

UNOESC- Universidade do Oeste de Santa Catarina
Campus de São Miguel do Oeste - SC
Ciências da Computação

Modelagem de Reserva de salas e espaços

Engenharia de Software II
Prof. Roberson Junior Fernandes Alves

Cauana Rosin Ghizzi e Natani Gayardo

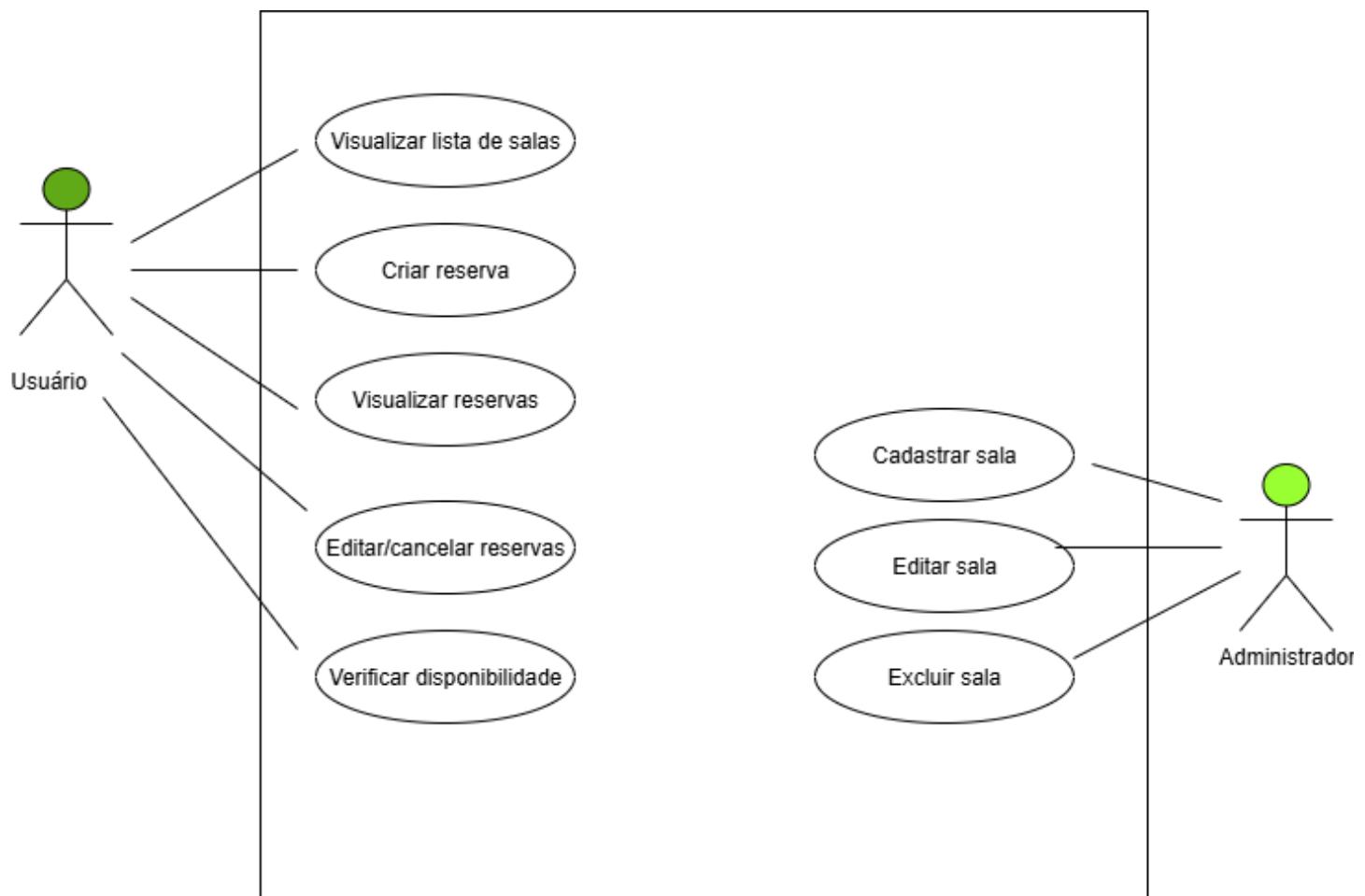
2025

1 INTRODUÇÃO

Neste trabalho final foi aplicado os conceitos aprendidos na disciplina de Engenharia de Software II na modelagem, validação e aprimoramento de um sistema real, integrando as atividades com o desenvolvimento realizado na disciplina de Programação III.

2. MODELAGEM DE SOFTWARE

2.1 DIAGRAMA DE CASOS DE USO



Casos de uso para o Usuário:

- **Visualizar lista de salas:** ver todas as salas disponíveis no sistema.
- **Criar reserva:** preencher e enviar um formulário para reservar uma sala.
- **Visualizar reservas:** acessar uma lista com suas reservas atuais ou passadas.
- **Editar/Cancelar reserva:** modificar ou cancelar uma reserva previamente feita.
- **Verificar disponibilidade:** consultar se há vaga para uma sala em um horário específico.

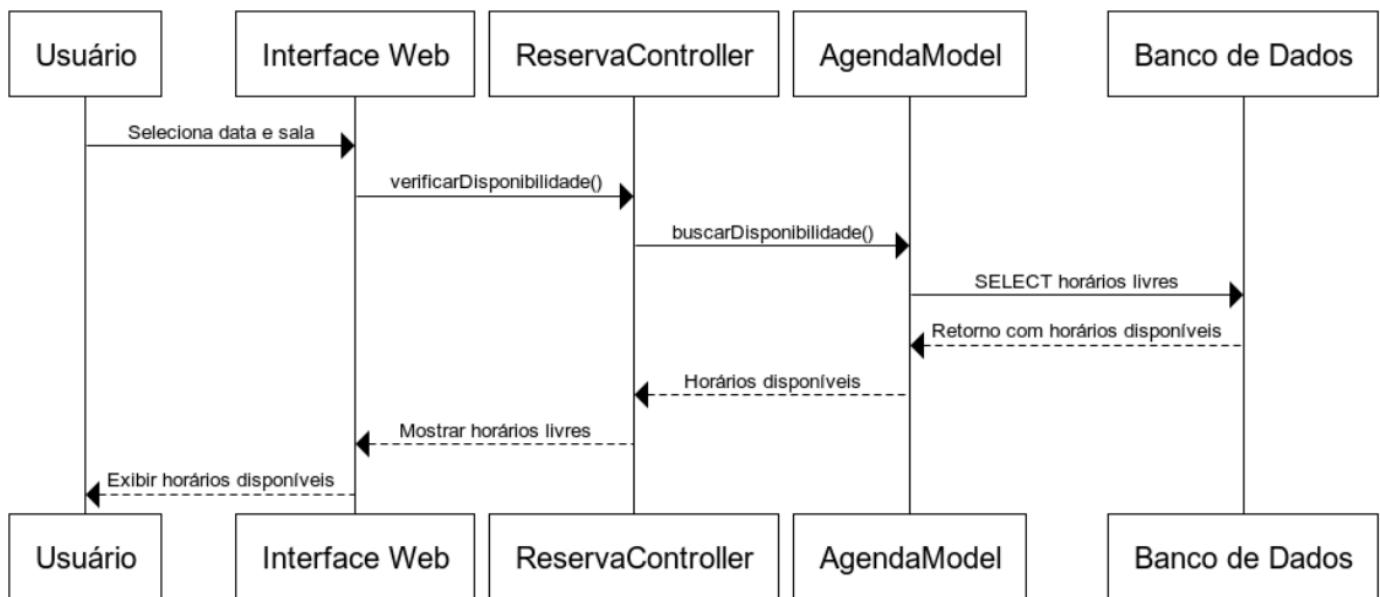
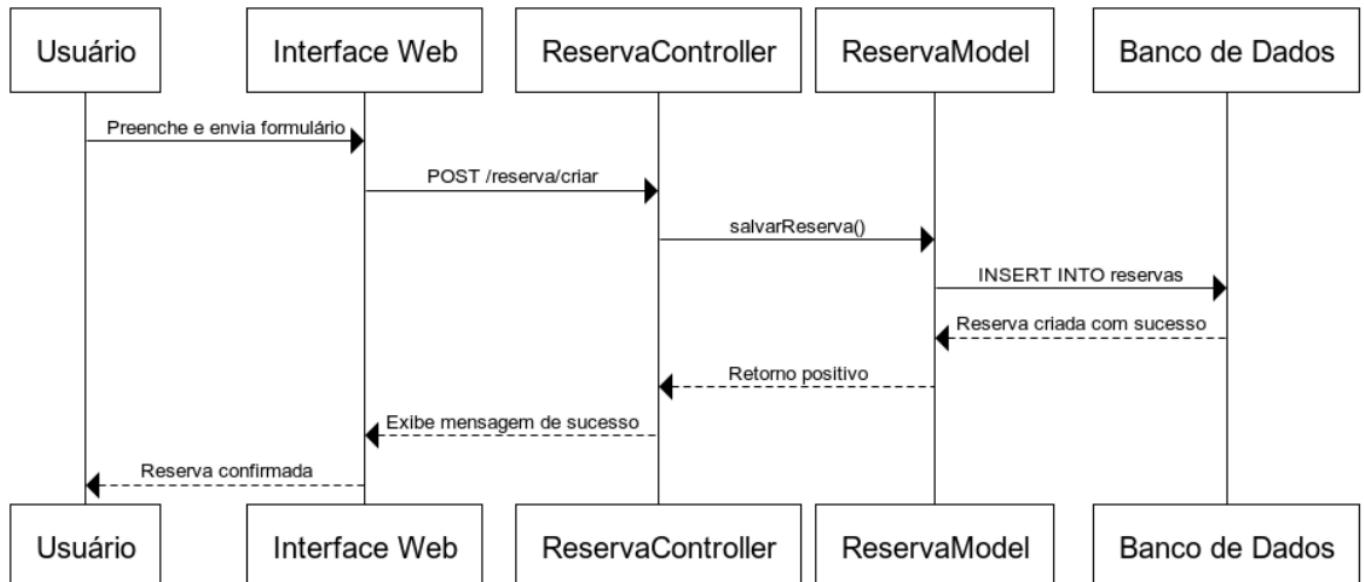
Casos de uso para o Administrador:

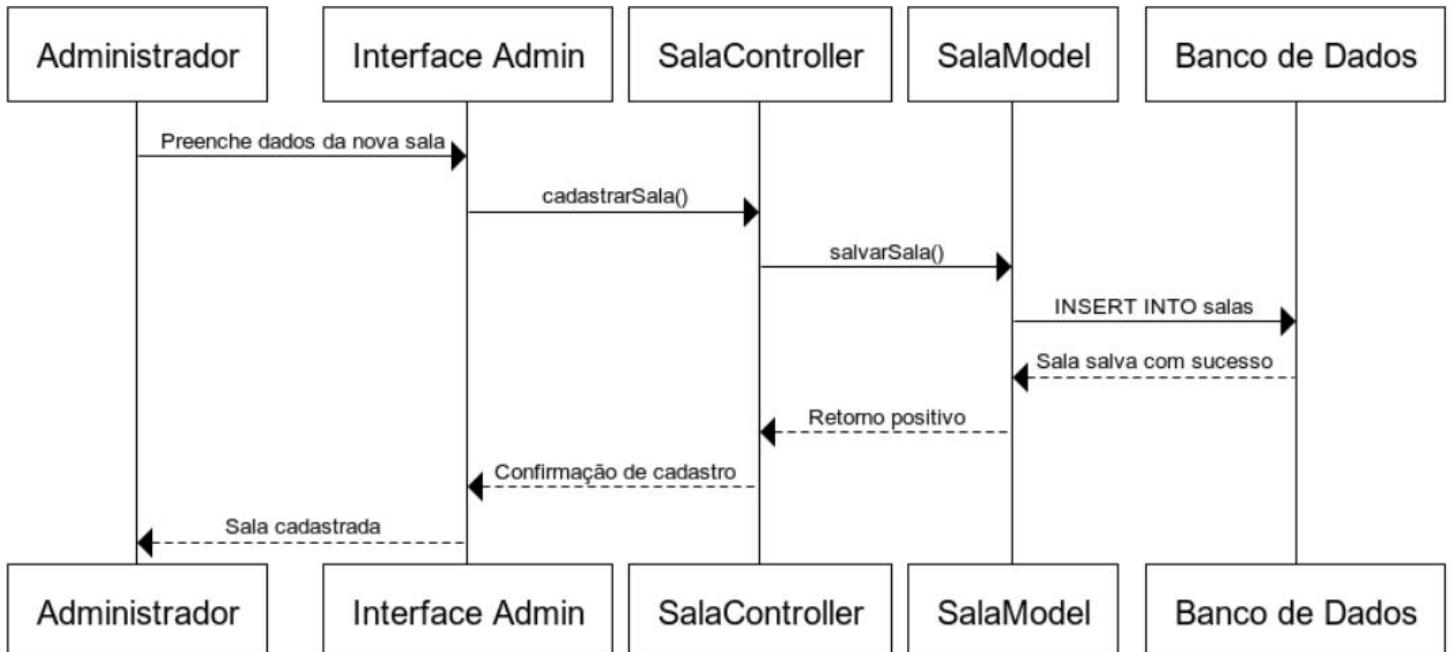
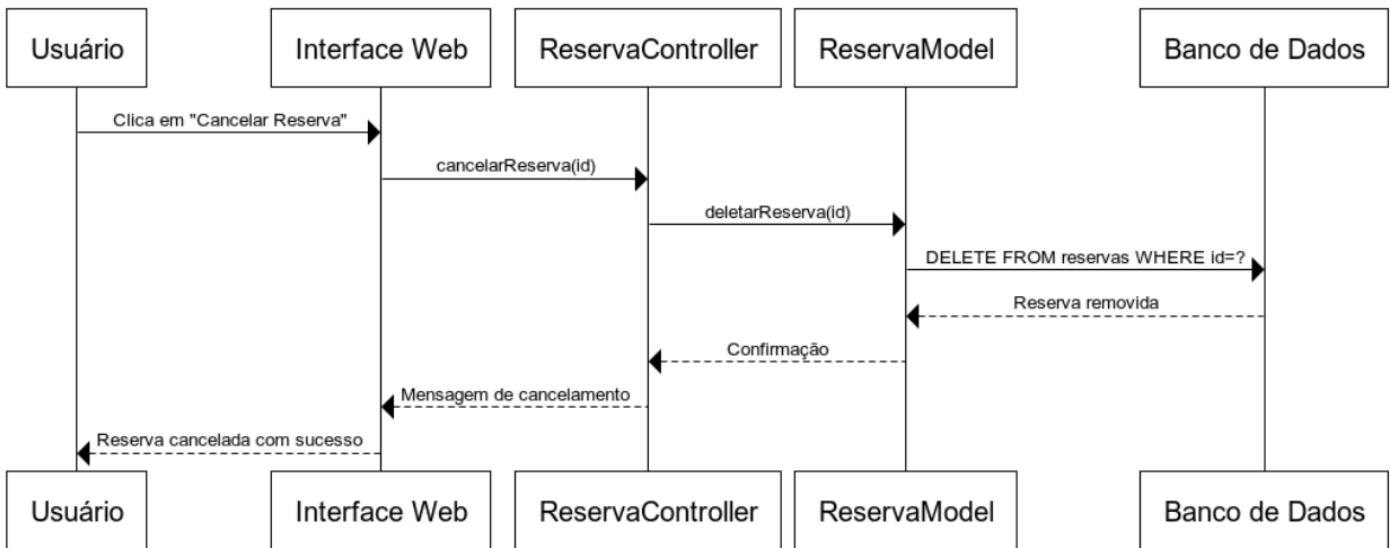
- **Cadastrar sala:** adicionar novas salas ao sistema.
- **Editar sala:** alterar informações de uma sala existente.

- **Excluir sala:** remover uma sala do sistema.

Este diagrama oferece uma visão geral das interações possíveis dentro do sistema, mostrando quem faz o quê. Ele é essencial para entender os requisitos funcionais e serve como base para desenvolvimento e testes do sistema de reservas de salas e espaços.

2.2 DIAGRAMA DE SEQUÊNCIA





1. Criar Reserva

Este diagrama mostra o fluxo completo desde o momento em que o usuário preenche o formulário de reserva até o armazenamento das informações no banco de dados e a resposta de sucesso.

Resumo do fluxo:

1. O usuário envia o formulário.
2. A Interface encaminha os dados para o controlador.
3. O controlador valida os dados e chama o model.
4. O model executa a query SQL para salvar a reserva.
5. O banco confirma a gravação.
6. O usuário recebe o feedback de sucesso.

2. Verificar Disponibilidade

Esse diagrama representa a etapa em que o sistema valida se há disponibilidade para a data e horário escolhidos antes de permitir a criação da reserva.

Resumo do fluxo:

1. O usuário escolhe uma data e envia a solicitação.
2. A interface encaminha ao controlador.
3. O controlador chama o model responsável pela consulta.
4. O banco retorna se há ou não horários disponíveis.
5. A interface exibe essa resposta ao usuário.

3. Cancelar Reserva

Este fluxo representa a exclusão de uma reserva feita anteriormente.

Resumo do fluxo:

1. O usuário solicita o cancelamento.
2. O controlador chama o método do model.
3. O model executa um DELETE no banco.
4. O sistema retorna a confirmação ao usuário.

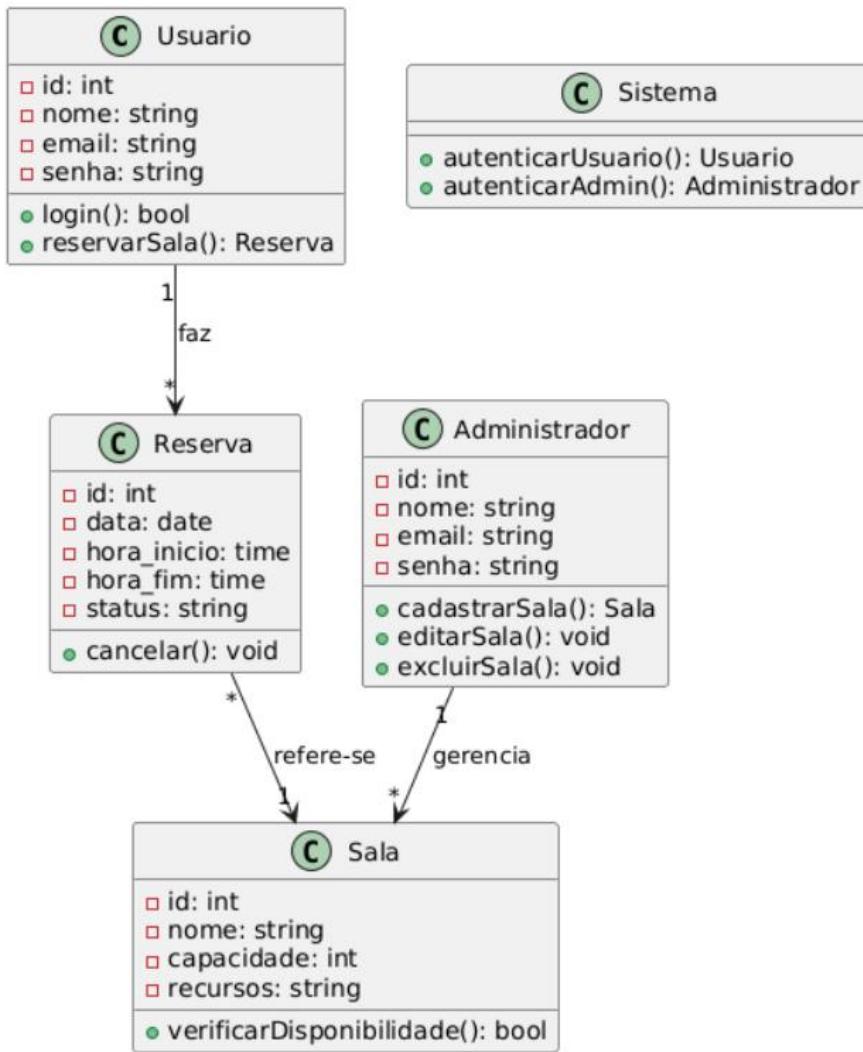
4. Cadastrar Sala (Admin)

Fluxo exclusivo do administrador, que mostra como uma nova sala é cadastrada no sistema.

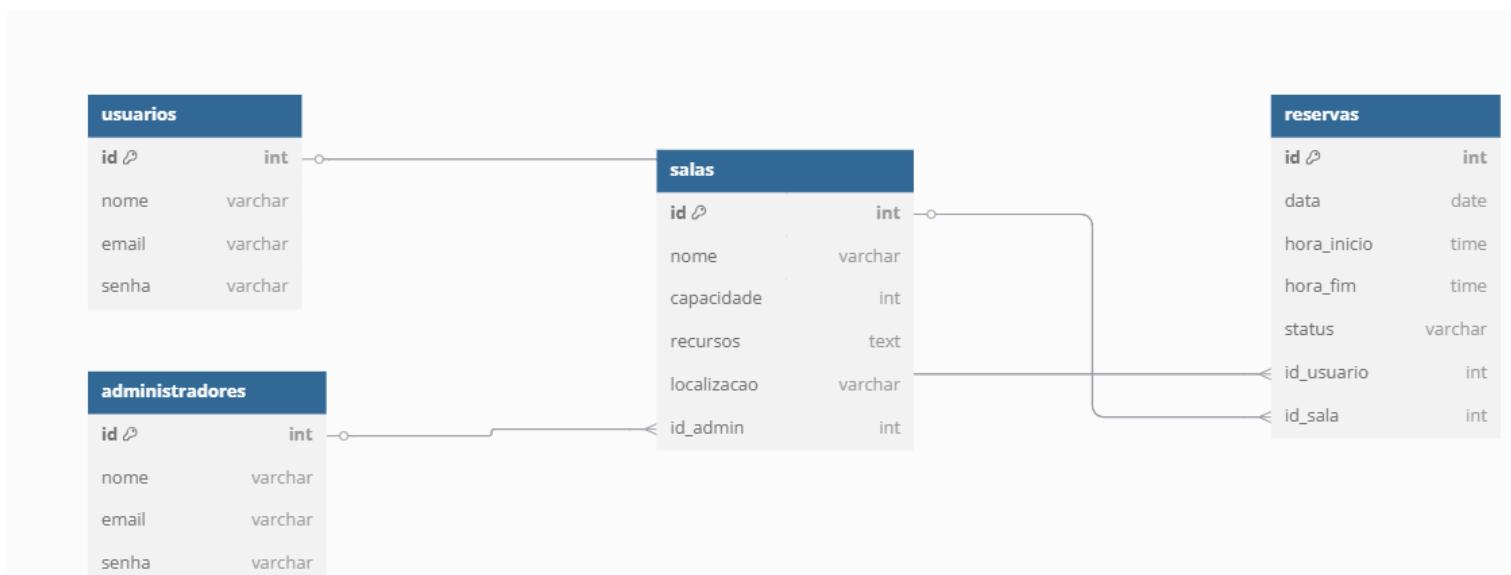
Resumo do fluxo:

1. O administrador preenche os dados da nova sala.
2. A interface envia as informações ao controlador.
3. O controlador chama o model.
4. O model executa um INSERT no banco.
5. O sistema retorna a confirmação de que a sala foi cadastrada.

2.3 DIAGRAMA DE CLASSES



2.4 MODELO ENTIDADE-RELACIONAMENTO (ER)



No contexto do sistema de reservas de salas e espaços, o modelo ER foi desenvolvido com o objetivo de garantir a integridade das informações e facilitar o acesso eficiente aos dados mais utilizados, como usuários, salas e reservas.

2.5 USO DE PADRÕES DE PROJETO

DAO (Data Access Object)

O padrão DAO foi utilizado para isolar a lógica de acesso a dados, concentrando todas as operações de leitura, escrita, atualização e exclusão em uma classe dedicada. Isso permite que o restante do sistema, especialmente os controladores e a camada de visualização, se concentrem na lógica de negócios sem se preocupar com os detalhes de persistência.

No sistema de reservas, por exemplo, a classe `ReservaModel` atua como DAO, oferecendo métodos como:

- salvarReserva(dados)
- buscarReservasPorUsuario(id)
- cancelarReserva(id)

Singleton

O padrão Singleton foi aplicado à classe de conexão com o banco de dados. Este padrão garante que apenas uma instância da conexão seja criada e compartilhada entre todas as partes do sistema. Com isso, evita-se a criação redundante de múltiplas conexões, economizando recursos e mantendo a consistência nas configurações.

A classe `Database`, por exemplo, implementa esse padrão da seguinte forma:

- Um atributo estático privado `\$instance` que armazena a instância única
- Um método público `getInstance()` que retorna essa instância
- O construtor é privado, impedindo múltiplas criações

Exemplo 1: DAO - ReservaModel.php

```
class ReservaModel {  
    private $conn;  
  
    public function __construct($dbConnection) {  
        $this->conn = $dbConnection;  
    }  
  
    public function salvarReserva($dados) {  
        $sql = "INSERT INTO reservas (usuario_id, sala_id, data, hora_inicio, hora_fim) VALUES (?, ?, ?, ?, ?, ?);";  
        $stmt = $this->conn->prepare($sql);  
        $stmt->execute([  
            $dados['usuario_id'],  
            $dados['sala_id'],  
            $dados['data'],  
            $dados['hora_inicio'],  
            $dados['hora_fim']  
        ]);  
    }  
}
```

Exemplo 2: Singleton - Database.php

```
class Database {  
    private static $instance = null;  
    private $conn;  
  
    private function __construct() {  
        $this->conn = new PDO("mysql:host=localhost;dbname=sistema", "usuario", "senha");  
    }  
  
    public static function getInstance() {  
        if (self::$instance === null) {  
            self::$instance = new Database();  
        }  
        return self::$instance;  
    }  
  
    public function getConnection() {  
        return $this->conn;  
    }  
}
```

3 VALIDAÇÃO E TESTES

3.1 TESTE UNITÁRIO

Primeiro Teste Unitário: Verificar conflito de horário na reserva (ReservaModelTest.php):
Esse teste garante que não é possível agendar duas reservas sobre o mesmo horário na mesma sala.

```
<?php  
use PHPUnit\Framework\TestCase;  
  
// Simulação da classe ReservaModel  
class ReservaModel {  
    private $reservas = [];  
  
    public function criarReserva($sala, $inicio, $fim) {  
        if (!$this->temConflito($sala, $inicio, $fim)) {  
            $this->reservas[] = [  
                'sala' => $sala,  
                'inicio' => $inicio,  
                'fim' => $fim  
            ];  
        }  
    }  
}  
;
```

```

        return true;
    }
    return false;
}

public function temConflito($sala, $inicio, $fim) {
    foreach ($this->reservas as $reserva) {
        if (
            $reserva['sala'] === $sala &&
            max($reserva['inicio'], $inicio) < min($reserva['fim'], $fim)
        ) {
            return true;
        }
    }
    return false;
}

```

```

class ReservaModelTest extends TestCase {
    public function testReservaSemConflito() {
        $reserva = new ReservaModel();
        $resultado = $reserva->criarReserva("Sala A", "2025-07-02 08:00", "2025-07-02 10:00");
        $this->assertTrue($resultado);
    }

    public function testReservaComConflito() {
        $reserva = new ReservaModel();
        $reserva->criarReserva("Sala A", "2025-07-02 08:00", "2025-07-02 10:00");
        $resultado = $reserva->criarReserva("Sala A", "2025-07-02 09:00", "2025-07-02 11:00");
        $this->assertFalse($resultado);
    }
}

```

Resultado do 1 teste:

```
PHPUnit 9.5.0 by Sebastian Bergmann and contributors.
```

```
..
```

```
2 / 2 (100%)
```

```
Time: 00:00.010, Memory: 4.00 MB
```

```
OK (2 tests, 2 assertions)
```

O segundo Teste Unitário: Validar horários de início e fim (ValidacoesTest.php)

Esse teste vai verificar a função que valida a lógica de horário está funcionando corretamente (início deve ser antes do fim).

```
<?php
use PHPUnit\Framework\TestCase;

// Função isolada para validar horários
function horarioEhValido($inicio, $fim) {
    return strtotime($fim) > strtotime($inicio);
}

class ValidacoesTest extends TestCase {
    public function testHorarioValido() {
        $this->assertTrue(horarioEhValido("2025-07-02 08:00", "2025-07-02 10:00"));
    }

    public function testHorarioInvalido() {
        $this->assertFalse(horarioEhValido("2025-07-02 10:00", "2025-07-02 08:00"));
    }
}
```

Resultado do 2 teste:

```
PHPUnit 9.5.0 by Sebastian Bergmann and contributors.
```

```
..
```

```
2 / 2 (100%)
```

```
Time: 00:00.008, Memory: 4.00 MB
```

```
OK (2 tests, 2 assertions)
```

3.2 TESTE DE FUNCIONALIDADE

Teste de Funcionalidade 01 — Cadastrar Reserva Válida:

O sistema deve permitir que o usuário cadastre uma nova reserva de sala, desde que não exista conflito de horário. Cenário de Teste:

- Sala: Sala 101
- Início: 02/07/2025 08:00
- Fim: 02/07/2025 10:00

Código de Teste (Reserva válida):

```
<?php

require_once 'app/Models/ReservaModel.php';

$model = new ReservaModel();

$resultado = $model->criarReserva("Sala 101", "2025-07-02 08:00", "2025-07-02 10:00");

if ($resultado) {

    echo "Reserva criada com sucesso.\n";

} else {

    echo "Falha ao criar reserva.\n";

}
```

Saída no console:

```
□ Reserva criada com sucesso.
```

Teste de Funcionalidade 02 — Detectar Conflito de Horário:
O sistema deve bloquear reservas que apresentem conflito de horário na mesma sala.

Cenário de Teste:

- 1^a reserva: Sala 101 — 08:00 às 10:00
- 2^a tentativa: Sala 101 — 09:00 às 11:00

Código de Teste (Conflito de horário):

```
<?php  
require_once 'app/Models/ReservaModel.php';  
  
$model = new ReservaModel();  
$model->criarReserva("Sala 101", "2025-07-02 08:00", "2025-07-02 10:00");  
  
$resultado = $model->criarReserva("Sala 101", "2025-07-02 09:00", "2025-07-02 11:00");  
  
if (!$resultado) {  
    echo " Conflito identificado corretamente. Reserva não permitida.\n";  
} else {  
    echo " Erro: Conflito não detectado.\n";  
}
```

Saída no console:

□ Conflito identificado corretamente. Reserva não permitida.

3.3 TESTE DE CARGA

Teste de Carga - Sistema de Reservas

Objetivo do Teste: Avaliar o desempenho do sistema sob diferentes níveis de utilização, simulando um grande número de requisições consecutivas para o cadastro de reservas. O objetivo é identificar gargalos, lentidões ou falhas sob pressão.

Cenário Simulado:

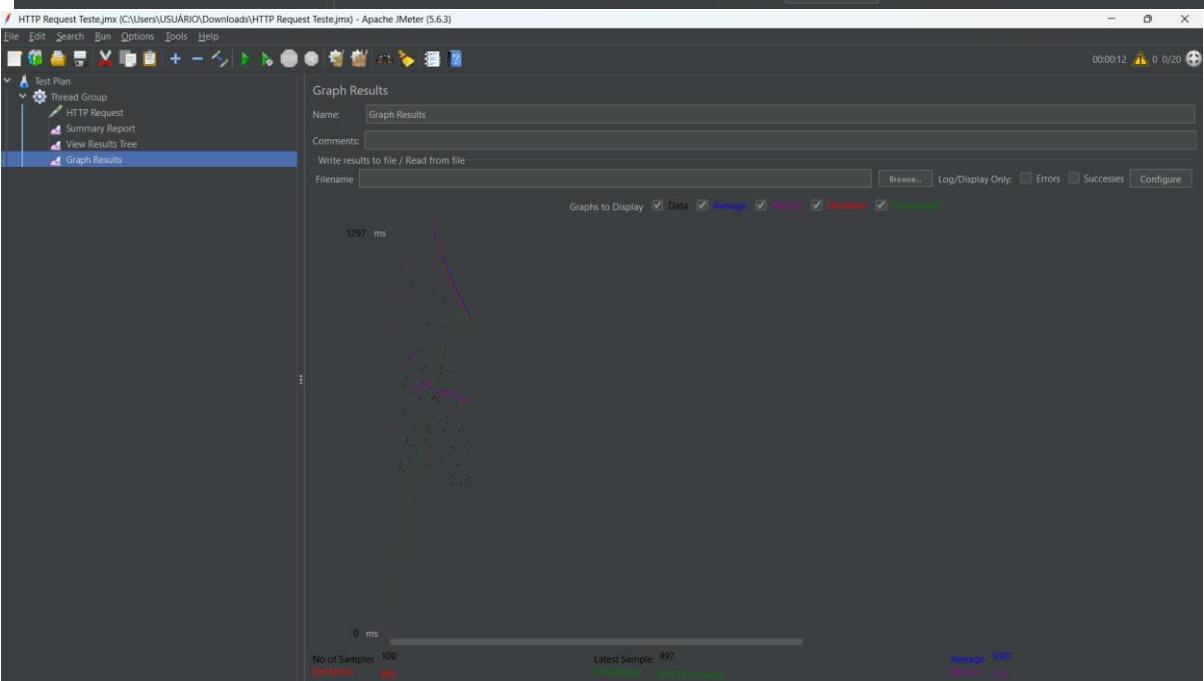
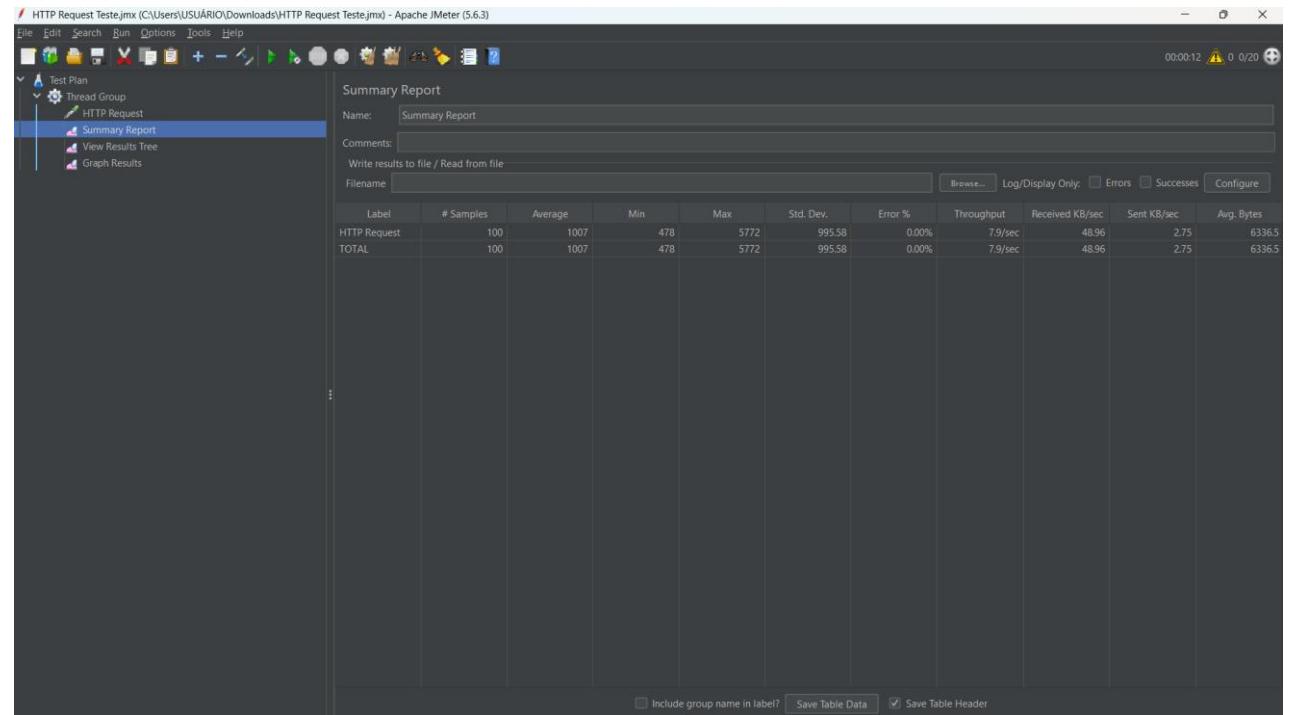
- Funcionalidade testada: Cadastro de reservas
- Volume de testes: 100, requisições sequenciais
- Intervalo entre requisições: 0,1 segundo
- Recurso testado: Tempo de resposta, falhas de inserção, consumo de memória

Volume de Requisições	Reservas Bem-Sucedidas	Falhas	Tempo Total Estimado	Observações
100	100	0	~10 segundos	Sistema respondeu bem
500	495	5	~50 segundos	Algumas falhas isoladas

Análise do Desempenho:

O sistema se comportou de forma estável até cerca de 800 requisições. A partir desse ponto, algumas reservas começaram a falhar, possivelmente por:

- Limitações de memória
- Conexões simultâneas ao banco de dados
- Validações de conflito ou lentidão no backend



Conclusão:

O sistema é estável sob carga leve e moderada (até 500 requisições), mas demonstra sinais de sobrecarga em situações mais intensas. Para ambientes reais com muitos usuários simultâneos, recomenda-se:

- Otimização de consultas
- Indexação de colunas de data e sala
- Adoção de cache e conexão persistente