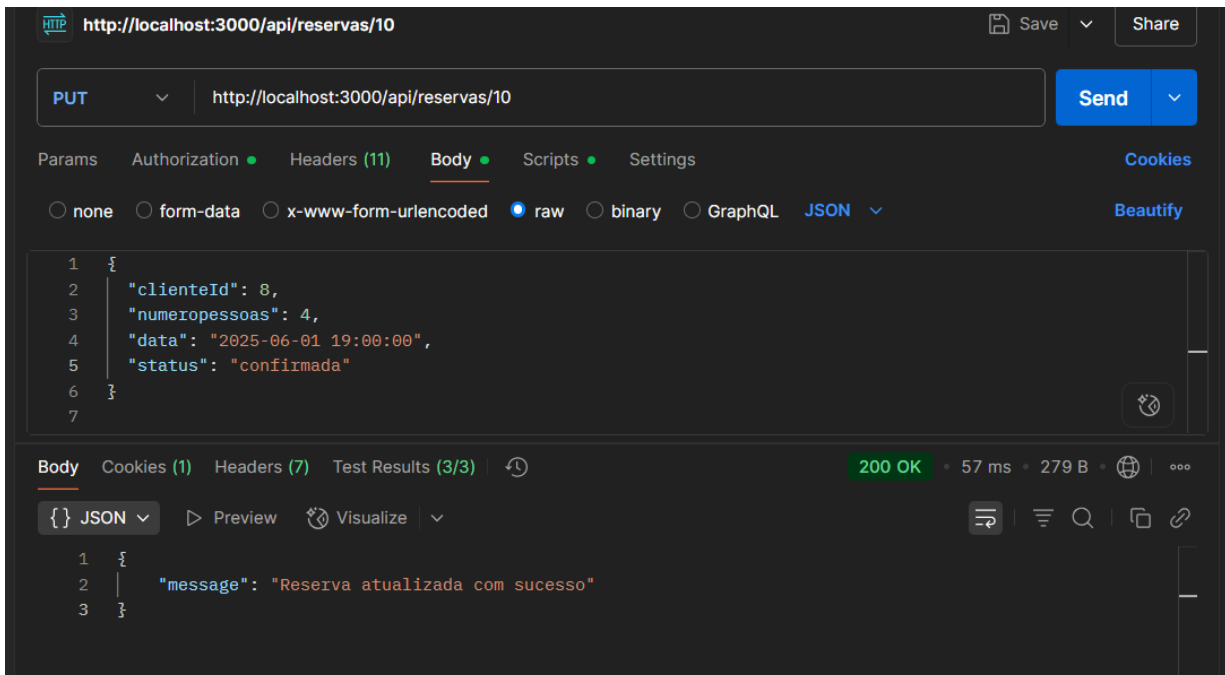
Post-response

Body Cookies (1) Headers (7) Test Results (3/3) 200 OK • 17 ms • 4.54 KB

Filter Results

- PASSED ✓ Status 200 OK
- PASSED ✓ Retornou pelo menos 30 reservas
- PASSED ⚙ Tempo de resposta menor que 800ms

- **PUT /reserva/:id:** editar uma reserva:



HTTP **http://localhost:3000/api/reservas/10** Save Share

PUT **http://localhost:3000/api/reservas/10** Send

Params Authorization Headers (11) Body Scripts Settings Cookies

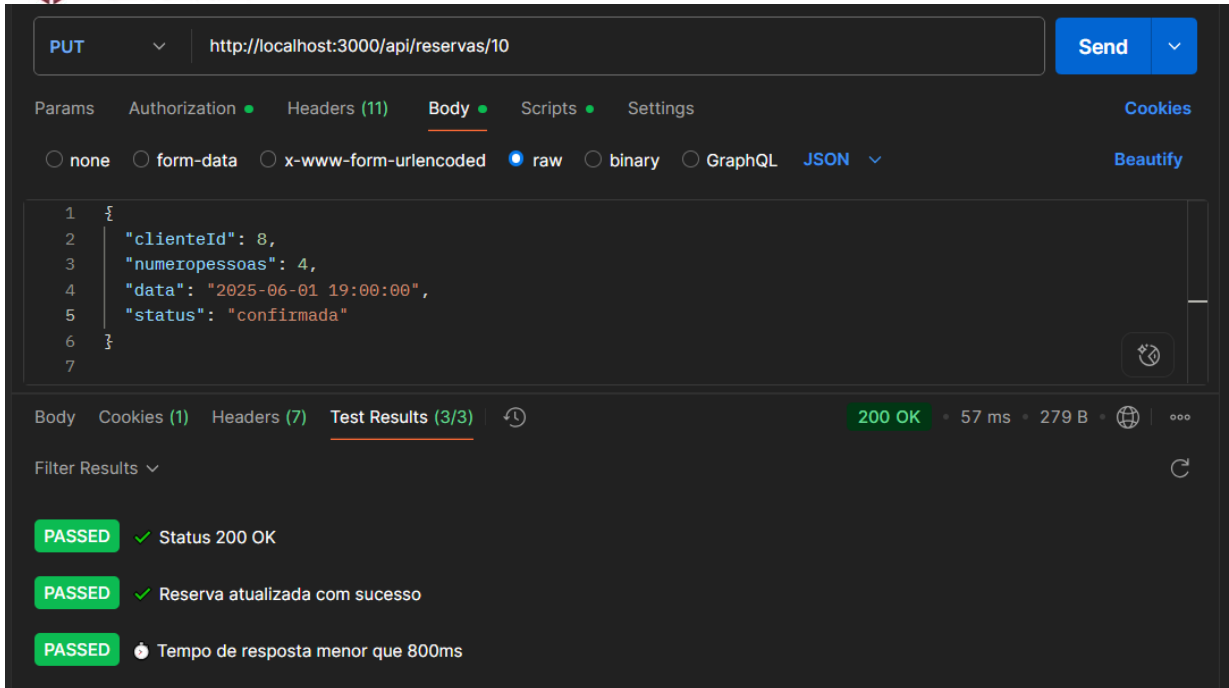
☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL JSON Beautify

```
1 {  
2   "clienteId": 8,  
3   "numeroPessoas": 4,  
4   "data": "2025-06-01 19:00:00",  
5   "status": "confirmada"  
6 }  
7
```

Body Cookies (1) Headers (7) Test Results (3/3) 200 OK • 57 ms • 279 B

{ } JSON Preview Visualize

```
1 {  
2   "message": "Reserva atualizada com sucesso"  
3 }
```



PUT ▼ http://localhost:3000/api/reservas/10 Send ▼

Params Authorization ● Headers (11) **Body** ● Scripts ● Settings Cookies

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL **JSON** ▼ Beautify

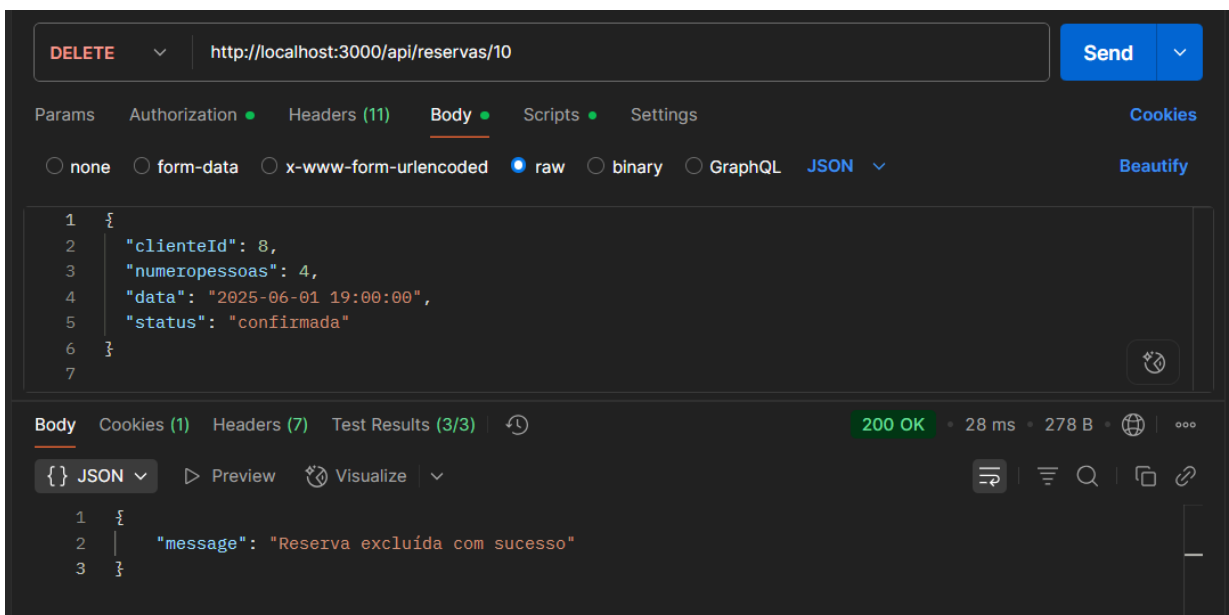
```
1 {
2   "clienteId": 8,
3   "numeroPessoas": 4,
4   "data": "2025-06-01 19:00:00",
5   "status": "confirmada"
6 }
7
```

Body Cookies (1) Headers (7) **Test Results (3/3)** 🕒 **200 OK** • 57 ms • 279 B • 🌐 ⋮

Filter Results ▼ 🔄

- PASSED** ✓ Status 200 OK
- PASSED** ✓ Reserva atualizada com sucesso
- PASSED** ⚙️ Tempo de resposta menor que 800ms

- **DELETE /reserva:id:** remover uma reserva:



DELETE ▼ http://localhost:3000/api/reservas/10 Send ▼

Params Authorization ● Headers (11) **Body** ● Scripts ● Settings Cookies

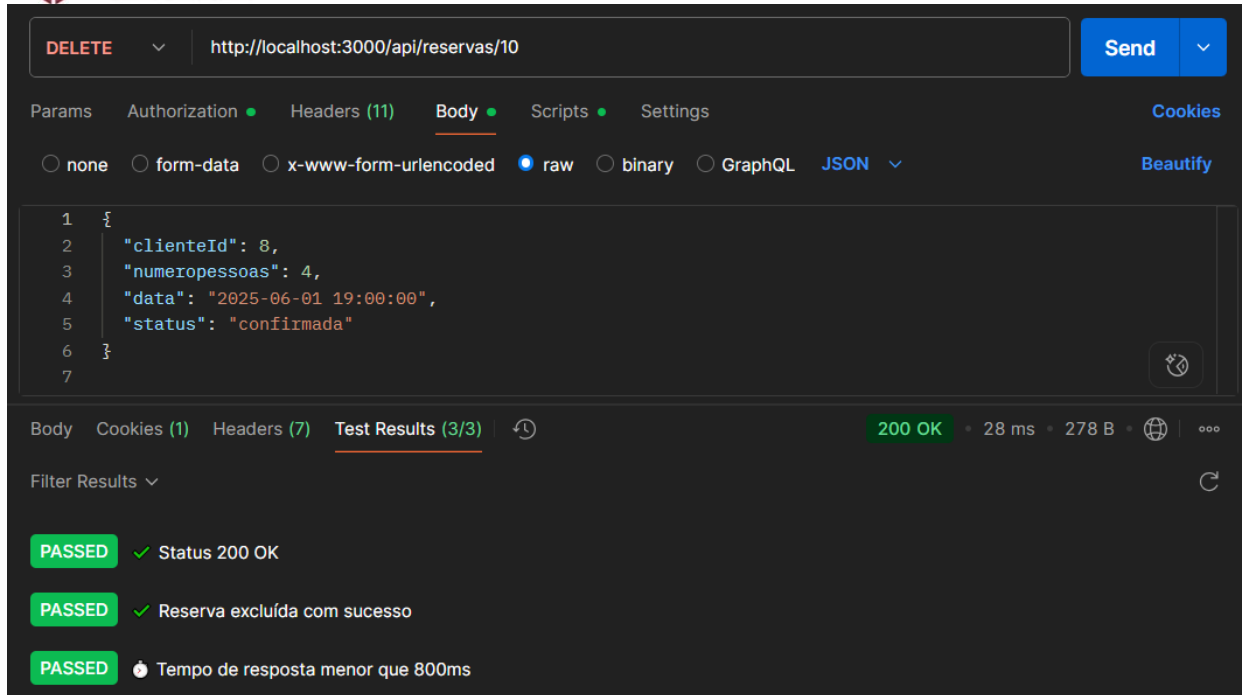
☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL **JSON** ▼ Beautify

```
1 {
2   "clienteId": 8,
3   "numeroPessoas": 4,
4   "data": "2025-06-01 19:00:00",
5   "status": "confirmada"
6 }
7
```

Body Cookies (1) Headers (7) **Test Results (3/3)** 🕒 **200 OK** • 28 ms • 278 B • 🌐 ⋮

{} **JSON** ▼ ▶ Preview 🔄 Visualize ▼ 🔍 📄 🔗

```
1 {
2   "message": "Reserva excluída com sucesso"
3 }
```



DELETE ▼ http://localhost:3000/api/reservas/10 Send ▼

Params Authorization Headers (11) **Body** Scripts Settings Cookies

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL **JSON** ▼ Beautify

```
1 {
2   "clienteId": 8,
3   "numeroPessoas": 4,
4   "data": "2025-06-01 19:00:00",
5   "status": "confirmada"
6 }
7
```

Body Cookies (1) Headers (7) **Test Results (3/3)** ↺ 200 OK • 28 ms • 278 B • 🌐 ⋮

Filter Results ▼ ↺

- PASSED** ✓ Status 200 OK
- PASSED** ✓ Reserva excluída com sucesso
- PASSED** ⚡ Tempo de resposta menor que 800ms

Observações:

- O tempo de resposta do **GET** não aumentou de maneira considerável.
- O sistema continua funcionando sem travamentos.
- AAPI responde corretamente às ações de edição e cancelamento mesmo com muitos dados.
- O sistema suportou ao menos 30 reservas cadastradas sem perda perceptível de desempenho.

6.5 M-06, M-07 e M-08 – Usabilidade com Usuário

Ferramenta: Observação, cronometro e tabela no word para anotações.

Cenário: Simulações de execução das tarefas no sistema.

Resultados esperados:

- **Taxa de Sucesso nas Tarefas:** Maior ou igual a 90%.
- **Tempo Médio para Completar Tarefa:** Inferior a 3 minutos.
- **Número de Erros de Usuário:** Mínimo possível.

Resultados Obtidos:

- Cinco usuarios testaram.
- 100% completaram as tarefas.
- Tempo médio: ~40 segundos por tarefa.

- Confusões com relação aos icons, principalmente os das reservas.

Evidências:

Usuário	Tarefa	Sucesso?	Tempo	Erros
Usuário 1	Cadastro	OK	1:06min	Dificuldade em identificar icons
	Login	OK	0:15min	
	Fazer reserva	OK	3:18min	
	Cancelar reserva	OK	1:04min	
	Sair	OK	0:40min	
Usuário 2	Cadastro	OK	1:20min	Dificuldade em identificar icons
	Login	OK	0:14min	
	Fazer reserva	OK	0:49min	
	Cancelar reserva	OK	0:10min	
	Ver perfil	OK	0:05min	
	Editar perfil	OK	0:18min	
	Sair	OK	0:05min	
Usuário 3	Cadastro	OK	1:30min	Tentou enviar cadastro sem preencher todos os campos
	Login	OK	0:20min	
	Fazer reserva	OK	1:02min	
	Cancelar reserva	OK	0:36min	
	Editar perfil	OK	0:25min	
	Sair	OK	0:08min	
Usuário 4	Cadastro	OK	1:10min	Dificuldade em identificar icons
	Login	OK	0:12min	
	Fazer reserva	OK	0:56min	
	Cancelar reserva	OK	0:30min	
	Ver perfil	OK	0:06min	
	Sair	OK	0:06min	
Usuário 5	Cadastro	OK	0:59min	
	Login	OK	0:11min	
	Fazer reserva	OK	1:15min	
	Cancelar reserva	OK	0:33min	
	Sair	OK	0:04min	

6.6 M-09 – Número de Vulnerabilidades Encontradas

Testes aplicados:

- Login sem token.
- SQL Injection.
- Token inválido.

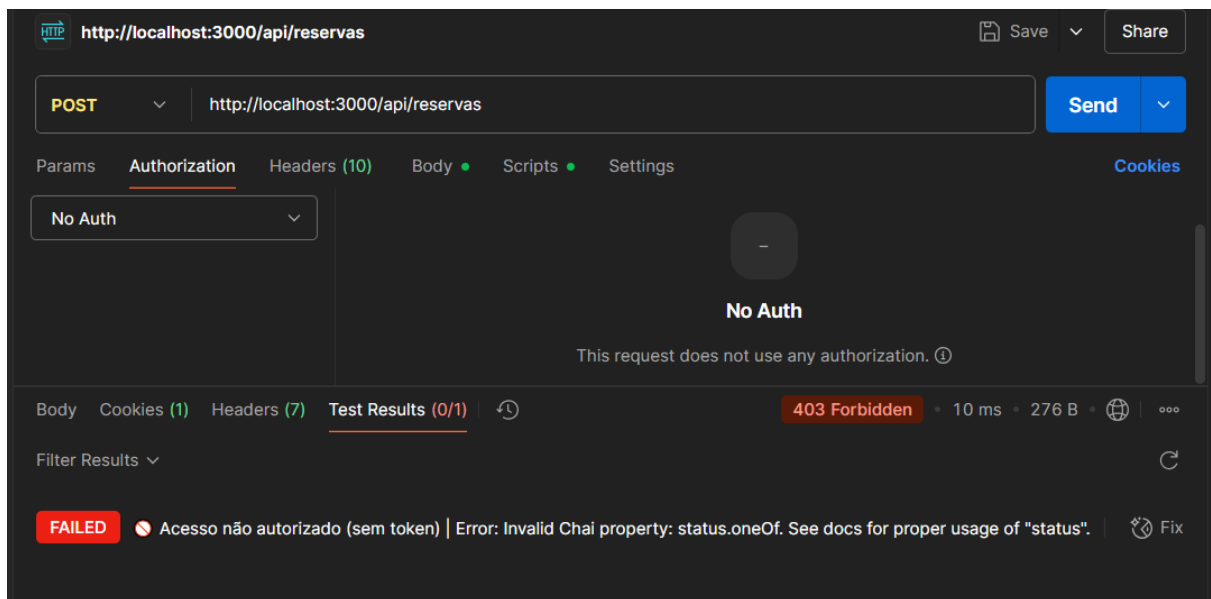
Ferramentas: Postman.

Resultado esperado: Zero na entrega final.

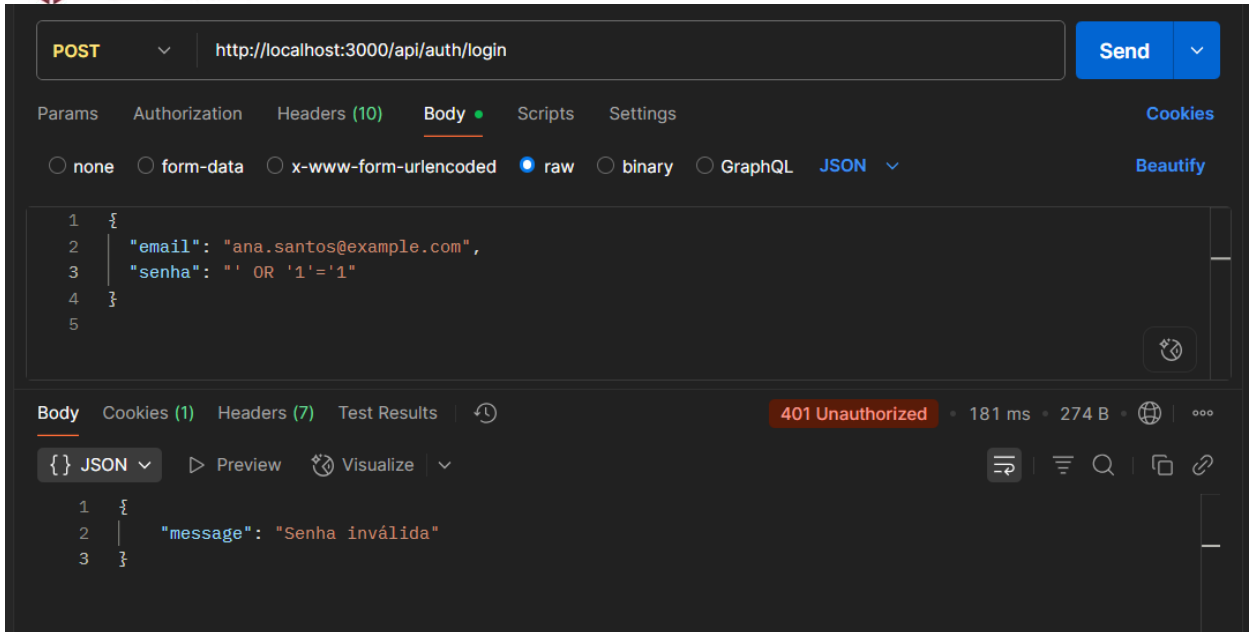
Resultado obtido: Nenhuma vulnerabilidade crítica detectada.

Evidências:

- Tentar acessar rotas de administrador sem passar o token de autenticação:



- Testando injeção de SQL:
 - Enviando ' OR '1'='1 no campo de senha:



POST http://localhost:3000/api/auth/login

Params Authorization Headers (10) Body Scripts Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

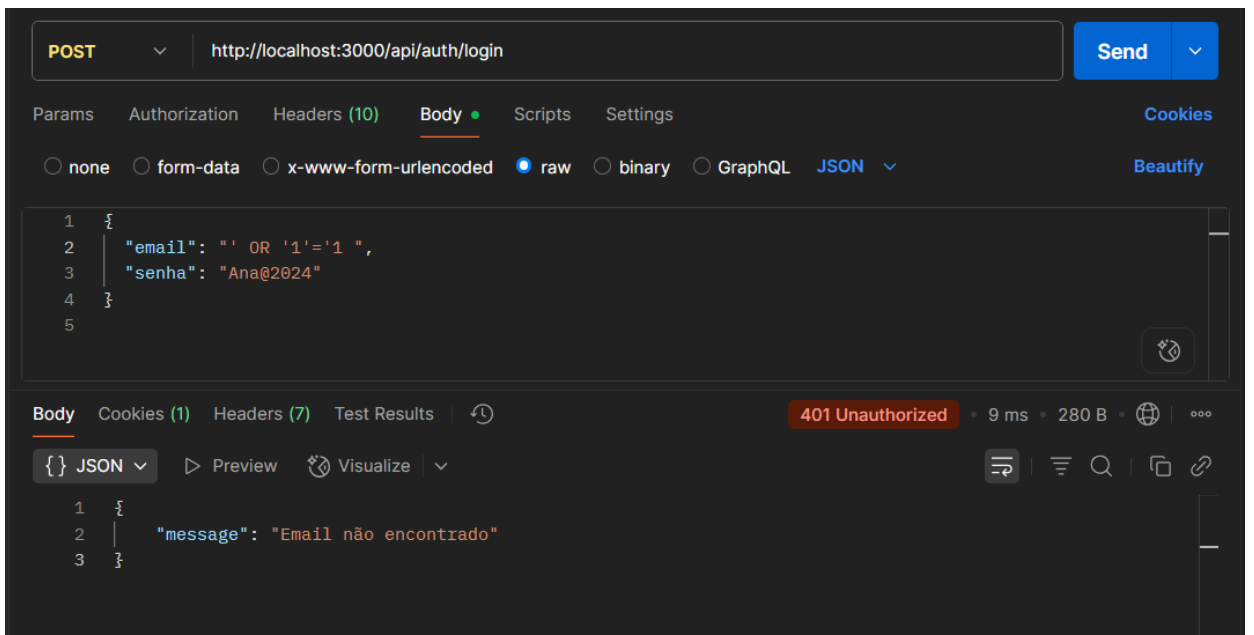
```
1 {
2   "email": "ana.santos@example.com",
3   "senha": " OR '1'='1"
4 }
5
```

Body Cookies (1) Headers (7) Test Results 401 Unauthorized 181 ms 274 B

{ } JSON Preview Visualize

```
1 {
2   "message": "Senha inválida"
3 }
```

- Enviando ' OR '1'='1 no campo de email:



POST http://localhost:3000/api/auth/login

Params Authorization Headers (10) Body Scripts Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

```
1 {
2   "email": " OR '1'='1 ",
3   "senha": "Ana@2024"
4 }
5
```

Body Cookies (1) Headers (7) Test Results 401 Unauthorized 9 ms 280 B

{ } JSON Preview Visualize

```
1 {
2   "message": "Email não encontrado"
3 }
```

- Testar o token manipulado:

Objetivo: Verificar se o sistema detecta um token manipulado.

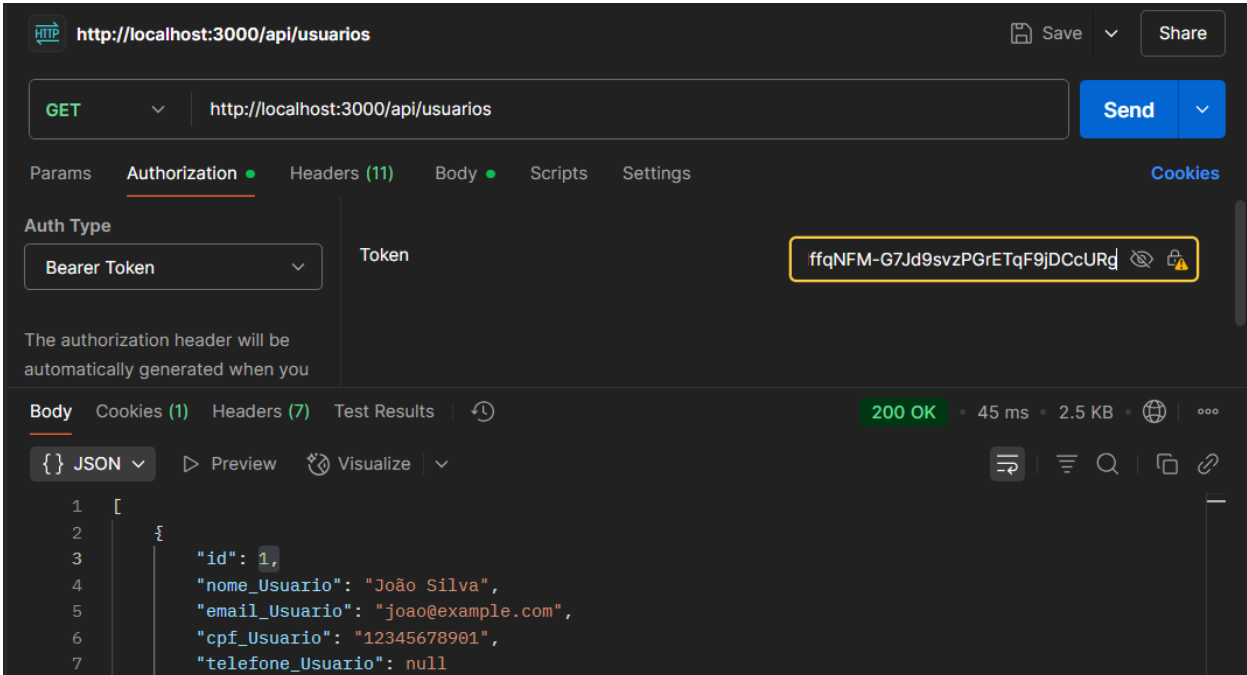
Cenário: GET /usuarios com um token inválido (último caractere do JWT modificado).

Resultado esperado: Não permitir o acesso, e exibir uma mensagem de erro de “Token inválido”.

Resultado obtido: O acesso foi corretamente bloqueado e a mensagem exibida, conforme o esperado.

Evidências:

- Token original gerado no login:

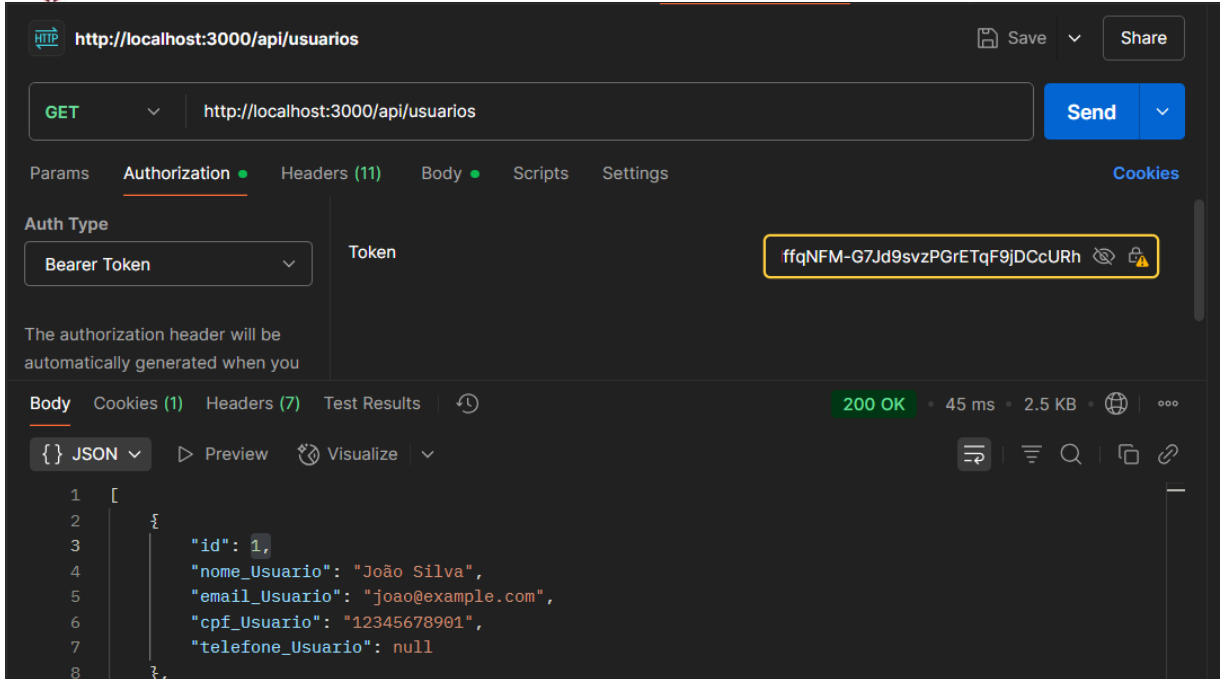


The screenshot shows a REST client interface with the following details:

- URL:** `http://localhost:3000/api/usuarios`
- Method:** GET
- Authorization:** Bearer Token with the token `ffqNFM-G7Jd9svzPGrETqF9jDCcURd` (highlighted with a yellow box).
- Response:** 200 OK, 45 ms, 2.5 KB.
- Body (JSON):**

```
[
  {
    "id": 1,
    "nome_usuario": "João Silva",
    "email_usuario": "joao@example.com",
    "cpf_usuario": "12345678901",
    "telefone_usuario": null
  }
]
```

- Alteração manual no último caractere do token (de “g” para “h”):



HTTP **http://localhost:3000/api/usuarios** Save Share

GET **http://localhost:3000/api/usuarios** Send

Params Authorization Headers (11) Body Scripts Settings Cookies

Auth Type: Bearer Token Token: ffqNFM-G7Jd9svzPGrETqF9jDCcURh

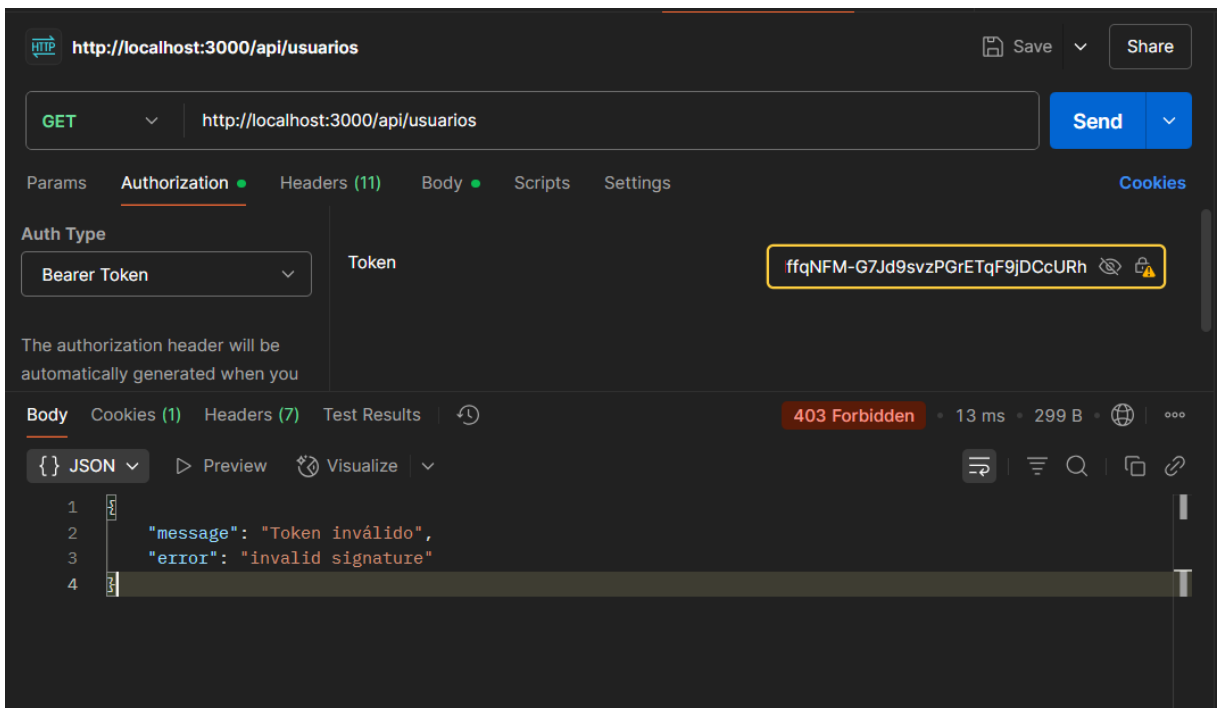
The authorization header will be automatically generated when you

Body Cookies (1) Headers (7) Test Results 200 OK • 45 ms • 2.5 KB

JSON Preview Visualize

```
1 [
2   {
3     "id": 1,
4     "nome_usuario": "João Silva",
5     "email_usuario": "joao@example.com",
6     "cpf_usuario": "12345678901",
7     "telefone_usuario": null
8   },
9 ]
```

- Requisição com token inválido resultando em erro de autenticação:



HTTP **http://localhost:3000/api/usuarios** Save Share

GET **http://localhost:3000/api/usuarios** Send

Params Authorization Headers (11) Body Scripts Settings Cookies

Auth Type: Bearer Token Token: ffqNFM-G7Jd9svzPGrETqF9jDCcURh

The authorization header will be automatically generated when you

Body Cookies (1) Headers (7) Test Results 403 Forbidden • 13 ms • 299 B

JSON Preview Visualize

```
1 [
2   {
3     "message": "Token inválido",
4     "error": "invalid signature"
5   },
6 ]
```


6.7 M-10 – Tempo Médio para Correção de Falhas

Durante os testes, foram identificadas algumas falhas no sistema, como, por exemplo, a aceitação de cadastro de reservas fora do horário de funcionamento do restaurante, permissão para cadastro de reservas duplicadas, entre outros. No momento da entrega deste relatório, essas falhas ainda não foram corrigidas, portanto não é possível calcular um tempo médio de correção. No entanto, todas as falhas foram documentadas, e serão corrigidas o quanto antes.

6.8 M-11 – Execução de Auditorias de Segurança

Foi realizada uma auditoria manual de segurança utilizando o Postman, com foco em testes de vulnerabilidade comuns, como SQL Injection, acesso sem token JWT e uso de tokens inválidos. Os testes demonstraram que o sistema conseguiu bloquear corretamente tentativas de ataque, como a injeção de código SQL e o uso de tokens incorretos, não sendo identificadas vulnerabilidades críticas.

Apesar disso, as auditorias foram limitadas e realizadas apenas com técnicas básicas, e talvez será necessária a realização de testes mais aprofundados no futuro.

Todos os testes realizados até o momento foram descritos ao longo deste documento.

6.9 Testes Extras

Além dos testes planejados no escopo original da auditoria, foram realizados testes complementares com o intuito de verificar o comportamento do sistema em situações reais de uso, bem como avaliar a segurança, resistência e aderência às regras de negócio.

Abaixo, estão descritos os testes extras conduzidos durante a fase de validação do sistema:

6.9.1 Login com Dados Válidos

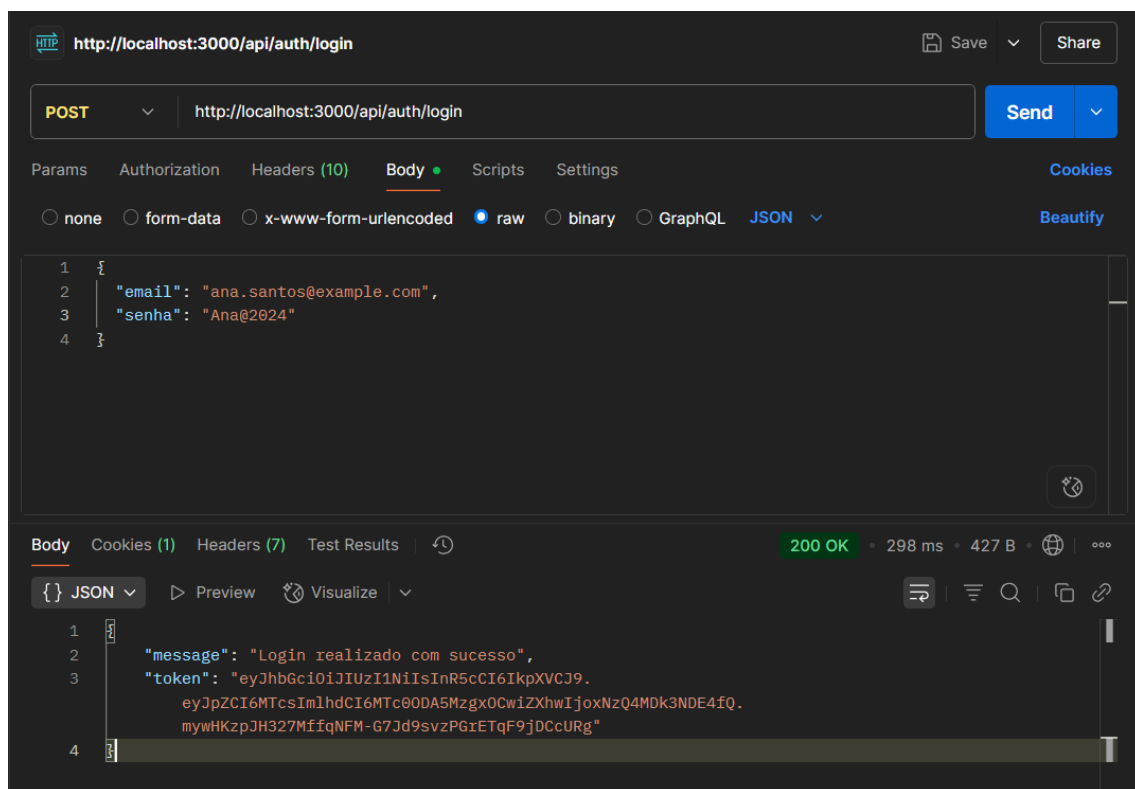
Objetivo: Verificar se o sistema autentica corretamente usuários com credenciais válidas.

Cenário: Requisição **POST /login** com email e senha corretos.

Resultado esperado: Login efetuado com sucesso.

Resultado obtido: Login efetuado com sucesso.

Evidências:



6.9.2 Login com Email Não Existente

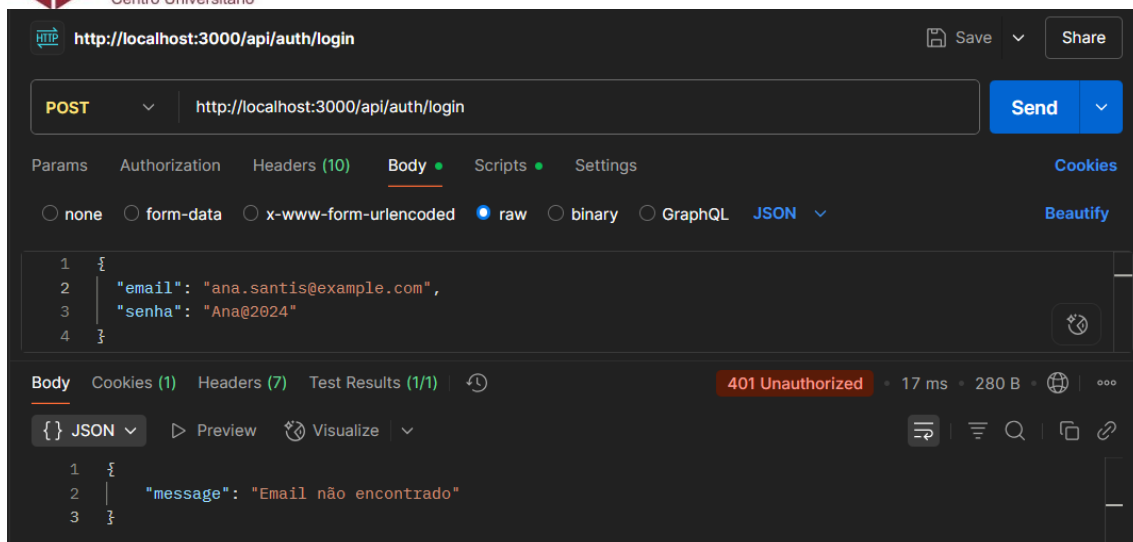
Objetivo: Verificar resposta ao tentar autenticar um usuário inexistente.

Cenário: Requisição **POST /login** com email inválido.

Resultado esperado: Não aceitar o login, e exibir uma mensagem de erro de “Email não encontrado”.

Resultado obtido: Bloqueio de login e mensagem retornada, conforme esperado.

Evidência:



6.9.3 Login com Senha Incorreta

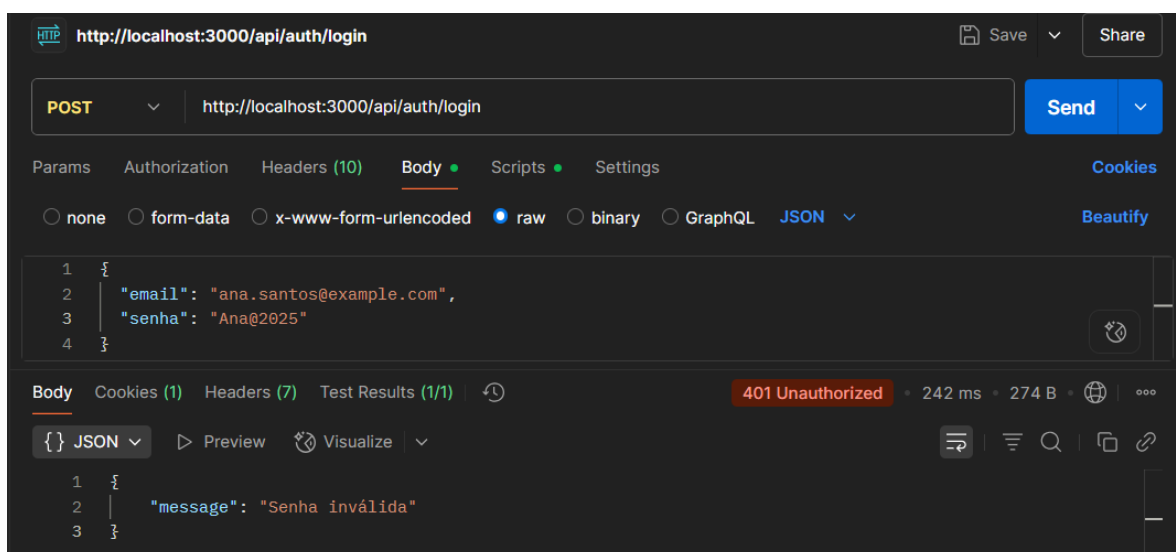
Objetivo: Verificar resposta ao inserir senha errada.

Cenário: Requisição **POST /login** com senha inválida.

Resultado esperado: Não aceitar o login, e exibir uma mensagem de erro de “Senha inválida”.

Resultado obtido: Bloqueio de login e mensagem retornada, conforme esperado.

Evidência:



6.9.4 Acesso à Rota Protegida com Token Válido

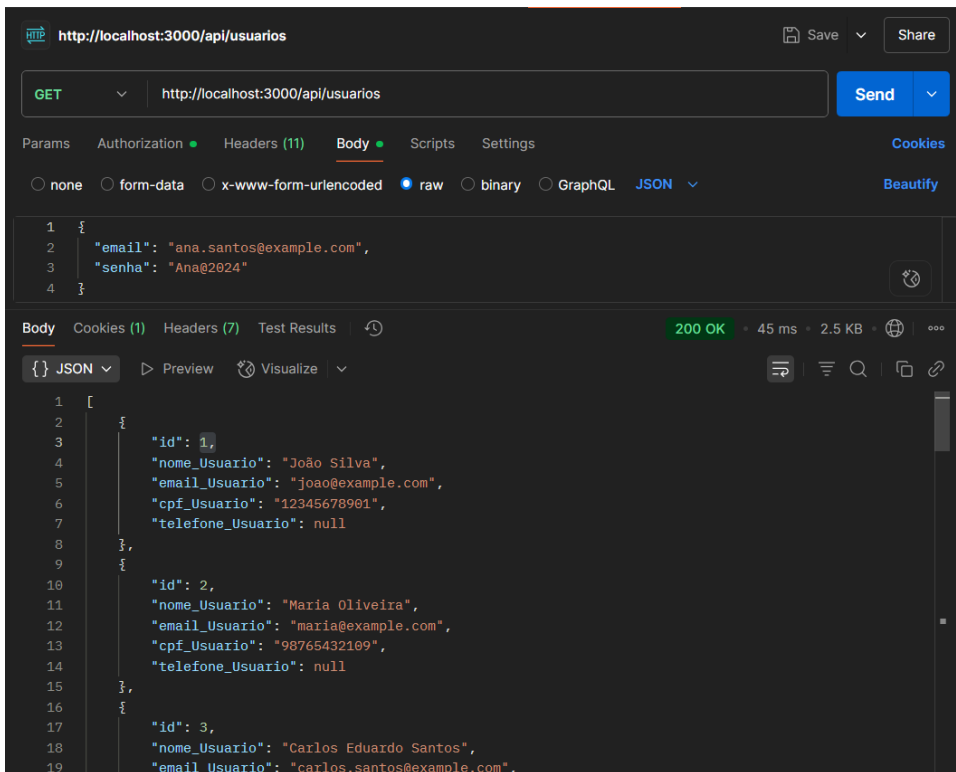
Objetivo: Validar se o token JWT permite acesso às rotas privadas (neste caso, listagem de usuários).

Cenário: GET /usuarios com token válido no cabeçalho.

Resultado esperado: Retorno da lista de usuários.

Resultado obtido: Listagem realizada com sucesso.

Evidência:



```
1 {}
2 {"email": "ana.santos@example.com",
3  "senha": "Ana@2024"}
4 {}

Body Cookies (1) Headers (7) Test Results
200 OK 45 ms 2.5 KB
JSON Preview Visualize
1 [
2   {
3     "id": 1,
4     "nome_usuario": "João Silva",
5     "email_usuario": "joao@example.com",
6     "cpf_usuario": "12345678901",
7     "telefone_usuario": null
8   },
9   {
10    "id": 2,
11    "nome_usuario": "Maria Oliveira",
12    "email_usuario": "maria@example.com",
13    "cpf_usuario": "98765432109",
14    "telefone_usuario": null
15  },
16  {
17    "id": 3,
18    "nome_usuario": "Carlos Eduardo Santos",
19    "email_usuario": "carlos.santos@example.com",
```

6.9.5 Acesso à Rota Protegida sem Token

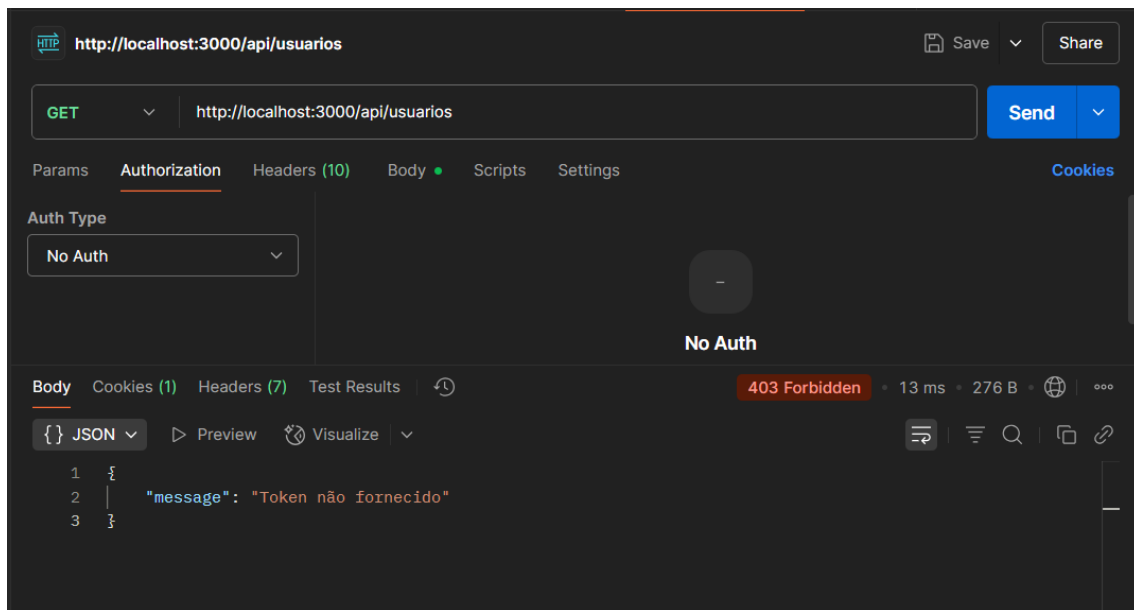
Objetivo: Garantir que rotas protegidas bloqueiem acessos sem autenticação.

Cenário: GET /usuarios sem envio de token no cabeçalho.

Resultado esperado: Não permitir o acesso, e exibir uma mensagem de erro.

Resultado obtido: Acesso não permitido e mensagem exibida, conforme o esperado.

Evidência:



6.9.6 CRUD de Clientes

Objetivo: Verificar operações básicas de criação, leitura, atualização e exclusão de usuários.

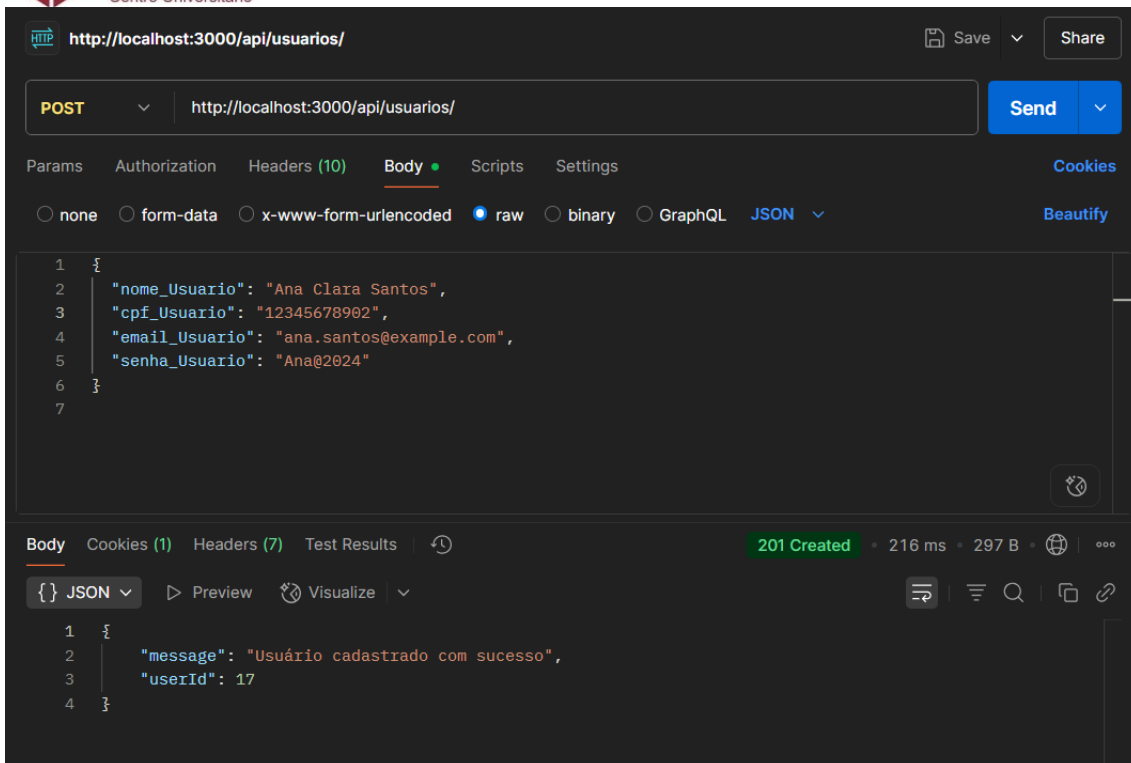
Cenários:

- **POST /usuarios** com dados válidos
 - **Resultado esperado:** Cadastro bem-sucedido.
- **PUT /usuarios/:id** com novos dados
 - **Resultado esperado:** Atualização bem-sucedida.
- **DELETE /usuarios/:id**
 - **Resultado esperado:** Usuário deletado com sucesso.
- **GET /usuarios**
 - **Resultado esperado:** Todos os usuários serem retornados.

Resultado obtido: Todas as operações foram bem-sucedidas.

Evidências:

- Cadastro com dados válidos:



HTTP <http://localhost:3000/api/usuarios/> Save Share

POST <http://localhost:3000/api/usuarios/> Send

Params Authorization Headers (10) Body Scripts Settings Cookies

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL JSON Beautify

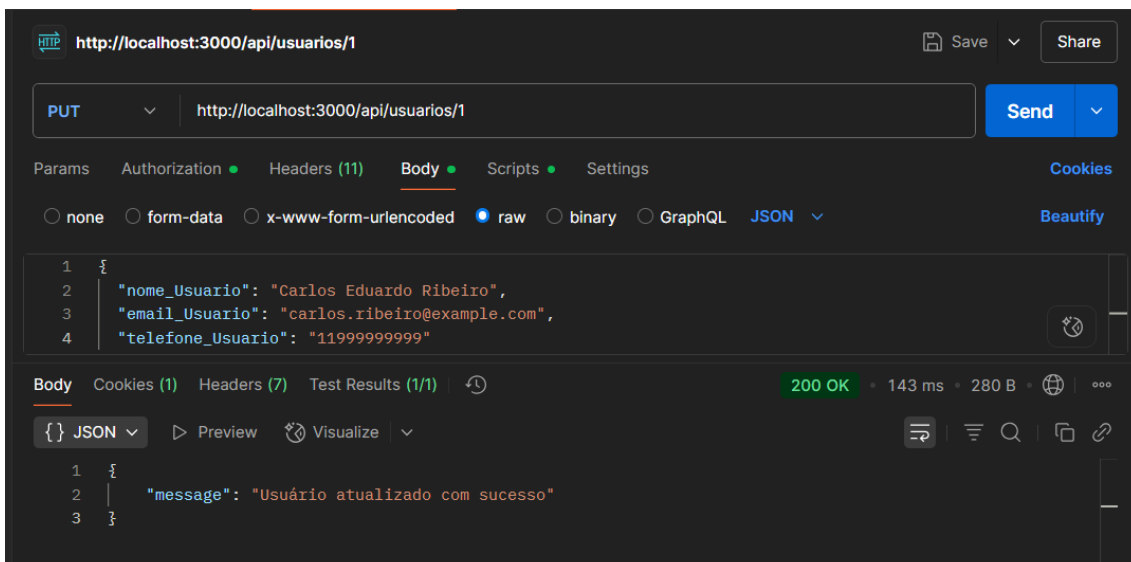
```
1 {
2   "nome_usuario": "Ana Clara Santos",
3   "cpf_usuario": "12345678902",
4   "email_usuario": "ana.santos@example.com",
5   "senha_usuario": "Ana@2024"
6 }
7
```

Body Cookies (1) Headers (7) Test Results 201 Created • 216 ms • 297 B

{ JSON Preview Visualize

```
1 {
2   "message": "Usuário cadastrado com sucesso",
3   "userId": 17
4 }
```

- Editar um usuário existente:



HTTP <http://localhost:3000/api/usuarios/1> Save Share

PUT <http://localhost:3000/api/usuarios/1> Send

Params Authorization Headers (11) Body Scripts Settings Cookies

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL JSON Beautify

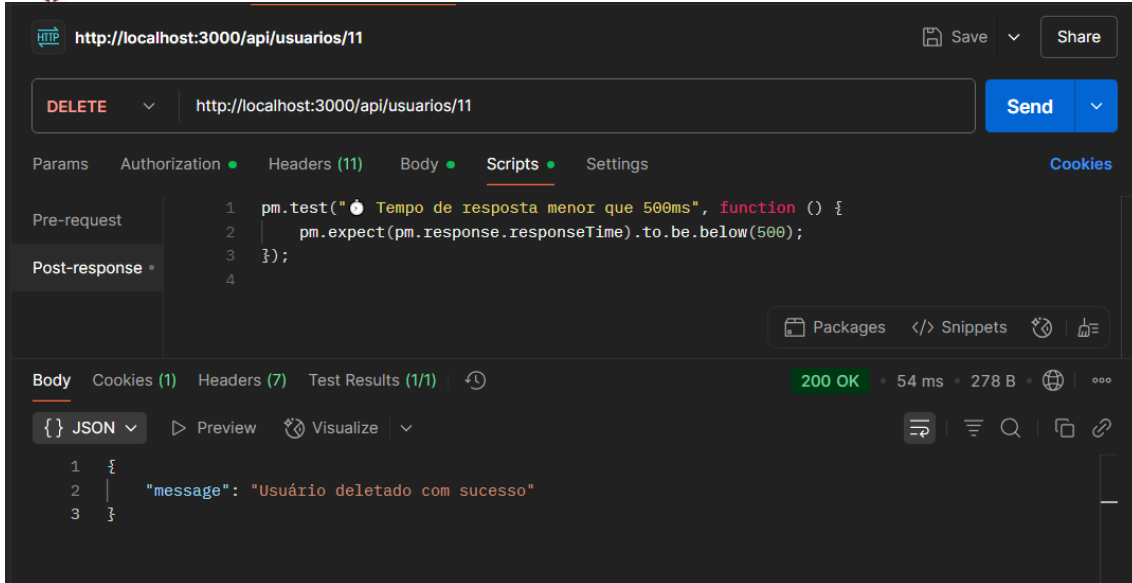
```
1 {
2   "nome_usuario": "Carlos Eduardo Ribeiro",
3   "email_usuario": "carlos.ribeiro@example.com",
4   "telefone_usuario": "11999999999"
}
```

Body Cookies (1) Headers (7) Test Results (1/1) 200 OK • 143 ms • 280 B

{ JSON Preview Visualize

```
1 {
2   "message": "Usuário atualizado com sucesso"
3 }
```

- Excluir um usuário:



HTTP <http://localhost:3000/api/usuarios/11> Save Share

DELETE <http://localhost:3000/api/usuarios/11> Send

Params Authorization Headers (11) Body Scripts Settings Cookies

Pre-request

```
1 pm.test("⚡ Tempo de resposta menor que 500ms", function () {  
2   pm.expect(pm.response.responseTime).to.be.below(500);  
3 }  
4
```

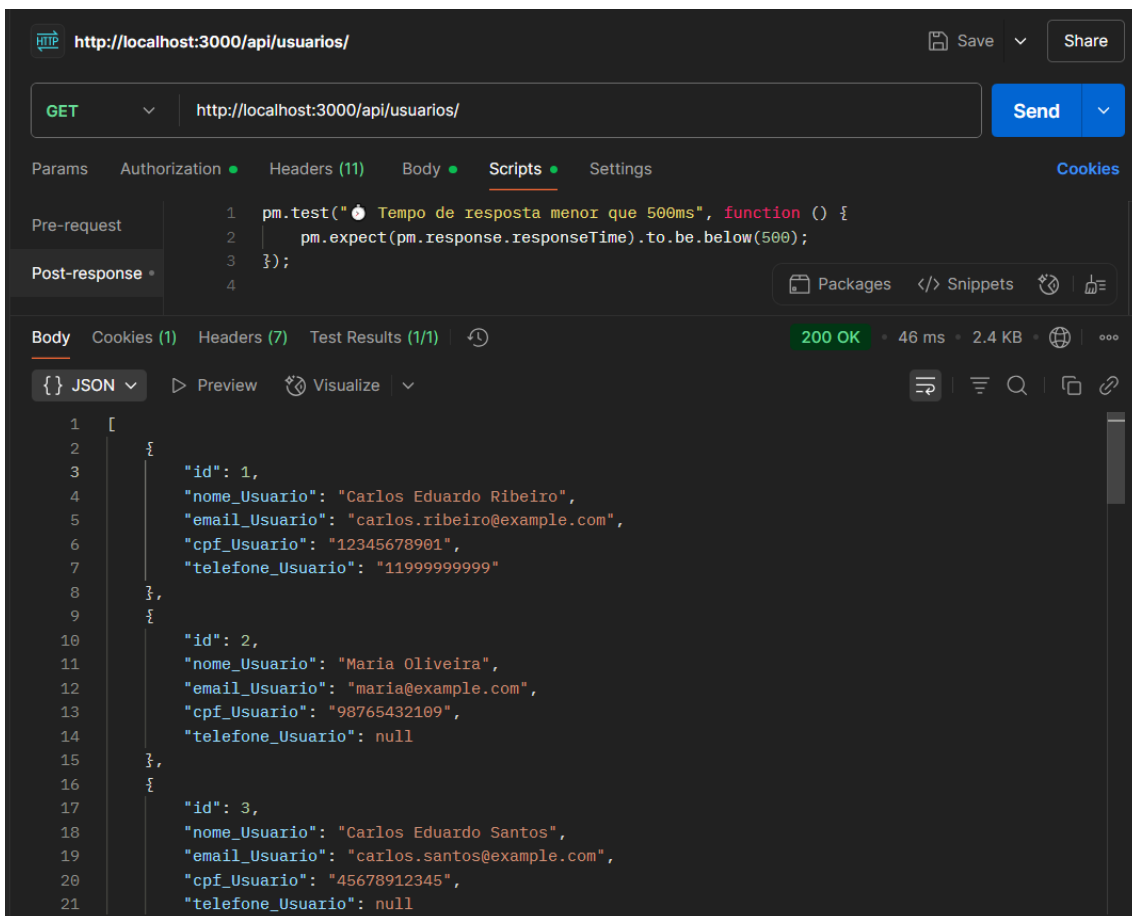
Post-response

200 OK • 54 ms • 278 B • 🌐 ⋮

{ } JSON Preview Visualize

```
1 {  
2   "message": "Usuário deletado com sucesso"  
3 }
```

- Consultar lista de usuários:



HTTP <http://localhost:3000/api/usuarios/> Save Share

GET <http://localhost:3000/api/usuarios/> Send

Params Authorization Headers (11) Body Scripts Settings Cookies

Pre-request

```
1 pm.test("⚡ Tempo de resposta menor que 500ms", function () {  
2   pm.expect(pm.response.responseTime).to.be.below(500);  
3 }  
4
```

Post-response

200 OK • 46 ms • 2.4 KB • 🌐 ⋮

{ } JSON Preview Visualize

```
1 [  
2   {  
3     "id": 1,  
4     "nome_usuario": "Carlos Eduardo Ribeiro",  
5     "email_usuario": "carlos.ribeiro@example.com",  
6     "cpf_usuario": "12345678901",  
7     "telefone_usuario": "11999999999"  
8   },  
9   {  
10    "id": 2,  
11    "nome_usuario": "Maria Oliveira",  
12    "email_usuario": "maria@example.com",  
13    "cpf_usuario": "98765432109",  
14    "telefone_usuario": null  
15  },  
16  {  
17    "id": 3,  
18    "nome_usuario": "Carlos Eduardo Santos",  
19    "email_usuario": "carlos.santos@example.com",  
20    "cpf_usuario": "45678912345",  
21    "telefone_usuario": null  
22  }  
23 ]
```

```
23  {
24    "id": 4,
25    "nome_usuario": "Pedro Henrique Lima",
26    "email_usuario": "pedro.lima@gmail.com",
27    "cpf_usuario": "32165498700",
28    "telefone_usuario": null
29  },
30  {
31    "id": 5,
32    "nome_usuario": "Francisco Almeida",
33    "email_usuario": "chico.almeida@example.com",
34    "cpf_usuario": "11122233344",
35    "telefone_usuario": null
36  },
37  {
38    "id": 6,
39    "nome_usuario": "MARIELE VIEIRASILVA",
40    "email_usuario": "silvamariele797@gmail.com",
41    "cpf_usuario": "04967345204",
42    "telefone_usuario": "47999660394"
43  },
44 }
```

6.9.7 Cadastro de Usuários com Dados Inválidos

Objetivo: Avaliar se o sistema valida campos como CPF e email.

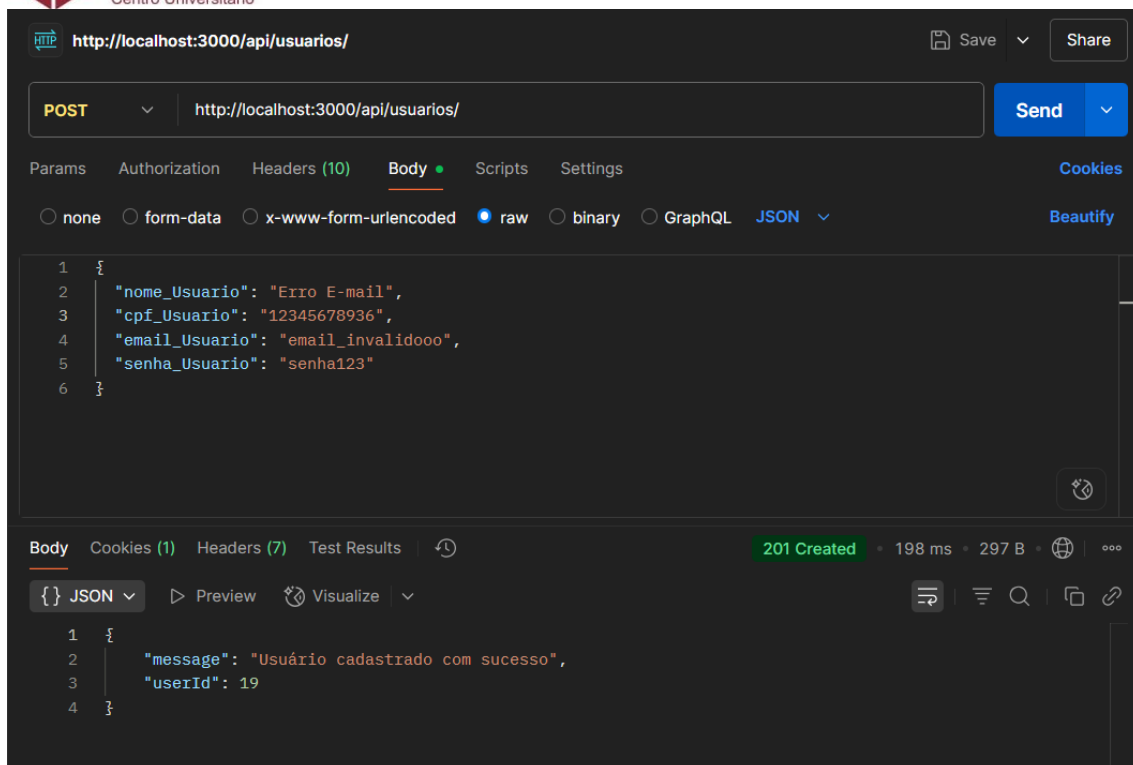
Cenário: **POST /usuarios** com dados inválidos (ex.: CPF inválido, e-mail mal formatado).

Resultado esperado: Rejeição da requisição com erro de validação.

Resultado obtido: O sistema permitiu o cadastro com dados inválidos.

Evidência:

- Tentar cadastrar usuário com e-mail inválido:



6.9.8 CRUD de Reservas

Objetivo: Verificar se a API executa corretamente as operações CRUD de reservas.

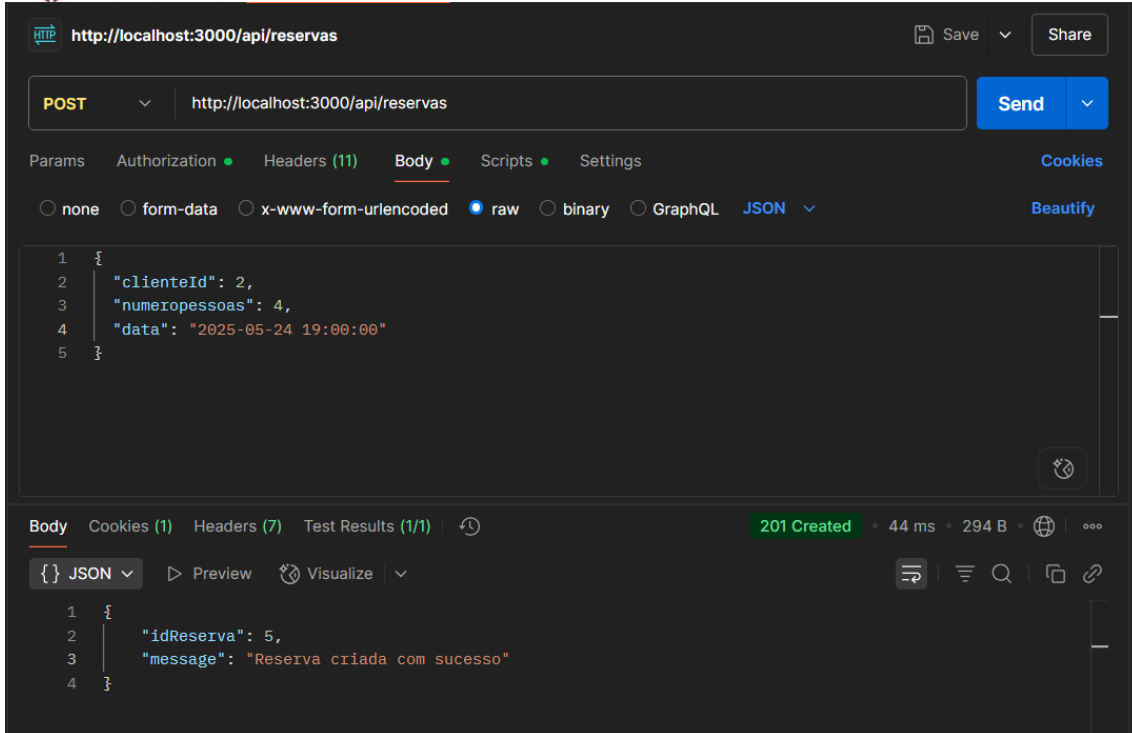
Cenários:

- **POST /reserva** com dados válidos
 - **Resultado esperado:** Reserva criada com sucesso.
- **PUT /reserva/:id** com novos dados
 - **Resultado esperado:** Atualização bem-sucedida.
- **DELETE /reserva/:id**
 - **Resultado esperado:** Reserva cancelada com sucesso.
- **GET /reserva**
 - **Resultado esperado:** Todas as reservas serem retornados.

Resultado obtido: Todas as operações foram bem-sucedidas.

Evidências:

- Criar reserva com as informações necessárias válidas:



The screenshot shows a REST client interface with the URL `http://localhost:3000/api/reservas`. The method is set to **POST**. The request body is a JSON object:

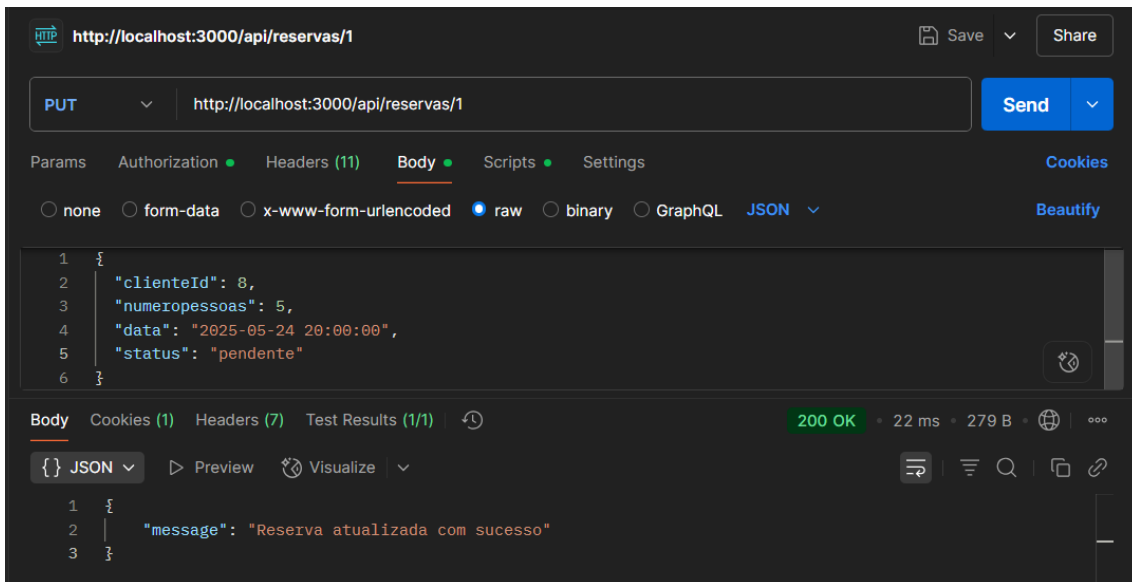
```
{  "clienteId": 2,  "numeropessoas": 4,  "data": "2025-05-24 19:00:00"}
```

. The response status is **201 Created** with a response body:

```
{  "idReserva": 5,  "message": "Reserva criada com sucesso"}
```

. The response is displayed in JSON format.

- Editar uma reserva:



The screenshot shows a REST client interface with the URL `http://localhost:3000/api/reservas/1`. The method is set to **PUT**. The request body is a JSON object:

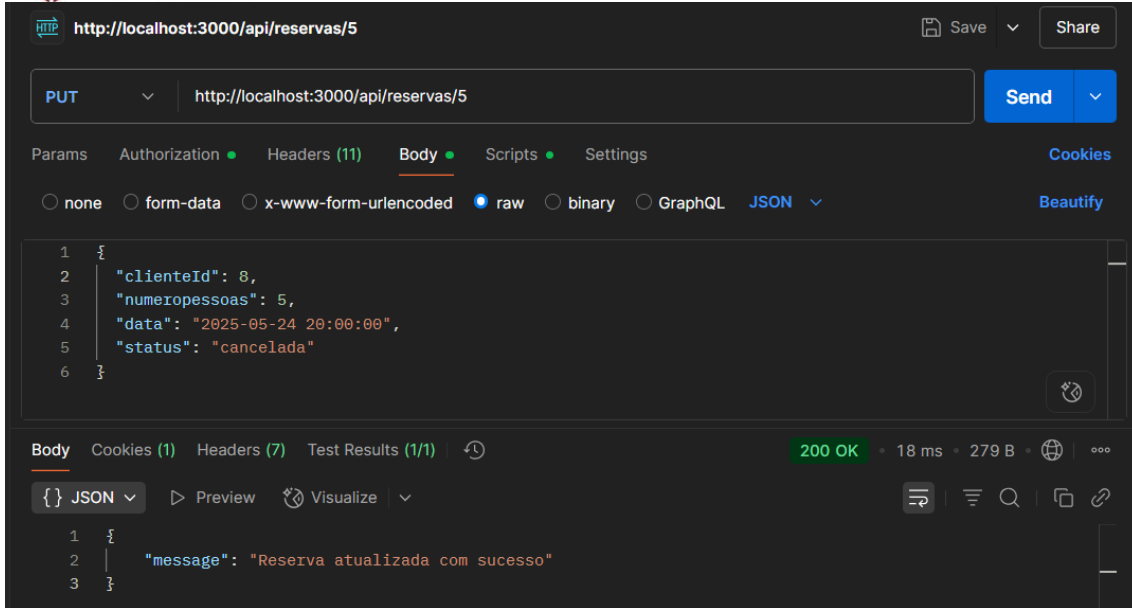
```
{  "clienteId": 8,  "numeropessoas": 5,  "data": "2025-05-24 20:00:00",  "status": "pendente"}
```

. The response status is **200 OK** with a response body:

```
{  "message": "Reserva atualizada com sucesso"}
```

. The response is displayed in JSON format.

- Cancelar uma reserva:



HTTP <http://localhost:3000/api/reservas/5> Save Share

PUT <http://localhost:3000/api/reservas/5> Send

Params Authorization Headers (11) Body Scripts Settings Cookies

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL JSON Beautify

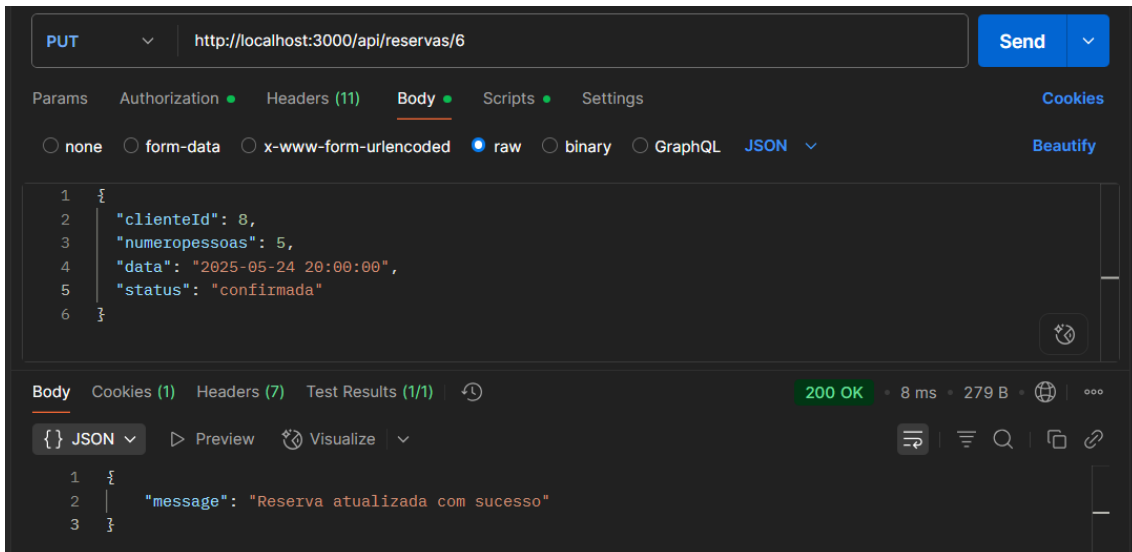
```
1 {
2   "clienteId": 8,
3   "numeroPessoas": 5,
4   "data": "2025-05-24 20:00:00",
5   "status": "cancelada"
6 }
```

Body Cookies (1) Headers (7) Test Results (1/1) 200 OK • 18 ms • 279 B

{ JSON Preview Visualize

```
1 {
2   "message": "Reserva atualizada com sucesso"
3 }
```

- Confirmar uma reserva:



PUT <http://localhost:3000/api/reservas/6> Send

Params Authorization Headers (11) Body Scripts Settings Cookies

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL JSON Beautify

```
1 {
2   "clienteId": 8,
3   "numeroPessoas": 5,
4   "data": "2025-05-24 20:00:00",
5   "status": "confirmada"
6 }
```

Body Cookies (1) Headers (7) Test Results (1/1) 200 OK • 8 ms • 279 B

{ JSON Preview Visualize

```
1 {
2   "message": "Reserva atualizada com sucesso"
3 }
```

- Listar reservas:

GET ▼ http://localhost:3000/api/reservas Send ▼

Params Authorization Headers (11) **Body** Scripts Settings Cookies

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL JSON ▼ Beautify

```
1 {
2   "clienteId": 8,
3   "numeropessoas": 5,
4   "data": "2025-05-24 20:00:00",
5   "status": "cancelada"
6 }
```

Body Cookies (1) Headers (7) Test Results (1/1) 200 OK • 17 ms • 953 B • 🌐 ⋮

{} JSON ▼ ▶ Preview 🔗 Visualize ▼ ↩ ☰ 🔍 📄 🔗

```
1 [
2   {
3     "id": 1,
4     "numeropessoas_Reserva": 5,
5     "data_Reserva": "2025-05-24T23:00:00.000Z",
6     "status_Reserva": "pendente",
7     "nome_usuario": "MARIELE VIEIRASILVA"
8   },
9   {
10    "id": 3,
11    "numeropessoas_Reserva": 2,
12    "data_Reserva": "2025-05-20T23:30:00.000Z",
13    "status_Reserva": "pendente",
14    "nome_usuario": "MARIELE VIEIRASILVA"
15  },
16 ]
```

GET ▼ http://localhost:3000/api/reservas Send ▼

Params Authorization Headers (11) **Body** Scripts Settings Cookies

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL JSON ▼ Beautify

```
1 {
2   "clienteId": 8,
3   "numeropessoas": 5,
4   "data": "2025-05-24 20:00:00",
5   "status": "cancelada"
6 }
```

Body Cookies (1) Headers (7) Test Results (1/1) 200 OK • 17 ms • 953 B • 🌐 ⋮

{} JSON ▼ ▶ Preview 🔗 Visualize ▼ ↩ ☰ 🔍 📄 🔗

```
22 ],
23 {
24   "id": 5,
25   "numeropessoas_Reserva": 5,
26   "data_Reserva": "2025-05-24T23:00:00.000Z",
27   "status_Reserva": "cancelada",
28   "nome_usuario": "Maria Alice Giuliani"
29 },
30 {
31   "id": 6,
32   "numeropessoas_Reserva": 5,
33   "data_Reserva": "2025-05-24T23:00:00.000Z",
34   "status_Reserva": "confirmada",
35   "nome_usuario": "Maria Alice Giuliani"
36 }
37 ]
```

6.9.9 Verificação de Status de Reservas

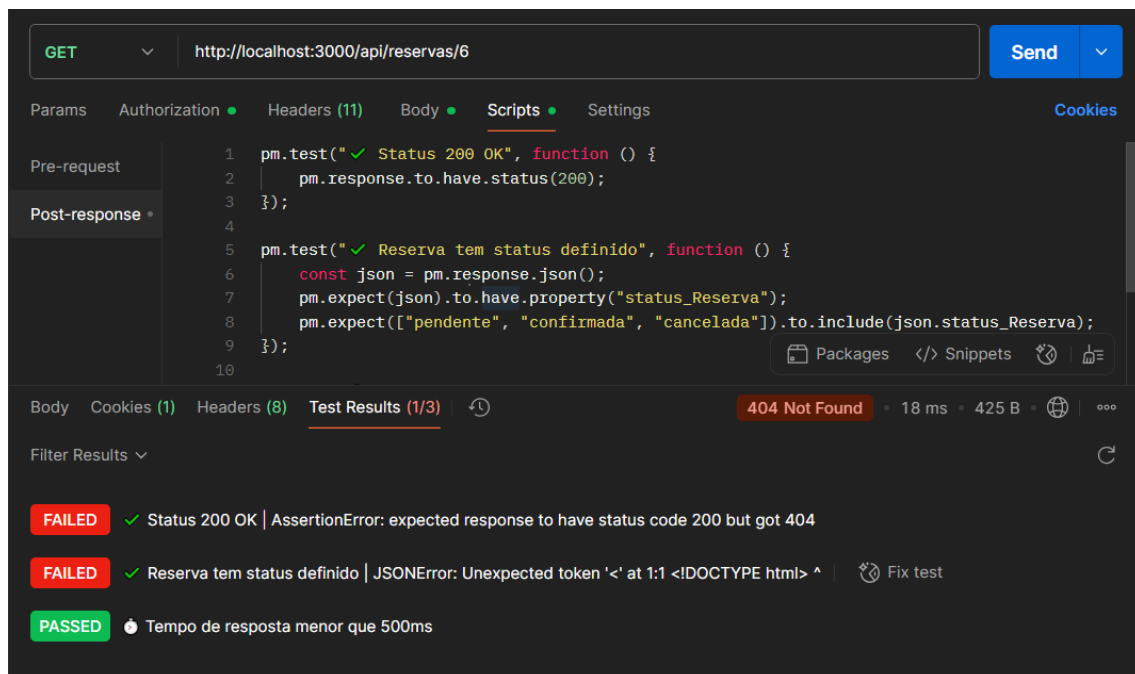
Objetivo: Verificar se o sistema retorna corretamente o status de uma reserva.

Cenário: Requisição **GET** para **/api/reservas/6**, com ID de uma reserva previamente cadastrada.

Resultado esperado: Status da reserva (pendente, confirmada ou cancelada) retornada.

Resultado obtido: Reserva com ID 6 não encontrada, impedindo a validação.

Evidência:



6.9.10 Criar Reserva Fora do Expediente

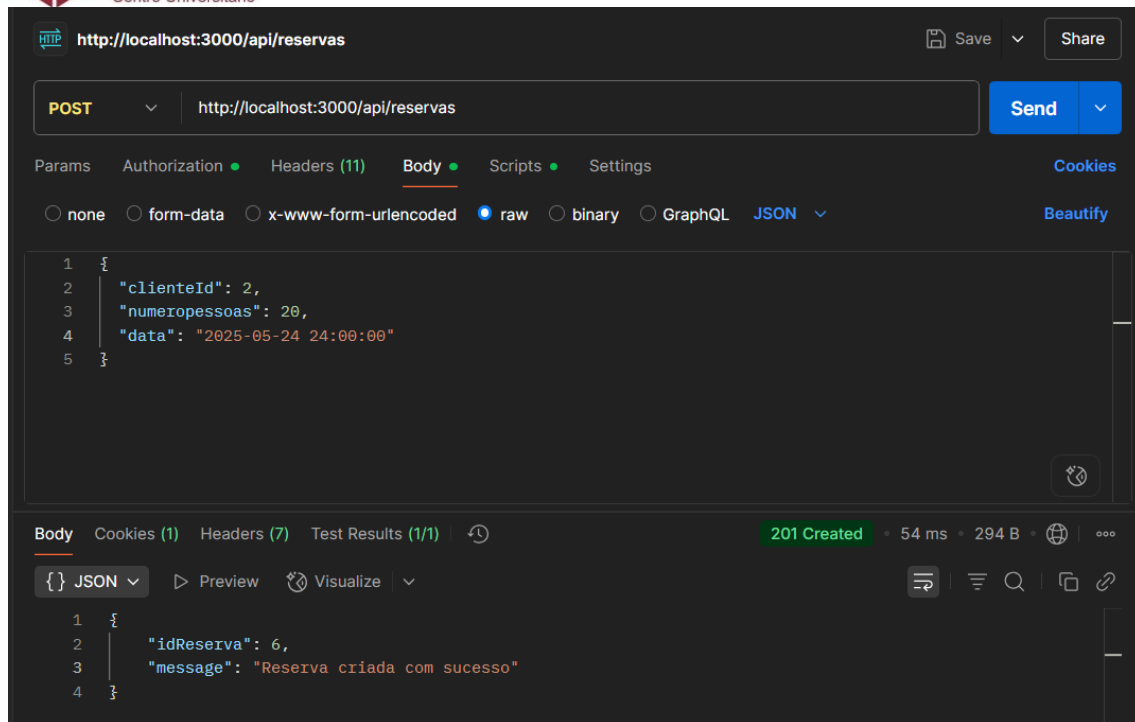
Objetivo: Verificar se o sistema impede reservas em horários inválidos.

Cenário: **POST /reserva** com horário fora do funcionamento do restaurante.

Resultado esperado: Impedimento de reserva e mensagem de erro.

Resultado obtido: O sistema aceitou a reserva.

Evidência:



6.9.11 Criar Reservas Duplicadas

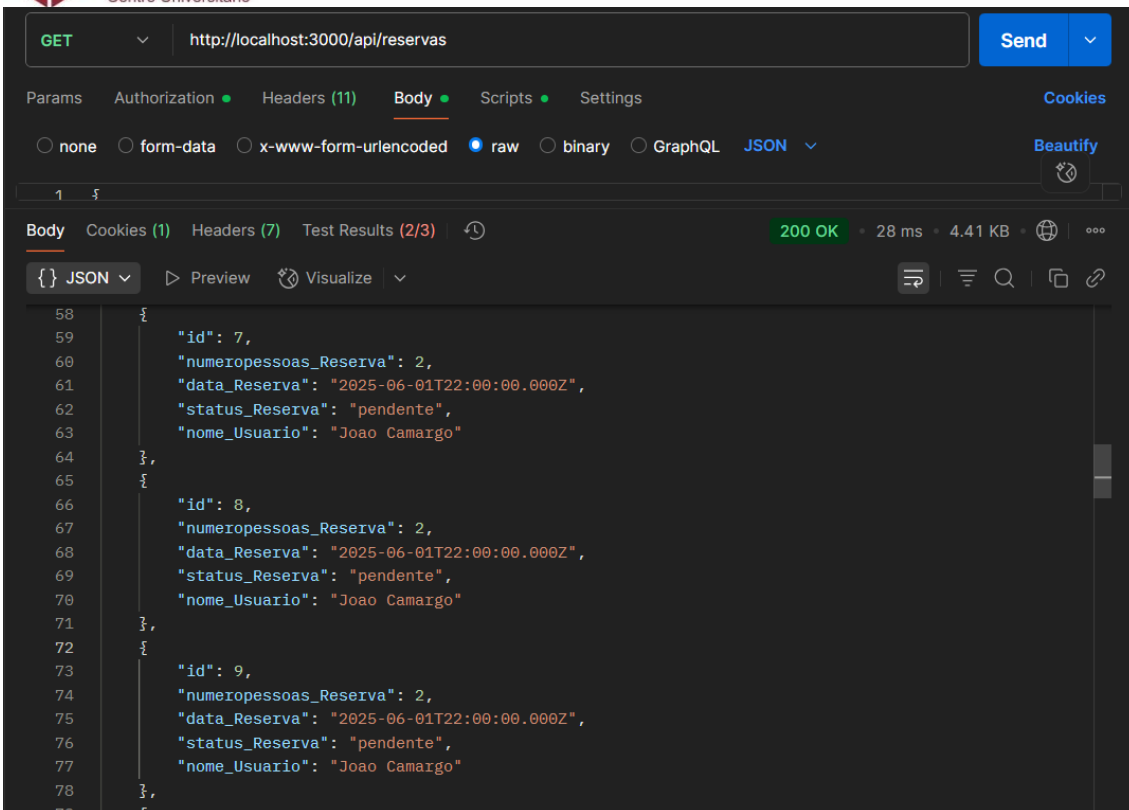
Objetivo: Verificar se o sistema impede conflito de horários para o mesmo cliente, na mesma data e mesmo horário.

Cenário: `POST /reserva` mais de uma vez com os mesmos dados.

Resultado esperado: Impedimento de cadastro da segunda reserva, e uma mensagem de erro.

Resultado obtido: Todas as reservas foram aceitas.

Evidência:



```
1  {
58  {
59    "id": 7,
60    "numero_pessoas_Reserva": 2,
61    "data_Reserva": "2025-06-01T22:00:00.000Z",
62    "status_Reserva": "pendente",
63    "nome_usuario": "Joao Camargo"
64  },
65  {
66    "id": 8,
67    "numero_pessoas_Reserva": 2,
68    "data_Reserva": "2025-06-01T22:00:00.000Z",
69    "status_Reserva": "pendente",
70    "nome_usuario": "Joao Camargo"
71  },
72  {
73    "id": 9,
74    "numero_pessoas_Reserva": 2,
75    "data_Reserva": "2025-06-01T22:00:00.000Z",
76    "status_Reserva": "pendente",
77    "nome_usuario": "Joao Camargo"
78  },
79  }
```

6.9.12 Cadastro com CPF já existente

Objetivo: Garantir que um CPF só possa ser cadastrado uma única vez.

Cenário: POST /usuario com um CPF já utilizado anteriormente.

Resultado esperado: Rejeição do cadastro, e uma mensagem de erro.

Resultado obtido: Impedimento de cadastro, com uma mensagem de erro de “Erro ao criar usuário”

Evidência:

HTTP <http://localhost:3000/api/usuarios/> Save Share

POST <http://localhost:3000/api/usuarios/> Send

Params Authorization Headers (10) **Body** Scripts Settings Cookies

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL **JSON** Beautify

```
1 {
2   "nome_usuario": "Ana Clara Santos",
3   "cpf_usuario": "12345678901",
4   "email_usuario": "ana.santos@example.com",
5   "senha_usuario": "Ana@2024"
6 }
7
```

Body Cookies (1) Headers (7) Test Results 500 Internal Server Error • 509 ms • 352 B • Visualize

{ } JSON Preview Visualize

```
1 {
2   "error": "Erro ao criar usuário",
3   "details": "Duplicate entry '12345678901' for key 'cpf_usuario'"
4 }
```